Kostenloses eBook

LERNEN

C Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#c

Inhaltsverzeichnis

Über 1

Kapitel 1: Erste Schritte mit C Language 2

Bemerkungen 2

Gemeinsame Compiler 2

Compiler C-Version Unterstützung 2

Code-Stil (Off-Topic hier): 3

Bibliotheken und APIs, die nicht durch den C-Standard abgedeckt sind (und daher hier außer 4

Versionen 4

Examples 4

Hallo Welt 4

Hallo c 4

Schauen wir uns dieses einfache Programm Zeile für Zeile an 5

Programm bearbeiten 6

Programm kompilieren und ausführen 6

Mit GCC kompilieren 6

Verwendung des Clang-Compilers 6

Verwenden des Microsoft C-Compilers über die Befehlszeile 7

Programm ausführen 7

Original "Hallo, Welt!" in K & R C 7

Kapitel 2: - Klassifizierung und Konvertierung von Zeichen 9

Examples 9

Klassifizieren von Zeichen, die aus einem Stream gelesen wurden 9

Zeichen aus einer Zeichenfolge klassifizieren 9

Einführung 10

Kapitel 3: Aliasing und effektiver Typ 14

Bemerkungen 14

Examples 15

Auf Zeichentypen kann nicht über Nicht-Zeichentypen zugegriffen werden. 15

Effektiver Typ 16

Verletzung der strengen Aliasing-Regeln 17

Qualifikation einschränken 17
Bytes ändern 18
Kapitel 4: Allgemeine C-Programmiersprachen und Entwicklerpraktiken 20
Examples 20
Vergleich von Literal und Variable 20
Lassen Sie die Parameterliste einer Funktion nicht leer - verwenden Sie void 20
Kapitel 5: Arrays 24
Einführung 24
Syntax 24
Bemerkungen 24
Examples 25
Array deklarieren und initialisieren 25
Löschen des Array-Inhalts (Nullstellen) 26
Arraylänge 27
Einstellungswerte in Arrays 28
Definieren Sie Array- und Access-Array-Elemente 29
Ordnen Sie ein Array mit benutzerdefinierter Größe zu und initialisieren Sie es mit Null 30
Iteration durch ein Array effizient und Reihenreihenfolge 30
Mehrdimensionale Arrays 32
Durchlaufen eines Arrays mit Zeigern 35
Übergeben von mehrdimensionalen Arrays an eine Funktion 35
Siehe auch 36
Kapitel 6: Atomik 37
Syntax 37
Bemerkungen 37
Examples 38
Atomik und Operatoren 38
Kapitel 7: Aufzählungen 39
Bemerkungen 39
Examples 39
Einfache Aufzählung 39
Beispiel 1 39

Beispiel 2 ... 40
Typedef-Enum ... 41
Aufzählung mit doppeltem Wert ... 42
Aufzählungskonstante ohne Typenname ... 42
Kapitel 8: Auswahlanweisungen ... 44
Examples ... 44
if () Anweisungen ... 44
if () ... else Anweisungen und Syntax ... 45
switch () Anweisungen ... 45
if () ... else Ladder Chaining zwei oder mehr if () ... else-Anweisungen ... 47
Geschachtelte if () ... else VS if () .. else Ladder ... 48
Kapitel 9: Behauptung ... 50
Einführung ... 50
Syntax ... 50
Parameter ... 50
Bemerkungen ... 50
Examples ... 51
Vorbedingung und Nachbedingung ... 51
Einfache Assertion ... 52
Statische Assertion ... 53
Geltendmachung eines unerreichbaren Codes ... 53
Bestätigen Sie Fehlermeldungen ... 54
Kapitel 10: Bemerkungen ... 56
Einführung ... 56
Syntax ... 56
Examples ... 56
/ * * / abgegrenzte Kommentare ... 56
// abgegrenzte Kommentare ... 57
Kommentieren mit dem Präprozessor ... 57
Möglicher Fallstrick durch Trigraphen ... 58
Kapitel 11: Bitfelder ... 59
Einführung ... 59

Syntax 59
Parameter 59
Bemerkungen 59
Examples 59
Bitfelder 59
Verwenden von Bitfeldern als kleine Ganzzahlen 61
Bitfeldausrichtung 61
Wann sind Bitfelder nützlich? 62
Don'ts für Bitfelder 63
Kapitel 12: Boolean 65
Bemerkungen 65
Examples 65
Stdbool.h verwenden 65
#Define verwenden 65
Verwendung des eingebauten (integrierten) Typs _Bool 66
Ganzzahlen und Zeiger in booleschen Ausdrücken. 66
Definieren eines Bool-Typs mit Typedef 67
Kapitel 13: Dateien und E / A-Streams 69
Syntax 69
Parameter 69
Bemerkungen 69
Moduszeichenfolgen: 69
Examples 70
Öffnen Sie und schreiben Sie in die Datei 70
fprintf 71
Prozess ausführen 72
Zeilen mit getline () aus einer Datei holen 72
Eingabedatei example.txt 73
Ausgabe 74
Beispielimplementierung von getline() 74
Öffnen und schreiben Sie in eine Binärdatei 76
fscanf () 77

Zeilen aus einer Datei lesen 78
Kapitel 14: Datentypen 81
Bemerkungen 81
Examples 81
Integer-Typen und Konstanten 81
String Literals 83
Integer-Typen mit fester Breite (seit C99) 84
Fließkommakonstanten 84
Erklärungen interpretieren 85
Beispiele 86
Mehrere Erklärungen 86
Alternative Interpretation 87
Kapitel 15: Deklaration vs. Definition 88
Bemerkungen 88
Examples 88
Erklärung und Definition verstehen 88
Kapitel 16: Einschränkungen 90
Bemerkungen 90
Examples 90
Doppelte Variablennamen im selben Bereich 90
Unäre arithmetische Operatoren 91
Kapitel 17: Erklärungen 92
Bemerkungen 92
Examples 92
Aufruf einer Funktion aus einer anderen C-Datei 92
Verwendung einer globalen Variablen 93
Globale Konstanten verwenden 94
Einführung 96
Typedef 99
Verwenden der Rechts-Links- oder Spiralregel zum Entschlüsseln der C-Deklaration 99
Kapitel 18: Erstellen Sie Header-Dateien und fügen Sie sie ein 105

Einführung 105
Examples 105
Einführung 105
Idempotenz 106
Kopfschutz 106
Die #pragma once Direktive 106
Selbstbeherrschung 107
Empfehlung: Header-Dateien sollten in sich geschlossen sein. 107
Historische Regeln 107
Moderne Regeln 107
Selbstbeherrschung prüfen 108
Minimalität 109
Einschließen, was Sie verwenden (IWYU) 109
Notation und Verschiedenes 110
Querverweise 111
Kapitel 19: Fehlerbehandlung 112
Syntax 112
Bemerkungen 112
Examples 112
errno 112
Strerror 112
perror 113
Kapitel 20: Folgepunkte 114
Bemerkungen 114
Examples 115
Sequenzierte Ausdrücke 115
Nicht aufeinanderfolgende Ausdrücke 115
Unbestimmte sequenzierte Ausdrücke 116
Kapitel 21: Formatierte Eingabe / Ausgabe 118
Examples 118
Den Wert eines Zeigers auf ein Objekt drucken 118
Verwenden von <inttypes.h> und uintptr_t 118

Vorgeschichte: 119
Drucken der Differenz der Werte zweier Zeiger auf ein Objekt 119
Konvertierungsspezifikationen für den Druck 120
Die Funktion printf () 122
Längenmodifikatoren 122
Formatflags drucken 124
Kapitel 22: Frameworks testen 126
Einführung 126
Bemerkungen 126
Examples 126
CppUTest 126
Unity Test Framework 127
CMocka 128
Kapitel 23: Funktionsparameter 130
Bemerkungen 130
Examples 130
Verwenden von Zeigerparametern, um mehrere Werte zurückzugeben 130
Übergabe von Arrays an Funktionen 130
Siehe auch 131
Parameter werden per Wert übergeben 131
Reihenfolge der Funktionsparameterausführung 132
Beispiel für eine Funktion, die eine Struktur zurückgibt, die Werte mit Fehlercodes enthäl 132
Kapitel 24: Funktionszeiger 135
Einführung 135
Syntax 135
Examples 135
Funktionszeiger zuweisen 135
Funktionszeiger von einer Funktion zurückgeben 136
Best Practices 136
Mit typedef 136
Beispiel: 137
Kontext-Zeiger nehmen 138

Beispiel.....138

Siehe auch.....138

Einführung.....138

Verwendungszweck.....139

Syntax.....139

Mnemonik zum Schreiben von Funktionszeigern.....139

Grundlagen.....140

Kapitel 25: Generische Auswahl.....142

Syntax.....142

Parameter.....142

Bemerkungen.....142

Examples.....142

Prüfen Sie, ob eine Variable einen bestimmten qualifizierten Typ hat.....142

Typisches generisches Druckmakro.....143

Generische Auswahl basierend auf mehreren Argumenten.....143

Kapitel 26: Gewerkschaften.....146

Examples.....146

Unterschied zwischen struct und union.....146

Verwendung von Vereinigungen, um Werte neu zu interpretieren.....146

Einem Gewerkschaftsmitglied schreiben und von einem anderen lesen.....147

Kapitel 27: Häufige Fehler.....149

Einführung.....149

Examples.....149

Mischen von vorzeichenbehafteten und vorzeichenlosen Ganzzahlen in arithmetischen Operatio.....149

Beim Vergleichen fälschlicherweise = anstelle von ==.....150

Unvorsichtige Verwendung von Semikolons.....151

Vergessen, ein zusätzliches Byte für \ 0 zuzuweisen.....152

Vergessen, Speicher freizugeben (Speicherverluste).....152

Zu viel kopieren.....154

Vergessen, den Rückgabewert von realloc in einen temporären Wert zu kopieren.....154

Vergleich von Fließkommazahlen.....155

Zusätzliche Skalierung in der Zeigerarithmetik.....156

Makros sind einfache Zeichenfolgen....157
Undefinierte Referenzfehler beim Linken....159
Missverständnis des Array-Zerfalls....162
Übergeben nicht benachbarter Arrays an Funktionen, die "echte" mehrdimensionale Arrays erw....164
Verwenden Sie Zeichenkonstanten anstelle von String-Literalen und umgekehrt....165
Rückgabewerte von Bibliotheksfunktionen ignorieren....166
Bei einem typischen Aufruf von scanf () wird kein Zeilenvorschubzeichen verwendet....167
Hinzufügen eines Semikolons zu einer # Definition....168
Mehrzeilige Kommentare können nicht geschachtelt werden....169
Array-Grenzen überschreiten....171
Rekursive Funktion - Die Basisbedingung wird übersehen....172
Logischen Ausdruck gegen 'true' prüfen....173
Fließkomma-Literale sind standardmäßig vom Typ double....174
Kapitel 28: Identifier-Bereich....175
Examples....175
Bereich blockieren....175
Funktionsprototyp-Umfang....175
Dateibereich....176
Funktionsumfang....177
Kapitel 29: Implementierungsdefiniertes Verhalten....179
Bemerkungen....179
Überblick....179
Programme und Prozessoren....179
Allgemeines....179
Quellübersetzung....180
Betriebsumgebung....180
Typen....181
Quellformular....182
Auswertung....182
Laufzeitverhalten....182
Präprozessor....183
Standardbibliothek....183

Allgemeines 184
Gleitkomma-Umgebungsfunktionen 184
Gebietsschema-bezogene Funktionen 184
Mathematische Funktionen 184
Signale 185
Verschiedenes 185
Dateibehandlungsfunktionen 185
E / A-Funktionen 186
Speicherbelegungsfunktionen 186
Systemumgebungsfunktionen 186
Datums- und Zeitfunktionen 187
Wide-Character-E / A-Funktionen 187
Examples 187
Rechtsverschiebung einer negativen Ganzzahl 187
Einer Ganzzahl einen Wert außerhalb des Bereichs zuweisen 188
Zuweisung von null Bytes 188
Darstellung von signierten ganzen Zahlen 188
Kapitel 30: Implizite und explizite Konvertierungen 189
Syntax 189
Bemerkungen 189
Examples 189
Ganzzahlkonvertierungen in Funktionsaufrufen 189
Zeigerkonvertierungen in Funktionsaufrufen 190
Kapitel 31: Initialisierung 192
Examples 192
Initialisierung von Variablen in C 192
Initialisieren von Strukturen und Arrays von Strukturen 194
Verwenden von festgelegten Initialisierern 194
Designierte Initialisierer für Array-Elemente 194
Designierte Initialisierer für Strukturen 195
Designierter Initialisierer für Gewerkschaften 195
Designierte Initialisierer für Arrays von Strukturen usw 196

Angeben von Bereichen in Array-Initialisierern 196

Kapitel 32: Inline-Montage 198

Bemerkungen 198

Pros 198

Cons 198

Examples 198

gcc Basic ASM-Unterstützung 198

gcc Extended ASM-Unterstützung 199

gcc Inline-Montage in Makros 200

Kapitel 33: Inlining 202

Examples 202

Inlining-Funktionen, die in mehreren Quelldateien verwendet werden 202

Haupt c: 202

source1.c: 202

source2.c: 202

Headerdatei.h: 203

Kapitel 34: Iterationsanweisungen / -schleifen: für, während, währenddessen 205

Syntax 205

Bemerkungen 205

Kopfgesteuerte Iterationsanweisung / -schleifen 205

Fußgesteuerte Iterationsanweisung / -schleifen 205

Examples 205

Für Schleife 205

While-Schleife 206

Do-While-Schleife 206

Struktur und Ablauf der Kontrolle in einer for-Schleife 207

Unendliche Schleifen 208

Endlosschleifen und Duff's Device 209

Kapitel 35: Kommandozeilenargumente 211

Syntax 211

Parameter 211

Bemerkungen 211

Examples 212
Befehlszeilenargumente drucken 212
Drucken Sie die Argumente an ein Programm und konvertieren Sie sie in Ganzzahlwerte 213
Verwenden von GNU-getopt-Werkzeugen 213
Kapitel 36: Literale für Zahlen, Zeichen und Zeichenfolgen 217
Bemerkungen 217
Examples 217
Ganzzahlige Literale 217
String-Literale 218
Fließkomma-Literale 218
Zeichenliterale 219
Kapitel 37: Multithreading 221
Einführung 221
Syntax 221
Bemerkungen 221
Examples 221
C11 Threads einfaches Beispiel 221
Kapitel 38: Nebenwirkungen 223
Examples 223
Operatoren vor / nach Inkrementieren / Verringern 223
Kapitel 39: Operatoren 225
Einführung 225
Syntax 225
Bemerkungen 225
Examples 227
Beziehungsoperatoren 227
Gleich "==" 227
Nicht gleich "! =" 227
Nicht "!" 228
Größer als ">" 228
Weniger als "<" 228

Größer oder gleich "> =" 228
Weniger oder gleich "<=" 228
Zuweisungsoperatoren 229
Rechenzeichen 230
Grundrechenarten 230
Zusatzoperator 230
Subtraktionsoperator 231
Multiplikationsoperator 231
Bereichsoperator 232
Modulo Operator 232
Inkrement- / Dekrementoperatoren 233
Logische Operatoren 233
Logisches UND 233
Logisches ODER 234
Logisch NICHT 234
Kurzschlussbewertung 234
Inkrement / Dekrement 235
Bedingter Operator / Ternärer Operator 235
Komma-Operator 236
Cast Operator 237
sizeof Operator 237
Mit einem Typ als Operand 237
Mit einem Ausdruck als Operand 237
Zeigerarithmetik 238
Zeigerzusatz 238
Zeigerabzug 239
Zugriffsoperatoren 239
Mitglied des Objekts 239
Mitglied des Objektes, auf das verwiesen wird 239
Adresse von 240
Dereferenz 240
Indizierung 240

Austauschbarkeit der Indexierung ... 241
Operator für Funktionsaufruf ... 241
Bitweise Operatoren ... 241
_Alignof ... 243
Kurzschlussverhalten logischer Operatoren ... 244
Kapitel 40: Preprozessor und Makros ... 246
Einführung ... 246
Bemerkungen ... 246
Examples ... 246
Bedingter Einschluss und Änderung der bedingten Funktionssignatur ... 246
Quelldatei-Aufnahme ... 249
Makro-Ersatz ... 249
Fehleranweisung ... 250
#if 0, um Codeabschnitte auszublenden ... 251
Token-Einfügen ... 252
Vordefinierte Makros ... 252
Obligatorische vordefinierte Makros ... 252
Andere vordefinierte Makros (nicht obligatorisch) ... 253
Header Include Guards ... 254
FOREACH-Implementierung ... 257
__cplusplus für die Verwendung von C-Externen in C ++ - Code, zusammengestellt mit C ++ - ... 260
Funktionsartige Makros ... 261
Variadische Argumente Makro ... 262
Kapitel 41: Prozessübergreifende Kommunikation (IPC) ... 265
Einführung ... 265
Examples ... 265
Semaphore ... 265
Beispiel 1.1: Rennen mit Fäden ... 266
Beispiel 1.2: Vermeiden Sie das Rennen mit Semaphoren ... 267
Kapitel 42: Signalverarbeitung ... 270
Syntax ... 270
Parameter ... 270

Bemerkungen 270

Examples 271

Signalbehandlung mit "Signal ()" 271

Kapitel 43: Speicherklassen 274

Einführung 274

Syntax 274

Bemerkungen 274

Lagerdauer 275

Statische Speicherdauer 276

Thread-Speicherdauer 276

Automatische Speicherdauer 276

Externe und interne Verknüpfung 276

Examples 277

Typedef 277

Auto 277

statisch 278

extern 279

registrieren 280

_Thread_local 281

Kapitel 44: Speicherverwaltung 282

Einführung 282

Syntax 282

Parameter 282

Bemerkungen 282

Examples 283

Speicher freigeben 283

Speicher zuweisen 284

Standardzuteilung 284

Nullspeicher 285

Ausgerichteter Speicher 286

Speicher neu zuordnen 286

Mehrdimensionale Arrays mit variabler Größe 287
Realloc (Ptr, 0) ist nicht gleich "Free" (Ptr) 288
Benutzerdefinierte Speicherverwaltung 289
Allocationa: Speicher auf Stack zuweisen 290
Zusammenfassung 291
Empfehlung 291
Kapitel 45: Sprunganweisungen 292
Syntax 292
Bemerkungen 292
Siehe auch 292
Examples 292
Mit goto aus verschachtelten Schleifen springen 292
Verwenden Sie Zurück 293
Rückgabe eines Wertes 293
Nichts zurückgeben 294
Verwenden Sie break und fahren Sie fort 294
Kapitel 46: Standard Math 296
Syntax 296
Bemerkungen 296
Examples 296
Fließkomma-Rest mit doppelter Genauigkeit: fmod () 296
Gleitkomma-Rest mit einfacher Genauigkeit und langer doppelter Genauigkeit: fmodf (), fmod 297
Power-Funktionen - pow (), powf (), powl () 298
Kapitel 47: Structs 300
Einführung 300
Examples 300
Einfache Datenstrukturen 300
Typedef-Strukturen 300
Verweise auf structs 302
Flexible Array-Mitglieder 304
Typenerklärung 304
Auswirkungen auf Größe und Polsterung 305

Verwendungszweck 305
Der "struct hack" 306
Kompatibilität 306
Übergabe von Strukturen an Funktionen 307
Objektbasierte Programmierung mit Strukturen 308
Kapitel 48: Strukturpolsterung und Verpackung 311
Einführung 311
Bemerkungen 311
Examples 311
Verpackungsstrukturen 311
Strukturverpackung 312
Strukturpolsterung 312
Kapitel 49: Themen (einheimisch) 314
Syntax 314
Bemerkungen 314
C-Bibliotheken, von denen bekannt ist, dass sie C11-Threads unterstützen, sind: 314
C-Bibliotheken, die noch keine C11-Threads unterstützen: 314
Examples 315
Starten Sie mehrere Threads 315
Initialisierung durch einen Thread 315
Kapitel 50: Typ Qualifiers 317
Bemerkungen 317
Qualifikationen auf höchstem Niveau 317
Zeigertyp-Qualifikationen 317
Examples 318
Unveränderbare (const) Variablen 318
Warnung 318
Flüchtige Variablen 319
Kapitel 51: Typedef 321
Einführung 321
Syntax 321

Bemerkungen 321
Examples 322
Typedef für Strukturen und Vereinigungen 322
Einfache Verwendung von Typedef 323
Um einem Datentyp kurze Namen zu geben 323
Portabilität verbessern 323
Zur Angabe einer Verwendung oder zur Verbesserung der Lesbarkeit 324
Typedef für Funktionszeiger 324
Kapitel 52: Übergeben Sie 2D-Arrays an Funktionen 327
Examples 327
Übergeben Sie ein 2D-Array an eine Funktion 327
Verwenden von flachen Arrays als 2D-Arrays 333
Kapitel 53: Undefiniertes Verhalten 335
Einführung 335
Bemerkungen 335
Examples 337
Dereferenzieren Sie einen Nullzeiger 337
Objekte mehr als einmal zwischen zwei Sequenzpunkten ändern 337
Fehlende return-Anweisung in der Funktion zur Wertrückgabe 338
Überlauf der signierten Ganzzahl 339
Verwendung einer nicht initialisierten Variablen 340
Dereferenzieren eines Zeigers auf eine Variable über ihre Lebensdauer hinaus 341
Durch Null teilen 342
Zugriff auf Speicherplatz außerhalb des zugewiesenen Blocks 343
Überlappenden Speicher kopieren 343
Lesen eines nicht initialisierten Objekts, das nicht durch den Speicher gesichert wird 344
Datenrennen 345
Lese den Wert des freigegebenen Zeigers 346
Ändern Sie das String-Literal 346
Speicherplatz zweimal freigeben 347
Verwendung eines falschen Formatbezeichners in printf 347
Die Konvertierung zwischen Zeigertypen führt zu einem falsch ausgerichteten Ergebnis 348

Addition oder Subtraktion des Zeigers nicht richtig begrenzt....348
Eine const-Variable mit einem Zeiger ändern....349
Übergabe eines Nullzeigers an die Konvertierung von printf% s....349
Inkonsistente Verknüpfung von Bezeichnern....350
Fflush für einen Eingabestrom verwenden....351
Bitverschiebung mit negativen Zählwerten oder über die Breite des Typs hinaus....351
Ändern der Zeichenfolge, die von den Funktionen getenv, strerror und setlocale zurückgegeb....352
Rückkehr von einer Funktion, die mit dem _ _Noreturn`- oder `Noreturn'-Funktionsbezeichner....353
Kapitel 54: Valgrind....355
Syntax....355
Bemerkungen....355
Examples....355
Valgrind laufen lassen....355
Flaggen hinzufügen....355
Bytes verloren - vergessen vergessen....355
Die häufigsten Fehler bei der Verwendung von Valgrind....356
Kapitel 55: Variable Argumente....358
Einführung....358
Syntax....358
Parameter....358
Bemerkungen....358
Examples....359
Verwenden eines expliziten count-Arguments, um die Länge der va_list zu bestimmen....359
Terminatorwerte verwenden, um das Ende von va_list zu bestimmen....360
Funktionen mit einer `printf ()` -ähnlichen Oberfläche implementieren....361
Verwenden einer Formatzeichenfolge....364
Kapitel 56: Verknüpfte Listen....366
Bemerkungen....366
Einfach verknüpfte Liste....366
Datenstruktur....366
Doppelt verknüpfte Liste....366
Datenstruktur....366

Topoliges 366
Linear oder offen 366
Kreisförmig oder ring 367
Verfahren 367
Binden 367
Erstellen einer kreisförmig verbundenen Liste 367
Erstellen einer linear verknüpften Liste 368
Einfügung 368
Examples 369
Einfügen eines Knotens am Anfang einer einfach verknüpften Liste 369
Erläuterung zum Einfügen von Knoten 370
Einfügen eines Knotens an der n-ten Position 371
Umkehrung einer verknüpften Liste 372
Erklärung für die Reverse List-Methode 373
Eine doppelt verknüpfte Liste 374
Kapitel 57: X-Makros 378
Einführung 378
Bemerkungen 378
Examples 378
Trivialer Einsatz von X-Makros für printfs 378
Aufzählungswert und Bezeichner 379
Erweiterung: Geben Sie das X-Makro als Argument an 379
Codegenerierung 380
Hier verwenden wir X-Makros, um eine Aufzählung mit 4 Befehlen und eine Zuordnung ihrer Na 380
In ähnlicher Weise können wir eine Sprungtabelle erzeugen, um Funktionen anhand des Aufzäh 381
Kapitel 58: Zeichenfolge mit mehreren Zeichen 382
Bemerkungen 382
Examples 382
Trigraphen 382
Digraphen 383
Kapitel 59: Zeichenketten 385
Einführung 385

Syntax 385
Examples 385
Länge berechnen: strlen () 385
Kopieren und Verketten: strcpy (), strcat () 386
Vergleich: strcmp (), strncmp (), strcasecmp (), strncasecmp () 387
Tokenisierung: strtok (), strtok_r () und strtok_s () 389
Finden Sie das erste / letzte Vorkommen eines bestimmten Zeichens: strchr (), strrchr () 391
Iteration über die Zeichen in einer Zeichenfolge 393
Grundlegende Einführung in Strings 393
Erstellen von Arrays von Strings 394
strstr 395
String-Literale 396
Einen String auf Null setzen 397
strspn und strcspn 398
Zeichenketten kopieren 399
Zeigerzuweisungen kopieren keine Zeichenfolgen 399
Zeichenketten mit Standardfunktionen kopieren 400
strcpy() 400
snprintf() 400
strncat() 401
strncpy() 401
Konvertieren Sie Strings in Number: atoi (), atof () (gefährlich, verwenden Sie sie nicht) 402
String formatierte Daten lesen / schreiben 403
Strings in Number: strtoX-Funktionen sicher konvertieren 404
Kapitel 60: Zeiger 406
Einführung 406
Syntax 406
Bemerkungen 406
Examples 406
Häufige Fehler 406
Nicht auf Zuordnungsfehler prüfen 407
Verwenden Sie beim Anfordern von Speicher Literalzahlen anstelle von sizeof 407

Speicherlecks 408
Logische Fehler 408
Erstellen von Zeigern zum Stapeln von Variablen 408
Inkrementieren / Dekrementieren und Dereferenzieren 410
Dereferenzieren eines Zeigers 410
Dereferenzieren eines Zeigers auf eine Struktur 411
Funktionszeiger 412
Siehe auch 413
Zeiger initialisieren 413
Vorsicht: 414
Vorsicht: 414
Adresse des Betreibers (&) 414
Zeigerarithmetik 415
void * -Punkte als Argumente und geben Werte an Standardfunktionen zurück 415
Const-Zeiger 415
Einzelzeiger 416
Zeiger auf Zeiger 416
Gleicher Stern, unterschiedliche Bedeutungen 418
Prämisse 418
Beispiel 418
Fazit 419
Zeiger auf Zeiger 419
Einführung 420
Zeiger und Arrays 422
Polymorphes Verhalten mit leeren Zeigern 423
Kapitel 61: Zufallszahlengenerierung 425
Bemerkungen 425
Examples 425
Generelle Zufallszahlengenerierung 425
Permutierter kongruentieller Generator 426
Beschränken Sie die Erzeugung auf einen bestimmten Bereich 427
Xorshift Generation 427

Kapitel 62: Zusammengesetzte Literale 429
Syntax 429
Bemerkungen 429
Examples 429
Definition / Initialisierung von zusammengesetzten Literalen 429
Beispiele aus dem C-Standard C11-§6.5.2.5 / 9: 429
Zusammengesetztes Literal mit Bezeichnern 430
Zusammengesetztes Literal ohne Angabe der Arraylänge 430
Zusammengesetztes Literal, dessen Initialisierungslänge weniger als die angegebene Arraygr 431
Schreibgeschütztes zusammengesetztes Literal 431
Zusammengesetztes Literal mit beliebigen Ausdrücken 431
Kapitel 63: Zusammenstellung 432
Einführung 432
Bemerkungen 432
Examples 433
Der Linker 433
Implizites Aufrufen des Linkers 434
Expliziter Aufruf des Linkers 434
Optionen für den Linker 434
Andere Zusammenstellungsoptionen 435
Datentypen 435
Der Präprozessor 436
Der Compiler 438
Die Übersetzungsphasen 440
Credits 441

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: c-language

It is an unofficial and free C Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official C Language.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Kapitel 1: Erste Schritte mit C Language

Bemerkungen

C ist eine universelle, zwingende Computerprogrammiersprache, die strukturierte Programmierung, den Umfang lexikalischer Variablen und Rekursion unterstützt, während ein statisches Typensystem viele unbeabsichtigten Operationen verhindert. Konstruktionsbedingt stellt C Konstrukte bereit, die typische Maschinenbefehle effizient abbilden, und hat daher in Anwendungen, die zuvor in Assemblersprache codiert wurden, einschließlich Betriebssystemen sowie verschiedener Anwendungssoftware für Computer, die von Supercomputern bis zu eingebetteten Systemen reichen, eine dauerhafte Verwendung gefunden .

Trotz ihrer geringen Fähigkeiten wurde die Sprache so konzipiert, dass sie die plattformübergreifende Programmierung fördert. Ein standardkonformes und portabel geschriebenes C-Programm kann für eine Vielzahl von Computerplattformen und Betriebssystemen mit wenigen Änderungen des Quellcodes erstellt werden. Die Sprache ist auf einer breiten Palette von Plattformen verfügbar, von eingebetteten Mikrocontrollern bis zu Supercomputern.

C wurde ursprünglich von Dennis Ritchie zwischen 1969 und 1973 in Bell Labs entwickelt und zur Neuinstallation der Unix- Betriebssysteme verwendet. Es ist seitdem zu einer der am weitesten verbreiteten Programmiersprachen aller Zeiten geworden, wobei C-Compiler verschiedener Hersteller für die Mehrzahl der vorhandenen Computerarchitekturen und Betriebssysteme verfügbar sind.

Gemeinsame Compiler

Der Prozess zum Kompilieren eines C-Programms unterscheidet sich zwischen Compilern und Betriebssystemen. Die meisten Betriebssysteme werden ohne Compiler ausgeliefert, daher müssen Sie einen installieren. Einige gängige Compiler-Optionen sind:

- GCC, die GNU Compiler Collection
- clang: ein Frontend für die C-Sprachfamilie für LLVM
- MSVC, Microsoft Visual C / C ++ - Erstellungstools

Die folgenden Dokumente sollten einen guten Überblick darüber geben, wie Sie einige der häufigsten Compiler verwenden können:

- Erste Schritte mit Microsoft Visual C
- Erste Schritte mit GCC

Compiler C-Version Unterstützung

Beachten Sie, dass Compiler unterschiedliche Standards für den Standard C haben, wobei C99 nach wie vor nicht vollständig unterstützt wird. Zum Beispiel unterstützt MSVC seit dem Release

2015 viel von C99, hat jedoch noch einige wichtige Ausnahmen für die Unterstützung der Sprache selbst (z. B. scheint die Vorverarbeitung nicht konform zu sein) und für die C-Bibliothek (z. B. `<tgmath.h>`) dokumentieren sie notwendigerweise ihre "implementierungsabhängigen Entscheidungen". Wikipedia hat eine Tabelle mit Unterstützung einiger populärer Compiler.

Einige Compiler (insbesondere GCC) haben *Compiler-Erweiterungen* angeboten oder bieten diese weiterhin an, die zusätzliche Funktionen implementieren, die die Compiler-Hersteller für notwendig erachten, hilfreich oder für wahrscheinlich in einer zukünftigen C-Version sind, die aber derzeit noch nicht in einem C-Standard enthalten sind. Da diese Erweiterungen compilerspezifisch sind, kann davon ausgegangen werden, dass sie nicht crosskompatibel sind, und Compilerentwickler können sie in späteren Compilerversionen entfernen oder ändern. Die Verwendung solcher Erweiterungen kann im Allgemeinen durch Compilerflags gesteuert werden.

Darüber hinaus verfügen viele Entwickler über Compiler, die nur bestimmte Versionen von C unterstützen, die von der Umgebung oder Plattform, auf die sie abzielen, festgelegt werden.

Wenn Sie einen Compiler auswählen, wird empfohlen, einen Compiler auszuwählen, der die beste Unterstützung für die neueste Version von C bietet, die für die Zielumgebung zulässig ist.

Code-Stil (Off-Topic hier):

Da Leerzeichen in C unbedeutend sind (das heißt, sie beeinflussen nicht die Funktionsweise des Codes), verwenden Programmierer häufig Leerzeichen, um den Code lesbarer und verständlicher zu machen. Dies wird als *Codestil bezeichnet* . Es ist ein Satz von Regeln und Richtlinien, die beim Schreiben des Quellcodes verwendet werden. Es werden Bedenken behandelt, wie Zeilen eingerückt werden sollen, ob Leerzeichen oder Tabulatoren verwendet werden sollen, wie Klammern gesetzt werden sollten, wie Leerzeichen um Operatoren und Klammern verwendet werden sollten, wie Variablen benannt werden sollten und so weiter.

Der Code-Stil wird nicht durch den Standard abgedeckt und basiert hauptsächlich auf der Meinung (verschiedene Personen finden unterschiedliche Stile für besser lesbar). Daher wird er im Allgemeinen für SO als Off-Topic betrachtet. Der übergeordnete Ratschlag zum Stil im eigenen Code lautet, dass Konsistenz oberste Priorität hat - wählen Sie einen Stil aus und machen Sie ihn zu einem Stil. Es genügt zu erklären, dass es verschiedene benannte Stile gibt, die häufig von Programmierern ausgewählt werden, anstatt einen eigenen Stil zu erstellen.

Einige gängige Einrückungsstile sind: K & R-Stil, Allman-Stil, GNU-Stil usw. Einige dieser Stile haben unterschiedliche Varianten. Allman wird beispielsweise als regulärer Allman oder als beliebte Variante Allman-8 verwendet. Informationen zu einigen der beliebtesten Stile finden Sie in der Wikipedia . Solche Stilnamen werden von den Standards verwendet, die die Autoren oder Organisationen häufig für die Verwendung durch viele Personen veröffentlichen, die zu ihrem Code beitragen, sodass jeder den Code leicht lesen kann, wenn er den Stil kennt, wie beispielsweise der GNU-Formatierungsleitfaden , aus dem ein Teil besteht das GNU-Kodierungsstandards- Dokument.

Einige gebräuchliche Namenskonventionen sind: UpperCamelCase, lowerCamelCase, lower_case_with_underscore, ALL_CAPS usw. Diese Stile werden auf verschiedene Weise

kombiniert, um sie mit verschiedenen Objekten und Typen zu verwenden (z. B. verwenden Makros häufig den ALL_CAPS-Stil).

Der K & R-Stil wird im Allgemeinen für die Verwendung in der SO-Dokumentation empfohlen, während die eher esoterischen Stile wie Pico nicht empfohlen werden.

Bibliotheken und APIs, die nicht durch den C-Standard abgedeckt sind (und daher hier außerhalb des Themas liegen):

- POSIX- API (z. B. für PThreads , Sockets , Signale)

Versionen

Ausführung	Standard	Veröffentlichungsdatum
K & R	n / a	1978-02-22
C89	ANSI X3.159-1989	1989-12-14
C90	ISO / IEC 9899: 1990	1990-12-20
C95	ISO / IEC 9899 / AMD1: 1995	1995-03-30
C99	ISO / IEC 9899: 1999	1999-12-16
C11	ISO / IEC 9899: 2011	2011-12-15

Examples

Hallo Welt

Um ein einfaches C-Programm zu erstellen, das *"Hello, World"* auf dem Bildschirm `hello.c` , erstellen Sie mit einem Texteditor eine neue Datei (z. B. `hello.c` - die Dateierweiterung muss `.c`), die folgenden Quellcode enthält:

Hallo c

```
#include <stdio.h>

int main(void)
{
    puts("Hello, World");
    return 0;
}
```

Live-Demo auf Coliru

Schauen wir uns dieses einfache Programm Zeile für Zeile an

```
#include <stdio.h>
```

Diese Zeile weist den Compiler an, den Inhalt der Standardbibliothekskopfdatei `stdio.h` in das Programm aufzunehmen. Header sind in der Regel Dateien, die Funktionsdeklarationen, Makros und Datentypen enthalten. Sie müssen die Headerdatei angeben, bevor Sie sie verwenden. Diese Zeile enthält `stdio.h` damit sie die Funktion `puts()` aufrufen kann.

Weitere Informationen zu Headern.

```
int main(void)
```

Diese Zeile startet die Definition einer Funktion. Sie enthält den Namen der Funktion (`main`), den Typ und die Anzahl der erwarteten Argumente (`void` , bedeutet none) und den Wertetyp, den diese Funktion zurückgibt (`int`). Die Programmausführung beginnt in der Funktion `main()` .

```
{
    …
}
```

Die geschweiften Klammern werden paarweise verwendet, um anzuzeigen, wo ein Codeblock beginnt und endet. Sie können auf viele Arten verwendet werden, aber in diesem Fall geben sie an, wo die Funktion beginnt und endet.

```
    puts("Hello, World");
```

Diese Zeile ruft die Funktion put `puts()` auf, um Text an die Standardausgabe (standardmäßig der Bildschirm) auszugeben, gefolgt von einem Zeilenumbruch. Die auszugebende Zeichenfolge ist in den Klammern enthalten.

`"Hello, World"` ist die Zeichenfolge, die auf den Bildschirm geschrieben wird. In C muss jeder String-Literalwert in den Anführungszeichen `"…"` .

Erfahren Sie mehr über Saiten.

In C-Programmen muss jede Anweisung mit einem Semikolon (dh `;`) abgeschlossen werden.

```
    return 0;
```

Bei der Definition von `main()` haben wir es als Funktion deklariert, die ein `int` , dh es muss eine ganze Zahl zurückgegeben werden. In diesem Beispiel wird der ganzzahlige Wert 0

zurückgegeben, der angibt, dass das Programm erfolgreich beendet wurde. Nach der `return 0;` Anweisung wird der Ausführungsprozess beendet.

Programm bearbeiten

Zu den einfachen Texteditoren zählen `vim` oder `gedit` unter Linux oder `Notepad` unter Windows. Plattformübergreifende Editoren enthalten auch `Visual Studio Code` oder `Sublime Text` .

Der Editor muss Klartextdateien erstellen, nicht RTF oder ein anderes Format.

Programm kompilieren und ausführen

Um das Programm auszuführen, muss diese Quelldatei (`hello.c`) zuerst in eine ausführbare Datei kompiliert werden (z. B. `hello` unter Unix / Linux oder `hello.exe` unter Windows). Dies geschieht mit einem Compiler für die C-Sprache.

Weitere Informationen zum Kompilieren

Mit GCC kompilieren

GCC (GNU Compiler Collection) ist ein weit verbreiteter C-Compiler. Öffnen Sie dazu ein Terminal, navigieren Sie mithilfe der Befehlszeile zum Speicherort der Quelldatei und führen Sie dann Folgendes aus:

```
gcc hello.c -o hello
```

Wenn im Quellcode (`hello.c`) keine Fehler gefunden werden, erstellt der Compiler eine **Binärdatei** , deren Name durch das Argument der Befehlszeilenoption `-o` (`hello`) angegeben wird. Dies ist die endgültige ausführbare Datei.

Wir können auch die `-Wall -Wextra -Werror` , mit deren Hilfe Probleme identifiziert werden, die dazu führen können, dass das Programm fehlschlägt oder unerwartete Ergebnisse `-Wall -Wextra -Werror` . Sie sind für dieses einfache Programm nicht notwendig, aber es ist eine Möglichkeit, sie hinzuzufügen:

```
gcc -Wall -Wextra -Werror -o hello hello.c
```

Verwendung des Clang-Compilers

Um das Programm mit `clang` zu kompilieren, können Sie `clang` verwenden:

```
clang -Wall -Wextra -Werror -o hello hello.c
```

Die Befehlszeilenoptionen von `clang` ähneln denen von GCC.

Verwenden des Microsoft C-Compilers über die Befehlszeile

Wenn Sie den Microsoft `cl.exe` Compiler auf einem Windows-System verwenden, das Visual Studio unterstützt, und wenn alle Umgebungsvariablen festgelegt sind, kann dieses C-Beispiel mit dem folgenden Befehl kompiliert werden, der eine ausführbare Datei `hello.exe` im Verzeichnis `hello.exe` in dem der Befehl ausgeführt wird (Es gibt `-Wall` wie `/W3` für `cl` , ungefähr analog zu `-Wall` usw. für GCC oder Clang).

```
cl hello.c
```

Programm ausführen

Nach dem Kompilieren kann die Binärdatei durch Eingabe von `./hello` in das Terminal ausgeführt werden. Nach der Ausführung gibt das kompilierte Programm `Hello, World` und eine neue Zeile an die Eingabeaufforderung aus.

Original "Hallo, Welt!" in K & R C

Folgendes ist das Original "Hallo, Welt!" Programm aus dem Buch The C Programming Language von Brian Kernighan und Dennis Ritchie (Ritchie war der ursprüngliche Entwickler der C-Programmiersprache bei Bell Labs), als "K & R" bezeichnet:

K & R

```
#include <stdio.h>

main()
{
    printf("hello, world\n");
}
```

Beachten Sie, dass die Programmiersprache C zum Zeitpunkt der Erstellung der ersten Ausgabe dieses Buches (1978) nicht standardisiert war und dass dieses Programm wahrscheinlich auf den meisten modernen Compilern nicht kompiliert werden kann, wenn sie nicht dazu aufgefordert werden, C90-Code zu akzeptieren.

Dieses erste Beispiel im K & R-Buch wird jetzt als schlechte Qualität angesehen, zum Teil weil es einen expliziten Rückgabetyp für `main()` und zum Teil weil es keine `return` . Die 2. Auflage des Buches wurde für den alten C89-Standard geschrieben. In C89 würde der Typ von `main` standardmäßig `int` , das K & R-Beispiel gibt jedoch keinen definierten Wert an die Umgebung zurück. In C99 und späteren Standards ist der Rückgabetyp erforderlich, aber es ist sicher, die `return` von `main` (und nur `main`) auszulassen, da ein Sonderfall mit C99 5.1.2.2.3 eingeführt wurde. Dies ist gleichbedeutend mit der Rückgabe von 0 was auf Erfolg hinweist.

Die empfohlene und meist tragbare Form von `main` für gehostete Systeme ist `int main (void)` wenn das Programm keine Befehlszeilenargumente verwendet, oder `int main(int argc, char **argv)` wenn das Programm die Befehlszeilenargumente verwendet.

C90 §5.1.2.2.3 **Programmbeendigung**

> Eine Rückkehr vom ersten Aufruf an die `main` entspricht dem Aufruf der `exit` Funktion mit dem von der `main` zurückgegebenen Wert als Argument. Wenn die `main` eine Rückgabe ausführt, die keinen Wert angibt, ist der an die Host-Umgebung zurückgegebene Beendigungsstatus nicht definiert.

C90 §6.6.6.4 **Die `return`**

> Wenn eine `return` Anweisung ohne Ausdruck ausgeführt wird und der Wert des Funktionsaufrufs vom Aufrufer verwendet wird, ist das Verhalten undefiniert. Das `}` , das eine Funktion beendet, entspricht der Ausführung einer `return` Anweisung ohne einen Ausdruck.

C99 §5.1.2.2.3 **Programmbeendigung**

> Wenn der Rückgabetyp der `main` ein mit `int` kompatibler Typ ist, entspricht eine Rückkehr vom ersten Aufruf an die `main` dem Aufruf der `exit` Funktion mit dem von der `main` zurückgegebenen Wert als Argument. Wenn Sie das `}` , das die `main` beendet, wird der Wert 0 zurückgegeben. Wenn der Rückgabetyp nicht mit `int` kompatibel ist, ist der an die Host-Umgebung zurückgegebene Beendigungsstatus nicht angegeben.

Erste Schritte mit C Language online lesen: https://riptutorial.com/de/c/topic/213/erste-schritte-mit-c-language

Kapitel 2: - Klassifizierung und Konvertierung von Zeichen

Examples

Klassifizieren von Zeichen, die aus einem Stream gelesen wurden

```
#include <ctype.h>
#include <stdio.h>

typedef struct {
  size_t space;
  size_t alnum;
  size_t punct;
} chartypes;

chartypes classify(FILE *f) {
  chartypes types = { 0, 0, 0 };
  int ch;

  while ((ch = fgetc(f)) != EOF) {
    types.space += !!isspace(ch);
    types.alnum += !!isalnum(ch);
    types.punct += !!ispunct(ch);
  }

  return types;
}
```

Die `classify` liest Zeichen aus einem Stream und zählt die Anzahl der Leerzeichen, alphanumerischen Zeichen und Satzzeichen. Es vermeidet mehrere Fallstricke.

- Beim Lesen eines Zeichens aus einem Stream wird das Ergebnis als `int` gespeichert, da andernfalls das Lesen von `EOF` (Dateiendemarker) und ein Zeichen, das dasselbe Bitmuster aufweist, nicht eindeutig sind.
- Die Zeichenklassifizierungsfunktionen (z. B. `isspace`) erwarten, dass ihr Argument *entweder als* `unsigned char` Zeichen *oder als Wert des* `EOF` *Makros darstellbar ist* . Da dies genau das ist, was die `fgetc` zurückgibt, ist hier keine Konvertierung erforderlich.
- Der Rückgabewert der Zeichenklassifizierungsfunktionen unterscheidet nur zwischen Null (Bedeutung `false`) und Nicht-Null (Bedeutung `true`). Um die Anzahl der Vorkommen zu zählen, muss dieser Wert in eine 1 oder 0 umgewandelt werden. Dies geschieht durch die doppelte Negation. `!!` .

Zeichen aus einer Zeichenfolge klassifizieren

```
#include <ctype.h>
#include <stddef.h>

typedef struct {
```

```
  size_t space;
  size_t alnum;
  size_t punct;
} chartypes;

chartypes classify(const char *s) {
  chartypes types = { 0, 0, 0 };
  const char *p;
  for (p= s; p != '\0'; p++) {
    types.space += !!isspace((unsigned char)*p);
    types.alnum += !!isalnum((unsigned char)*p);
    types.punct += !!ispunct((unsigned char)*p);
  }

  return types;
}
```

Die `classify` überprüft alle Zeichen einer Zeichenfolge und zählt die Anzahl der Leerzeichen, alphanumerischen Zeichen und Satzzeichen. Es vermeidet mehrere Fallstricke.

- Die Zeichenklassifizierungsfunktionen (z. B. `isspace`) erwarten, dass ihr Argument *entweder als `unsigned char` Zeichen oder als Wert des `EOF` Makros darstellbar ist* .
- Der Ausdruck `*p` ist vom Typ `char` und muss daher entsprechend dem obigen Wortlaut konvertiert werden.
- Der `char` Typ ist definiert als entweder `signed char` oder `unsigned char` .
- Wenn `char` einem `unsigned char` , besteht kein Problem, da jeder mögliche Wert des Typs " `char` als `unsigned char` .
- Wenn `char` äquivalent zu `signed char` , muss es in `unsigned char` umgewandelt werden, bevor es an die Zeichenklassifizierungsfunktionen übergeben wird. Und obwohl sich der Wert des Zeichens aufgrund dieser Konvertierung ändern kann, erwarten genau diese Funktionen diese Funktion.
- Der Rückgabewert der Zeichenklassifizierungsfunktionen unterscheidet nur zwischen Null (Bedeutung `false`) und Nicht-Null (Bedeutung `true`). Um die Anzahl der Vorkommen zu zählen, muss dieser Wert in eine 1 oder 0 umgewandelt werden. Dies geschieht durch die doppelte Negation. `!!` .

Einführung

Der Header `ctype.h` ist Teil der Standard-C-Bibliothek. Es bietet Funktionen zum Klassifizieren und Konvertieren von Zeichen.

Alle diese Funktionen benötigen einen Parameter, ein `int` , das entweder *EOF sein muss* oder als vorzeichenloses Zeichen dargestellt werden kann.

Die Namen der klassifizierenden Funktionen sind mit 'is' versehen. Jeder gibt einen ganzzahligen Wert ungleich Null (TRUE) zurück, wenn das übergebene Zeichen die zugehörige Bedingung erfüllt. Wenn die Bedingung nicht erfüllt ist, gibt die Funktion einen Nullwert zurück (FALSE).

Diese Klassifizierungsfunktionen funktionieren wie gezeigt, vorausgesetzt, das Standardgebietsschema C wird verwendet:

```
int a;
int c = 'A';
a = isalpha(c); /* Checks if c is alphabetic (A-Z, a-z), returns non-zero here. */
a = isalnum(c); /* Checks if c  is alphanumeric (A-Z, a-z, 0-9), returns non-zero here. */
a = iscntrl(c); /* Checks is c is a control character (0x00-0x1F, 0x7F), returns zero here. */
a = isdigit(c); /* Checks if c is a digit (0-9), returns zero here. */
a = isgraph(c); /* Checks if c has a graphical representation (any printing character except
space), returns non-zero here. */
a = islower(c); /* Checks if c is a lower-case letter (a-z), returns zero here. */
a = isprint(c); /* Checks if c is any printable character (including space), returns non-zero
here. */
a = isupper(c); /* Checks if c is a upper-case letter (a-z), returns zero here. */
a = ispunct(c); /* Checks if c is a punctuation character, returns zero here. */
a = isspace(c); /* Checks if c is a white-space character, returns zero here. */
a = isupper(c); /* Checks if c is an upper-case letter (A-Z), returns non-zero here. */
a = isxdigit(c); /* Checks if c is a hexadecimal digit (A-F, a-f, 0-9), returns non-zero here.
*/
```

C99

```
a = isblank(c); /* Checks if c is a blank character (space or tab), returns non-zero here. */
```

Es gibt zwei Konvertierungsfunktionen. Diese werden mit dem Präfix 'bis' benannt. Für diese Funktionen gelten dieselben Argumente wie oben. Der Rückgabewert ist jedoch keine einfache Null oder Nicht-Null, sondern das übergebene Argument hat sich in gewisser Weise geändert.

Diese Konvertierungsfunktionen funktionieren wie gezeigt, vorausgesetzt, das Standardgebietsschema C wird verwendet:

```
int a;
int c = 'A';

/* Converts c to a lower-case letter (a-z).
 * If conversion is not possible the unchanged value is returned.
 * Returns 'a' here.
 */
a = tolower(c);

/* Converts c to an upper-case letter (A-Z).
 * If conversion is not possible the unchanged value is returned.
 * Returns 'A' here.
 */
a = toupper(c);
```

Die folgende Information wird aus cplusplus.com-Zitate zitiert, wie der ursprüngliche aus 127 Zeichen bestehende ASCII-Satz von jeder der klassifizierenden Typfunktionen betrachtet wird (a • zeigt an, dass die Funktion für dieses Zeichen nicht Null zurückgibt)

ASCII-Werte	Zeichen	iscntrl	ist leer	isspace	Isupper	ist tiefer	isalpha	isdigit
0x00 .. 0x08	NUL, (andere Steuercodes)	•						

ASCII-Werte	Zeichen	iscntrl	ist leer	isspace	Isupper	ist tiefer	isalpha	isdigit
0x09	Tab ('\ t')	•	•	•				
0x0A .. 0x0D	(Leerraumkontrollcodes: '\ f', '\ v', '\ n', '\ r')	•		•				
0x0E .. 0x1F	(andere Steuercodes)	•						
0x20	Platz (' ')		•	•				
0x21 .. 0x2F	! "# $% & '() * +, -. /							
0x30 .. 0x39	0123456789							•
0x3a .. 0x40	:; <=>? @							
0x41 .. 0x46	ABCDEF				•		•	
0x47 .. 0x5A	GHIJKLMNOPQRSTUVWXYZ				•		•	
0x5B .. 0x60	[] ^ _ `							
0x61 .. 0x66	abcdef					•	•	
0x67 .. 0x7A	ghijklmnopqrstuvwxyz					•	•	
0x7B .. 0x7E	{} ~ bar							

ASCII-Werte	Zeichen	iscntrl	ist leer	isspace	Isupper	ist tiefer	isalpha	isdigit
0x7F	(DEL)	•						

- Klassifizierung und Konvertierung von Zeichen online lesen: https://riptutorial.com/de/c/topic/6846/-ctype-h----klassifizierung-und-konvertierung-von-zeichen

Kapitel 3: Aliasing und effektiver Typ

Bemerkungen

Verstöße gegen Aliasing-Regeln und gegen den effektiven Typ eines Objekts sind zwei verschiedene Dinge und sollten nicht verwechselt werden.

- *Aliasing* ist die Eigenschaft von zwei Zeigern `a` und `b` , die auf dasselbe Objekt verweisen, d. `a == b` .
- Der *effektive Typ* eines Datenobjekts wird von C verwendet, um zu bestimmen, welche Operationen an diesem Objekt ausgeführt werden können. Insbesondere wird der effektive Typ verwendet, um zu bestimmen, ob zwei Zeiger sich gegenseitig aliasieren können.

Aliasing kann ein Problem für die Optimierung, weil das Objekt durch einen Zeiger zu ändern, `a` sagen wir, kann das Objekt ändern , die durch den anderen Zeiger sichtbar ist `b` . Wenn Ihr C-Compiler davon ausgehen muss, dass Zeiger sich unabhängig von ihrem Typ und ihrer Herkunft immer gegenseitig Aliasnamen geben, gehen viele Optimierungsmöglichkeiten verloren und viele Programme laufen langsamer.

Die strengen Aliasing-Regeln von C beziehen sich auf Fälle, in denen der Compiler *annehmen kann,* welche Objekte sich gegenseitig aliasieren (oder nicht). Es gibt zwei Daumenregeln, die Sie für Datenzeiger immer beachten sollten.

> Wenn nicht anders angegeben, können zwei Zeiger mit demselben Basistyp einen Alias haben.
>
> Zwei Zeiger mit unterschiedlichen Basistypen können keinen Aliasnamen verwenden, es sei denn, einer der beiden Typen ist ein Zeichentyp.

Basistyp bedeutet hier, dass wir Typqualifikationen wie `const` beiseite stellen, z. B. Wenn `a double*` und `b const double*` , *muss* der Compiler im Allgemeinen annehmen, dass eine Änderung von `*a` `*b` ändern kann.

Die Verletzung der zweiten Regel kann katastrophale Folgen haben. Unter Verletzung der strengen Aliasing-Regel werden hier dem Compiler zwei Zeiger `a` und `b` unterschiedlichen Typs präsentiert, die in Wirklichkeit auf dasselbe Objekt zeigen. Der Compiler kann dann immer davon ausgehen, dass die beiden auf unterschiedliche Objekte verweisen, und wird seine Vorstellung von `*b` nicht aktualisieren, wenn Sie das Objekt durch `*a` geändert haben.

Wenn Sie dies tun, wird das Verhalten Ihres Programms undefiniert. Daher setzt C ziemlich strikte Einschränkungen für die Zeigerumwandlung, um zu vermeiden, dass eine solche Situation versehentlich auftritt.

> Wenn der Quell- oder Zieltyp nicht `void` , müssen alle Zeigerumwandlungen zwischen Zeigern mit unterschiedlichem Basistyp *explizit sein* .

Oder anders ausgedrückt, sie benötigen einen *Cast* , es sei denn, Sie führen eine Konvertierung durch, die dem Zieltyp lediglich ein Qualifikationsmerkmal wie `const` hinzufügt.

Das Vermeiden von Zeigerkonvertierungen im Allgemeinen und von Casts im Besonderen schützt Sie vor Aliasing-Problemen. Wenn Sie sie nicht wirklich brauchen und diese Fälle sehr speziell sind, sollten Sie sie nach Möglichkeit vermeiden.

Examples

Auf Zeichentypen kann nicht über Nicht-Zeichentypen zugegriffen werden.

Wenn ein Objekt mit einer statischen, Thread- oder automatischen Speicherdauer definiert ist und einen Zeichentyp hat, entweder: `char` , `unsigned char` oder `signed char` , kann nicht auf einen Nicht-Zeichentyp zugegriffen werden. Im folgenden Beispiel wird ein `char` Array als Typ `int` neu interpretiert, und das Verhalten ist bei jeder Dereferenzierung des `int` Zeigers `b` undefiniert.

```
int main( void )
{
    char a[100];
    int* b = ( int* )&a;
    *b = 1;

    static char c[100];
    b = ( int* )&c;
    *b = 2;

    _Thread_local char d[100];
    b = ( int* )&d;
    *b = 3;
}
```

Dies ist undefiniert, da es gegen die Regel des "effektiven Typs" verstößt. Auf ein Datenobjekt mit einem effektiven Typ darf nicht über einen anderen Typ, der kein Zeichentyp ist, zugegriffen werden. Da der andere Typ hier `int` , ist dies nicht zulässig.

Selbst wenn bekannt wäre, dass Ausrichtungs- und Zeigergrößen passen würden, wäre dies nicht von dieser Regel ausgenommen, das Verhalten wäre immer noch undefiniert.

Dies bedeutet insbesondere, dass es in Standard C nicht möglich ist, ein Pufferobjekt vom Zeichentyp zu reservieren, das durch Zeiger mit unterschiedlichen Typen verwendet werden kann, wie Sie einen Puffer verwenden würden, der von `malloc` oder einer ähnlichen Funktion empfangen wurde.

Ein korrekter Weg, um das gleiche Ziel wie im obigen Beispiel zu erreichen, wäre die Verwendung einer `union` .

```
typedef union bufType bufType;
union bufType {
   char c[sizeof(int[25])];
   int i[25];
};
```

```
int main( void )
{
    bufType a = { .c = { 0 } }; // reserve a buffer and initialize
    int* b = a.i;      // no cast necessary
    *b = 1;

    static bufType a = { .c = { 0 } };
    int* b = a.i;
    *b = 2;

    _Thread_local bufType a = { .c = { 0 } };
    int* b = a.i;
    *b = 3;
}
```

Hier sorgt die `union` dafür, dass der Compiler von Anfang an weiß, dass auf den Puffer über verschiedene Ansichten zugegriffen werden kann. Dies hat auch den Vorteil , dass nun der Puffer eine „Ansicht“ hat `ai` , dass bereits vom Typ `int` und keine Zeiger Konvertierung erforderlich ist .

Effektiver Typ

Der *effektive Typ* eines Datenobjekts ist, falls vorhanden, die letzte Typinformation, die ihm zugeordnet wurde.

```
// a normal variable, effective type uint32_t, and this type never changes
uint32_t a = 0.0;

// effective type of *pa is uint32_t, too, simply
// because *pa is the object a
uint32_t* pa = &a;

// the object pointed to by q has no effective type, yet
void* q = malloc(sizeof uint32_t);
// the object pointed to by q still has no effective type,
// because nobody has written to it
uint32_t* qb = q;
// *qb now has effective type uint32_t because a uint32_t value was written
*qb = 37;

// the object pointed to by r has no effective type, yet, although
// it is initialized
void* r = calloc(1, sizeof uint32_t);
// the object pointed to by r still has no effective type,
// because nobody has written to or read from it
uint32_t* rc = r;
// *rc now has effective type uint32_t because a value is read
// from it with that type. The read operation is valid because we used calloc.
// Now the object pointed to by r (which is the same as *rc) has
// gained an effective type, although we didn't change its value.
uint32_t c = *rc;

// the object pointed to by s has no effective type, yet.
void* s = malloc(sizeof uint32_t);
// the object pointed to by s now has effective type uint32_t
// because an uint32_t value is copied into it.
memcpy(s, r, sizeof uint32_t);
```

Beachten Sie, dass für letzteres nicht einmal ein `uint32_t*` -Zeiger auf dieses Objekt vorhanden war. Die Tatsache, dass wir ein anderes `uint32_t` Objekt kopiert haben, ist ausreichend.

Verletzung der strengen Aliasing-Regeln

Nehmen wir im folgenden Code an, dass `float` und `uint32_t` die gleiche Größe haben.

```
void fun(uint32_t* u, float* f) {
    float a = *f
    *u = 22;
    float b = *f;
    print("%g should equal %g\n", a, b);
}
```

`u` und `f` haben unterschiedliche Basistypen, und der Compiler kann daher davon ausgehen, dass sie auf verschiedene Objekte zeigen. Es gibt keine Möglichkeit, dass sich `*f` zwischen den beiden Initialisierungen von `a` und `b` geändert haben könnte. Der Compiler kann daher den Code auf etwas Äquivalent optimieren

```
void fun(uint32_t* u, float* f) {
    float a = *f
    *u = 22;
    print("%g should equal %g\n", a, a);
}
```

Das heißt, die zweite Ladeoperation von `*f` kann vollständig optimiert werden.

Wenn wir diese Funktion "normal" nennen

```
float fval = 4;
uint32_t uval = 77;
fun(&uval, &fval);
```

alles geht gut und sowas

> 4 sollte gleich 4 sein

wird gedruckt. Aber wenn wir betrügen und den gleichen Zeiger übergeben, nachdem wir ihn konvertiert haben,

```
float fval = 4;
uint32_t* up = (uint32_t*)&fval;
fun(up, &fval);
```

wir verstoßen gegen die strikte Aliasing-Regel. Dann wird das Verhalten undefiniert. Die Ausgabe könnte wie oben sein, wenn der Compiler den zweiten Zugriff optimiert hat, oder etwas völlig anderes, so dass Ihr Programm in einem völlig unzuverlässigen Zustand landet.

Qualifikation einschränken

Wenn wir zwei Zeigerargumente des gleichen Typs haben, kann der Compiler keine Annahme machen , und wird immer müssen davon ausgehen , dass die Änderung `*e` kann sich ändern `*f` :

```
void fun(float* e, float* f) {
    float a = *f
    *e = 22;
    float b = *f;
    print("is %g equal to %g?\n", a, b);
}

float fval = 4;
float eval = 77;
 fun(&eval, &fval);
```

alles geht gut und sowas

> ist 4 gleich 4?

wird gedruckt. Wenn wir denselben Zeiger übergeben, macht das Programm immer noch das Richtige und druckt

> ist 4 gleich 22?

Dies kann sich als ineffizient erweisen, wenn wir durch einige externe Informationen *wissen,* dass `e` und `f` niemals auf dasselbe Datenobjekt zeigen. Wir können dieses Wissen widerspiegeln, indem Sie den Zeigerparametern `restrict` hinzufügen:

```
void fan(float*restrict e, float*restrict f) {
    float a = *f
    *e = 22;
    float b = *f;
    print("is %g equal to %g?\n", a, b);
}
```

Dann kann der Compiler immer annehmen, dass `e` und `f` auf verschiedene Objekte zeigen.

Bytes ändern

Sobald ein Objekt einen effektiven Typ hat, sollten Sie nicht versuchen, es über einen Zeiger eines anderen Typs zu ändern, es sei denn, dieser andere Typ ist ein Zeichentyp, `char` , `signed char` oder `unsigned char` .

```
#include <inttypes.h>
#include <stdio.h>

int main(void) {
  uint32_t a = 57;
  // conversion from incompatible types needs a cast !
  unsigned char* ap = (unsigned char*)&a;
  for (size_t i = 0; i < sizeof a; ++i) {
    /* set each byte of a to 42 */
    ap[i] = 42;
  }
```

```
  printf("a now has value %" PRIu32 "\n", a);
}
```

Dies ist ein gültiges Programm, das druckt

> a hat jetzt den Wert 707406378

Das funktioniert aus folgenden Gründen:

- Der Zugriff erfolgt auf die einzelnen Bytes, die vom Typ `unsigned char` sodass jede Änderung gut definiert ist.
- Die beiden Ansichten des Objekts durch `a` und durch `*ap` , ein Alias, aber da `ap` ein Zeiger auf einen Zeichentyp ist, gilt die strikte Aliasing-Regel nicht. Der Compiler muss also davon ausgehen, dass der Wert von `a` in der `for` Schleife geändert wurde. Der geänderte Wert von `a` muss aus den Bytes erstellt werden, die geändert wurden.
- Der Typ von `a` , `uint32_t` enthält keine Füllbits. Alle seine Bits der Repräsentation zählen für den Wert, hier `707406378` , und es kann keine Trap-Darstellung geben.

Aliasing und effektiver Typ online lesen: https://riptutorial.com/de/c/topic/1301/aliasing-und-effektiver-typ

Kapitel 4: Allgemeine C-Programmiersprachen und Entwicklerpraktiken

Examples

Vergleich von Literal und Variable

Angenommen, Sie vergleichen den Wert mit einer Variablen

```
if ( i  == 2) //Bad-way
{
    doSomething;
}
```

Angenommen, Sie haben `==` mit `=` verwechselt. Dann braucht es deine süße Zeit, um es herauszufinden.

```
if( 2 == i) //Good-way
{
    doSomething;
}
```

Wenn versehentlich ein Gleichheitszeichen ausgelassen wird, beschwert sich der Compiler über eine „versuchte Zuweisung von Literalen“. Dies schützt Sie nicht beim Vergleich zweier Variablen, aber jedes kleine bisschen hilft.

Sehen Sie hier für weitere Informationen.

Lassen Sie die Parameterliste einer Funktion nicht leer - verwenden Sie void

Angenommen, Sie erstellen eine Funktion, die beim Aufruf keine Argumente erfordert, und Sie stehen vor dem Dilemma, wie Sie die Parameterliste im Funktionsprototyp und in der Funktionsdefinition definieren.

- Sie haben die Wahl, die Parameterliste sowohl für den Prototyp als auch für die Definition leer zu lassen. Dabei sehen sie genauso aus wie die Funktionsaufruf-Anweisung, die Sie benötigen.
- Sie haben irgendwo gelesen, dass eine der Verwendungen des Schlüsselworts ***void*** (es gibt nur einige davon) gibt, ist die Definition der Parameterliste der Funktionen, die in ihrem Aufruf keine Argumente akzeptieren. Das ist also auch eine Wahl.

Welches ist die richtige Wahl?

ANTWORT: Verwenden Sie das Schlüsselwort ***void***

ALLGEMEINE HINWEISE: Wenn eine Sprache bestimmte Funktionen enthält, die für einen bestimmten Zweck verwendet werden sollen, ist es besser, diese in Ihrem Code zu verwenden. Verwenden Sie beispielsweise `enum` s anstelle von `#define` Makros (das ist ein anderes Beispiel).

C11 Abschnitt 6.7.6.3 "Funktionsdeklaratoren", Absatz 10, bestimmt:

> Der Sonderfall eines unbenannten Parameters vom Typ void als einziges Element in der Liste gibt an, dass die Funktion keine Parameter hat.

Paragraph 14 desselben Abschnitts zeigt den einzigen Unterschied:

> … Eine leere Liste in einem Funktionsdeklarator, die Teil einer Definition dieser Funktion ist, gibt an, dass die Funktion keine Parameter hat. Die leere Liste in einem Funktionsdeklarator, der nicht Teil einer Definition dieser Funktion ist, gibt an, dass keine Informationen zu Anzahl oder Typ der Parameter angegeben werden.

Eine vereinfachte Erklärung von K & R (S. 72-73) für das oben genannte Zeug:

> Wenn eine Funktionsdeklaration keine Argumente enthält, wie in `double atof();` `atof` bedeutet auch, dass von den Argumenten von `atof` nichts angenommen werden `atof` . Alle Parameterüberprüfung ist deaktiviert. Diese besondere Bedeutung der leeren Argumentliste soll älteren C-Programmen ermöglichen, mit neuen Compilern zu kompilieren. Aber es ist keine gute Idee, es mit neuen Programmen zu verwenden. Wenn die Funktion Argumente akzeptiert, deklarieren Sie sie. Wenn es keine Argumente erfordert, verwenden Sie `void` .

So sollte Ihr Funktionsprototyp aussehen:

```
int foo(void);
```

Und so sollte die Funktionsdefinition sein:

```
int foo(void)
{
    ...
    <statements>
    ...
    return 1;
}
```

Die Verwendung des obigen Deklarationstyps " `int foo()` (dh ohne Verwendung des Schlüsselworts ***void***) hat den Vorteil, dass der Compiler den Fehler erkennen kann, wenn Sie Ihre Funktion mit einer fehlerhaften Anweisung wie `foo(42)` aufrufen. Diese Art einer Funktionsaufrufanweisung würde keine Fehler verursachen, wenn Sie die Parameterliste leer lassen. Der Fehler würde unbemerkt unbemerkt weitergegeben und der Code würde immer noch ausgeführt.

Das bedeutet auch, dass Sie die `main()` Funktion folgendermaßen definieren sollten:

```
int main(void)
{
    ...
    <statements>
    ...
    return 0;
}
```

Beachten Sie, dass eine mit einer leeren Parameterliste definierte Funktion zwar keine Argumente enthält, jedoch keinen Prototyp für die Funktion bereitstellt. Der Compiler beschwert sich also nicht, wenn die Funktion anschließend mit Argumenten aufgerufen wird. Zum Beispiel:

```
#include <stdio.h>

static void parameterless()
{
    printf("%s called\n", __func__);
}

int main(void)
{
    parameterless(3, "arguments", "provided");
    return 0;
}
```

Wenn dieser Code in der Datei `proto79.c` gespeichert `proto79.c` , kann er unter GCC (Version 7.1.0 unter macOS Sierra 10.12.5 für Demonstrationszwecke) unter Unix folgendermaßen kompiliert werden:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -pedantic proto79.c -o
proto79
$
```

Wenn Sie mit strengeren Optionen kompilieren, erhalten Sie Fehler:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes -Wold-
style-definition -pedantic proto79.c -o proto79
proto79.c:3:13: error: function declaration isn’t a prototype [-Werror=strict-prototypes]
 static void parameterless()
             ^~~~~~~~~~~~~
proto79.c: In function ‘parameterless’:
proto79.c:3:13: error: old-style function definition [-Werror=old-style-definition]
cc1: all warnings being treated as errors
$
```

Wenn Sie der Funktion den formalen Prototyp `static void parameterless(void)` , führt die Kompilierung zu Fehlern:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes -Wold-
style-definition -pedantic proto79.c -o proto79
proto79.c: In function ‘main’:
proto79.c:10:5: error: too many arguments to function ‘parameterless’
     parameterless(3, "arguments", "provided");
     ^~~~~~~~~~~~~
```

```
proto79.c:3:13: note: declared here
 static void parameterless(void)
             ^~~~~~~~~~~~~
$
```

Moral - Stellen Sie immer sicher, dass Sie Prototypen haben, und stellen Sie sicher, dass Ihr Compiler Ihnen sagt, wenn Sie sich nicht an die Regeln halten.

Allgemeine C-Programmiersprachen und Entwicklerpraktiken online lesen: https://riptutorial.com/de/c/topic/10543/allgemeine-c-programmiersprachen-und-entwicklerpraktiken

Kapitel 5: Arrays

Einführung

Arrays sind abgeleitete Datentypen, die eine geordnete Sammlung von Werten ("Elementen") eines anderen Typs darstellen. Die meisten Arrays in C haben eine feste Anzahl von Elementen eines beliebigen Typs und ihre Darstellung speichert die Elemente zusammenhängend im Speicher ohne Lücken oder Auffüllen. C erlaubt mehrdimensionale Arrays, deren Elemente andere Arrays sind, sowie Arrays von Zeigern.

C unterstützt dynamisch zugewiesene Arrays, deren Größe zur Laufzeit bestimmt wird. C99 und höher unterstützt Arrays mit variabler Länge oder VLAs.

Syntax

- Typname [Länge]; / * Definiere ein Array von 'type' mit Namen 'name' und length 'length'. * /
- int arr [10] = {0}; / * Definiere ein Array und initialisiere ALLE Elemente auf 0. * /
- int arr [10] = {42}; / * Definiere ein Array und initialisiere die ersten Elemente auf 42 und den Rest auf 0. * /
- int arr [] = {4, 2, 3, 1}; / * Definiere und initialisiere ein Array der Länge 4. * /
- arr [n] = Wert; / * Wert an Index n setzen. * /
- Wert = arr [n]; / * Wert an Index n abrufen. * /

Bemerkungen

Warum brauchen wir Arrays?

Arrays bieten eine Möglichkeit, Objekte zu einem Aggregat mit eigener Bedeutung zu organisieren. Zum Beispiel C - Strings sind Arrays von Zeichen (`char` s) und eine Zeichenkette wie „Hallo, Welt!“ hat eine Bedeutung als Aggregat, das den Zeichen nicht individuell inhärent ist. In ähnlicher Weise werden Arrays üblicherweise verwendet, um mathematische Vektoren und Matrizen sowie Listen vieler Arten darzustellen. Ohne die Elemente gruppieren zu können, müsste man jedes einzeln ansprechen, z. B. über separate Variablen. Das ist nicht nur unhandlich, es kann nicht einfach Sammlungen unterschiedlicher Längen aufnehmen.

Arrays werden in den meisten Kontexten implizit in Zeiger konvertiert .

Außer , wenn als Operand der erscheinen `sizeof` - Operators, der `_Alignof` Operator (C2011) oder die unären `&` (Adresse-of) Operator oder als Stringliteral verwendet , um einen (anderen) Array zu initialisieren, wird ein Array implizit umgewandelt in ("decays to") einen Zeiger auf sein erstes Element. Diese implizite Konvertierung ist eng an die Definition des Array-Subskriptionsoperators (`[]`) gekoppelt: Der Ausdruck `arr[idx]` ist als äquivalent zu `*(arr + idx)` . Da die Zeigerarithmetik kommutativ ist, ist `*(arr + idx)` auch äquivalent zu `*(idx + arr)` , was wiederum `idx[arr]` . Alle diese Ausdrücke sind gültig und werden mit demselben Wert ausgewertet, vorausgesetzt, entweder `idx` oder `arr` ist ein Zeiger (oder ein Array, das in einen Zeiger zerfällt), der andere ist

eine Ganzzahl und die Ganzzahl ist ein gültiger Index für das Array auf den der Zeiger zeigt.

Beachten Sie als Sonderfall, dass `&(arr[0])` äquivalent zu `&*(arr + 0)` , was die Vereinfachung von `arr` vereinfacht. Alle diese Ausdrücke sind überall dort austauschbar, wo der letzte zu einem Zeiger zerfällt. Dies drückt einfach wieder aus, dass ein Array zu einem Zeiger auf sein erstes Element zerfällt.

Im Gegensatz dazu hat das Ergebnis den Typ `T (*)[N]` und zeigt auf das gesamte Array, wenn der Address-of-Operator auf ein Array vom Typ `T[N]` angewendet wird (*dh* `&arr`). Dies unterscheidet sich von einem Zeiger auf das erste Array-Element zumindest in Bezug auf die Zeigerarithmetik, die in Bezug auf die Größe des Zeigetyps definiert ist.

Funktionsparameter sind keine Arrays .

```
void foo(int a[], int n);
void foo(int *a, int n);
```

Obwohl die erste Deklaration von `foo` für Parameter a `a` arrayartige Syntax verwendet, wird mit dieser Syntax ein Funktionsparameter deklariert, der diesen Parameter als *Zeiger* auf den Elementtyp des Arrays deklariert. Somit ist die zweite Signatur für `foo()` semantisch identisch mit der ersten. Dies entspricht den Zerfall von Array - Wert auf Zeiger , wo sie als Argument für einen *Funktionsaufruf* angezeigt wird , so dass , wenn eine Variable , und ein Funktionsparameter mit dem gleichen Array - Typ deklariert werden dann Wert, den Variable zur Verwendung in einem Funktionsaufruf als das geeignet ist , Argument, das dem Parameter zugeordnet ist.

Examples

Array deklarieren und initialisieren

Die allgemeine Syntax zum Deklarieren eines eindimensionalen Arrays lautet

```
type arrName[size];
```

wobei `type` ein beliebiger integrierter Typ oder benutzerdefinierte Typen wie Strukturen sein kann, ist `arrName` eine benutzerdefinierte Kennung und `size` eine Ganzzahlkonstante.

Das Deklarieren eines Arrays (in diesem Fall ein Array von 10 int-Variablen) erfolgt folgendermaßen:

```
int array[10];
```

es enthält jetzt unbestimmte Werte. Um sicherzustellen, dass es während der Deklaration null Werte enthält, können Sie Folgendes tun:

```
int array[10] = {0};
```

Arrays können auch Initialisierer enthalten. In diesem Beispiel wird ein Array von 10 `int` 's

deklariert, wobei die ersten 3 `int` 's die Werte `1` , `2` , `3` enthalten.

```
int array[10] = {1, 2, 3};
```

Bei der obigen Initialisierungsmethode wird der erste Wert in der Liste dem ersten Mitglied des Arrays zugewiesen, der zweite Wert wird dem zweiten Mitglied des Arrays usw. zugewiesen. Wenn die Listengröße kleiner als die Arraygröße ist, werden wie im obigen Beispiel die verbleibenden Mitglieder des Arrays mit Nullen initialisiert. Mit der Initialisierung der benannten Liste (ISO C99) ist eine explizite Initialisierung der Array-Mitglieder möglich. Zum Beispiel,

```
int array[5] = {[2] = 5, [1] = 2, [4] = 9}; /* array is {0, 2, 5, 0, 9} */
```

In den meisten Fällen kann der Compiler die Länge des Arrays für Sie ableiten. Dies kann erreicht werden, indem Sie die eckigen Klammern leer lassen:

```
int array[] = {1, 2, 3}; /* an array of 3 int's */
int array[] = {[3] = 8, [0] = 9}; /* size is 4 */
```

Das Deklarieren eines Arrays mit der Länge Null ist nicht zulässig.

C99 C11

Arrays mit variabler Länge (kurz VLA) wurden in C99 hinzugefügt und in C11 optional gemacht. Sie sind normalen Arrays gleich, mit einem wichtigen Unterschied: Die Länge muss zum Zeitpunkt des Kompilierens nicht bekannt sein. VLAs haben eine automatische Speicherdauer. Nur Zeiger auf VLAs können eine statische Speicherdauer haben.

```
size_t m = calc_length(); /* calculate array length at runtime */
int vla[m];               /* create array with calculated length */
```

Wichtig:

VLAs sind potenziell gefährlich. Wenn das Array `vla` im obigen Beispiel mehr Speicherplatz auf dem Stack benötigt als verfügbar, wird der Stack überlaufen. Die Verwendung von VLAs wird daher häufig in Style-Guides, Büchern und Übungen empfohlen.

Löschen des Array-Inhalts (Nullstellen)

Manchmal ist es erforderlich, ein Array auf Null zu setzen, nachdem die Initialisierung durchgeführt wurde.

```
#include <stdlib.h> /* for EXIT_SUCCESS */

#define ARRLEN (10)

int main(void)
{
  int array[ARRLEN]; /* Allocated but not initialised, as not defined static or global. */
```

```
  size_t i;
  for(i = 0; i < ARRLEN; ++i)
  {
    array[i] = 0;
  }

  return EXIT_SUCCESS;
}
```

Eine übliche Abkürzung für die obige Schleife ist die Verwendung von `memset()` aus `<string.h>` . Wenn Sie das `array` wie unten gezeigt übergeben, verfällt es in einen Zeiger auf sein 1. Element.

```
memset(array, 0, ARRLEN * sizeof (int)); /* Use size explicitly provided type (int here). */
```

oder

```
memset(array, 0, ARRLEN * sizeof *array); /* Use size of type the pointer is pointing to. */
```

Wie in diesem Beispiel `array` *ist* ein Array und nicht nur einen Zeiger auf ein erstes Element des Arrays (siehe Array Länge auf , warum dies wichtig ist) eine dritte Option auf 0-out Array ist möglich:

```
 memset(array, 0, sizeof array); /* Use size of the array itself. */
```

Arraylänge

Arrays haben feste Längen, die im Rahmen ihrer Deklarationen bekannt sind. Trotzdem ist es möglich und manchmal bequem, die Feldlänge zu berechnen. Dies kann insbesondere den Code flexibler machen, wenn die Arraylänge automatisch von einem Initialisierer bestimmt wird:

```
int array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

/* size of `array` in bytes */
size_t size = sizeof(array);

/* number of elements in `array` */
size_t length = sizeof(array) / sizeof(array[0]);
```

In den meisten Kontexten, in denen ein Array in einem Ausdruck enthalten ist, wird es jedoch automatisch in einen Zeiger auf das erste Element konvertiert ("zerlegt"). Der Fall, in dem ein Array der Operand des Operators `sizeof` , ist einer von `sizeof` Ausnahmen. Der resultierende Zeiger ist selbst kein Array und enthält keine Informationen über die Länge des Arrays, von dem er abgeleitet wurde. Wenn diese Länge in Verbindung mit dem Zeiger benötigt wird, beispielsweise wenn der Zeiger an eine Funktion übergeben wird, muss er separat übermittelt werden.

Angenommen, wir möchten eine Funktion schreiben, um das letzte Element eines Arrays von `int` . Wenn wir von oben aus fortfahren, könnten wir es so nennen:

```
/* array will decay to a pointer, so the length must be passed separately */
int last = get_last(array, length);
```

Die Funktion könnte folgendermaßen implementiert werden:

```
int get_last(int input[], size_t length) {
    return input[length - 1];
}
```

Insbesondere ist zu beachten , dass , obwohl die Erklärung der `input` ähnelt dem eines Arrays, **es in** der **Tat erklärt `input` als Zeiger** (bis `int`). Es ist genau gleichbedeutend mit der Deklaration der `input` als `int *input` . Das Gleiche würde auch gelten, wenn eine Dimension gegeben würde. Dies ist möglich, weil Arrays nicht immer tatsächliche Argumente für Funktionen sein können (sie zerfallen zu Zeigern, wenn sie in Funktionsaufrufausdrücken erscheinen) und sie können als Mnemonik betrachtet werden.

Es ist ein sehr häufiger Fehler, zu versuchen, die Array-Größe anhand eines Zeigers zu ermitteln, der nicht funktionieren kann. MACH DAS NICHT:

```
int BAD_get_last(int input[]) {
    /* INCORRECTLY COMPUTES THE LENGTH OF THE ARRAY INTO WHICH input POINTS: */
    size_t length = sizeof(input) / sizeof(input[0]));

    return input[length - 1];  /* Oops -- not the droid we are looking for */
}
```

Tatsächlich ist dieser bestimmte Fehler so häufig, dass einige Compiler ihn erkennen und davor warnen. `clang` wird beispielsweise folgende Warnung ausgegeben:

```
warning: sizeof on array function parameter will return size of 'int *' instead of 'int []' [-
Wsizeof-array-argument]
        int length = sizeof(input) / sizeof(input[0]);
                           ^
note: declared here
int BAD_get_last(int input[])
                     ^
```

Einstellungswerte in Arrays

Der Zugriff auf Array-Werte erfolgt im Allgemeinen über eckige Klammern:

```
int val;
int array[10];

/* Setting the value of the fifth element to 5: */
array[4] = 5;

/* The above is equal to: */
*(array + 4) = 5;

/* Reading the value of the fifth element: */
val = array[4];
```

Als Nebeneffekt der Operanden auf den austauschbaren Operator `+` (-> Kommutativgesetz) ist Folgendes gleichwertig:

```
*(array + 4) = 5;
*(4 + array) = 5;
```

so gut wie die nächsten Aussagen sind gleichwertig:

```
array[4] = 5;
4[array] = 5; /* Weird but valid C ... */
```

und diese beiden auch:

```
val = array[4];
val = 4[array]; /* Weird but valid C ... */
```

C führt keine Grenzprüfungen, die Inhalte außerhalb der deklarierte Array Zugriff ist nicht definiert (Zugriff über chunk zugewiesen Speicher):

```
int val;
int array[10];

array[4] = 5;    /* ok */
val = array[4];  /* ok */
array[19] = 20;  /* undefined behavior */
val = array[15]; /* undefined behavior */
```

Definieren Sie Array- und Access-Array-Elemente

```
#include <stdio.h>

#define ARRLEN (10)

int main (void)
{

   int n[ ARRLEN ]; /* n is an array of 10 integers */
   size_t i, j; /* Use size_t to address memory, that is to index arrays, as its guaranteed to

                   be wide enough to address all of the possible available memory.
                   Using signed integers to do so should be considered a special use case,
                   and should be restricted to the uncommon case of being in the need of
                   negative indexes. */

   /* Initialize elements of array n. */
   for ( i = 0; i < ARRLEN ; i++ )
   {
      n[ i ] = i + 100; /* Set element at location i to i + 100. */
   }

   /* Output each array element's value. */
   for (j = 0; j < ARRLEN ; j++ )
   {
      printf("Element[%zu] = %d\n", j, n[j] );
```

```
    }

    return 0;
}
```

Ordnen Sie ein Array mit benutzerdefinierter Größe zu und initialisieren Sie es mit Null

```
#include <stdio.h>
#include <stdlib.h>


int main (void)
{
  int * pdata;
  size_t n;

  printf ("Enter the size of the array: ");
  fflush(stdout); /* Make sure the prompt gets printed to buffered stdout. */

  if (1 != scanf("%zu", &n)) /* If zu is not supported (Windows?) use lu. */
  {
    fprintf("scanf() did not read a in proper value.\n");
    exit(EXIT_FAILURE);
  }

  pdata = calloc(n, sizeof *pdata);
  if (NULL == pdata)
  {
    perror("calloc() failed"); /* Print error. */
    exit(EXIT_FAILURE);
  }

  free(pdata); /* Clean up. */

  return EXIT_SUCCESS;
}
```

Dieses Programm versucht, einen vorzeichenlosen Integer-Wert von der Standardeingabe zu scannen, ordnet einen Speicherblock für ein Array von `n` Elementen des Typs `int` indem die Funktion `calloc()` . Der Speicher wird von letzteren auf alle Nullen initialisiert.

Im Erfolgsfall wird der Speicher durch den Aufruf von `free()` .

Iteration durch ein Array effizient und Reihenreihenfolge

Arrays in C können als zusammenhängender Speicherbereich betrachtet werden. Genauer gesagt ist die letzte Dimension des Arrays der zusammenhängende Teil. Wir nennen dies die *Reihenreihenfolge* . Wenn Sie dies und die Tatsache verstehen, dass ein Cache-Fehler beim Zugriff auf nicht zwischengespeicherte Daten eine vollständige Cache-Zeile in den Cache lädt, um nachfolgende Cache-Fehler zu vermeiden, können Sie sehen, warum beim Zugriff auf ein Array der Dimension 10000x10000 mit `array[0][0]` **möglicherweise** `array[0][1]` im Cache, aber ein direkter Zugriff auf `array[1][0]` würde zu einem zweiten Cache-Fehler führen, da er `sizeof(type)*10000` Byte von `array[0][0]` ist und daher sicherlich nicht in derselben Cachezeile.

Deshalb ist das Iterieren ineffizient:

```
#define ARRLEN 10000
int array[ARRLEN][ARRLEN];

size_t i, j;
for (i = 0; i < ARRLEN; ++i)
{
    for(j = 0; j < ARRLEN; ++j)
    {
        array[j][i] = 0;
    }
}
```

Und so zu iterieren ist effizienter:

```
#define ARRLEN 10000
int array[ARRLEN][ARRLEN];

size_t i, j;
for (i = 0; i < ARRLEN; ++i)
{
    for(j = 0; j < ARRLEN; ++j)
    {
        array[i][j] = 0;
    }
}
```

Dies ist der gleiche Grund, warum es beim Umgang mit einem Array mit einer Dimension und mehreren Indizes (der Einfachheit halber mit den Indizes i und j zwei Dimensionen genannt wird) wichtig, das Array folgendermaßen zu durchlaufen:

```
#define DIM_X 10
#define DIM_Y 20

int array[DIM_X*DIM_Y];

size_t i, j;
for (i = 0; i < DIM_X; ++i)
{
    for(j = 0; j < DIM_Y; ++j)
    {
        array[i*DIM_Y+j] = 0;
    }
}
```

Oder mit 3 Dimensionen und Indizes i, j und k:

```
#define DIM_X 10
#define DIM_Y 20
#define DIM_Z 30

int array[DIM_X*DIM_Y*DIM_Z];

size_t i, j, k;
for (i = 0; i < DIM_X; ++i)
```

```
{
    for(j = 0; j < DIM_Y; ++j)
    {
        for (k = 0; k < DIM_Z; ++k)
        {
            array[i*DIM_Y*DIM_Z+j*DIM_Z+k] = 0;
        }
    }
}
```

Oder generischer, wenn wir ein Array mit **N1 x N2 x ... x Nd** Elementen, **d** Dimensionen und Indizes haben, die als **n1, n2, ... bezeichnet werden, und** der Versatz so berechnet wird

$$n_d + N_d \cdot (n_{d-1} + N_{d-1} \cdot (n_{d-2} + N_{d-2} \cdot (\cdots + N_2 n_1)\cdots))) = \sum_{k=1}^{d} \left(\prod_{\ell=k+1}^{d} N_\ell \right) n_k$$

Bild / Formel entnommen aus: https://en.wikipedia.org/wiki/Row-major_order

Mehrdimensionale Arrays

Die Programmiersprache C erlaubt mehrdimensionale Arrays . Hier ist die allgemeine Form einer multidimensionalen Arraydeklaration:

```
type name[size1][size2]...[sizeN];
```

Die folgende Deklaration erstellt beispielsweise ein dreidimensionales (5 x 10 x 4) Ganzzahl-Array:

```
int arr[5][10][4];
```

Zweidimensionale Arrays

Die einfachste Form eines mehrdimensionalen Arrays ist das zweidimensionale Array. Ein zweidimensionales Array ist im Wesentlichen eine Liste von eindimensionalen Arrays. Um ein zweidimensionales ganzzahliges Array mit Dimensionen mxn zu deklarieren, können wir wie folgt schreiben:

```
type arrayName[m][n];
```

Dabei kann `type` ein beliebiger gültiger C-Datentyp (`int` , `float` usw.) und `arrayName` ein beliebiger gültiger C-Bezeichner sein. Ein zweidimensionales Array kann als Tabelle mit `m` Zeilen und `n` Spalten dargestellt werden. **Hinweis** : Die Reihenfolge *ist* in C von Bedeutung. Das Array `int a[4][3]` *stimmt* nicht mit dem Array `int a[3][4]` . Die Anzahl der Zeilen kommt zuerst als eine *Zeile* C -Major Sprache ist.

Ein zweidimensionales Array `a` , das drei Zeilen und vier Spalten enthält, kann wie folgt dargestellt werden:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Somit wird jedes Element in dem Array `a` durch einen Elementnamen der Form `a[i][j]` identifiziert, wobei `a` der Name des Arrays ist, `i` die Zeile und `j` die Spalte darstellt. Erinnern Sie sich daran, dass Zeilen und Spalten null sind. Dies ist der mathematischen Schreibweise für die Subskription von 2-D-Matrizen sehr ähnlich.

Initialisieren von zweidimensionalen Arrays

Mehrdimensionale Arrays können durch Angabe von Klammerwerten für jede Zeile initialisiert werden. Im Folgenden wird ein Array mit 3 Zeilen definiert, wobei jede Zeile 4 Spalten hat.

```
int a[3][4] = {
   {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
   {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
   {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

Die geschachtelten geschweiften Klammern, die die beabsichtigte Zeile angeben, sind optional. Die folgende Initialisierung entspricht dem vorherigen Beispiel:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Die Methode zum Erstellen von Arrays mit geschachtelten geschweiften Klammern ist optional, wird jedoch dringend empfohlen, da sie lesbarer und übersichtlicher ist.

Zugriff auf zweidimensionale Arrayelemente

Auf ein Element in einem zweidimensionalen Array wird mit den Indizes zugegriffen, z. B. Zeilenindex und Spaltenindex des Arrays. Zum Beispiel -

```
int val = a[2][3];
```

Die obige Anweisung nimmt das 4. Element aus der 3. Zeile des Arrays. Lassen Sie uns das folgende Programm überprüfen, in dem wir eine geschachtelte Schleife verwendet haben, um ein zweidimensionales Array zu behandeln:

```
#include <stdio.h>

int main () {

   /* an array with 5 rows and 2 columns*/
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
   int i, j;

   /* output each array element's value */
```

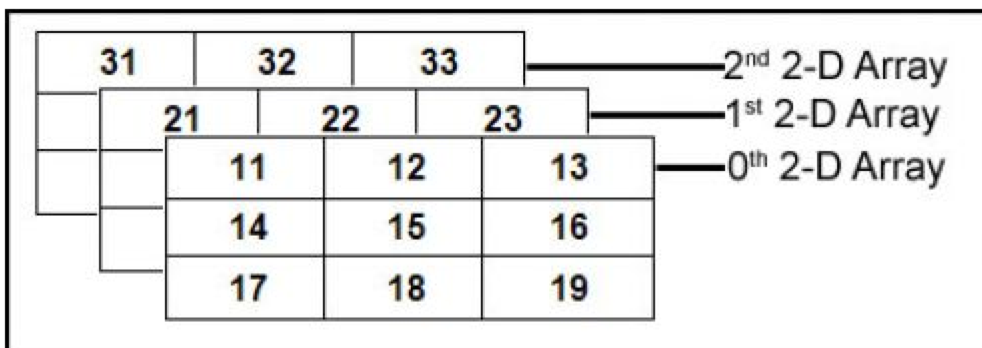
```
   for ( i = 0; i < 5; i++ ) {

      for ( j = 0; j < 2; j++ ) {
         printf("a[%d][%d] = %d\n", i,j, a[i][j] );
      }
   }

   return 0;
}
```

Wenn der obige Code kompiliert und ausgeführt wird, führt er zu folgendem Ergebnis:

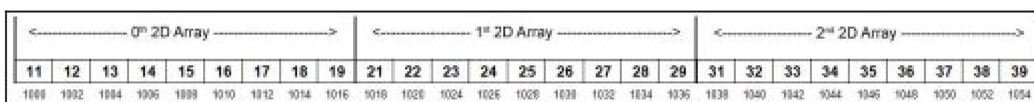
```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

Dreidimensionales Array:

Ein 3D-Array ist im Wesentlichen ein Array von Arrays: Es handelt sich um ein Array oder eine Sammlung von 2D-Arrays, und ein 2D-Array ist ein Array von 1D-Arrays.



3D-Array-Speicherzuordnung:



Initialisieren eines 3D-Arrays:

```
double cprogram[3][2][4]={
{{-0.1, 0.22, 0.3, 4.3}, {2.3, 4.7, -0.9, 2}},
 {{0.9, 3.6, 4.5, 4}, {1.2, 2.4, 0.22, -1}},
 {{8.2, 3.12, 34.2, 0.1}, {2.1, 3.2, 4.3, -2.0}}
};
```

Wir können Arrays mit einer beliebigen Anzahl von Dimensionen haben, obwohl es wahrscheinlich ist, dass die meisten der erstellten Arrays eine oder zwei Dimensionen haben.

Durchlaufen eines Arrays mit Zeigern

```
#include <stdio.h>
#define SIZE (10)
int main()
{
    size_t i = 0;
    int *p = NULL;
    int a[SIZE];

    /* Setting up the values to be i*i */
    for(i = 0; i < SIZE; ++i)
    {
        a[i] = i * i;
    }

    /* Reading the values using pointers */
    for(p = a; p < a + SIZE; ++p)
    {
        printf("%d\n", *p);
    }

    return 0;
}
```

Bei der Initialisierung von `p` in der ersten `for` Schleifenbedingung *zerfällt* das Array `a` zu einem Zeiger auf sein erstes Element, wie es an fast allen Stellen der Fall wäre, an denen eine solche Arrayvariable verwendet wird.

Dann führt `++p` eine Zeigerarithmetik für den Zeiger `p` und geht die Elemente des Arrays nacheinander durch und referenziert sie, indem sie mit `*p` dereferenziert werden.

Übergeben von mehrdimensionalen Arrays an eine Funktion

Mehrdimensionale Arrays folgen den gleichen Regeln wie eindimensionale Arrays, wenn sie an eine Funktion übergeben werden. Die Kombination von Decay-to-Pointer, Operator-Vorrang und die zwei verschiedenen Arten, ein mehrdimensionales Array (Array von Arrays vs. Array von Zeigern) zu deklarieren, kann die Deklaration solcher Funktionen jedoch nicht intuitiv machen. Das folgende Beispiel zeigt, wie Sie mehrdimensionale Arrays richtig übergeben können.

```
#include <assert.h>
#include <stdlib.h>

/* When passing a multidimensional array (i.e. an array of arrays) to a
   function, it decays into a pointer to the first element as usual.  But only
   the top level decays, so what is passed is a pointer to an array of some fixed
   size (4 in this case). */
void f(int x[][4]) {
    assert(sizeof(*x) == sizeof(int) * 4);
}

/* This prototype is equivalent to f(int x[][4]).
   The parentheses around *x are required because [index] has a higher
   precedence than *expr, thus int *x[4] would normally be equivalent to int
   *(x[4]), i.e. an array of 4 pointers to int.  But if it's declared as a
```

```
   function parameter, it decays into a pointer and becomes int **x,
   which is not compatable with x[2][4]. */
void g(int (*x)[4]) {
    assert(sizeof(*x) == sizeof(int) * 4);
}

/* An array of pointers may be passed to this, since it'll decay into a pointer
   to pointer, but an array of arrays may not. */
void h(int **x) {
    assert(sizeof(*x) == sizeof(int*));
}

int main(void) {
    int foo[2][4];
    f(foo);
    g(foo);

    /* Here we're dynamically creating an array of pointers.  Note that the
       size of each dimension is not part of the datatype, and so the type
       system just treats it as a pointer to pointer, not a pointer to array
       or array of arrays. */
    int **bar = malloc(sizeof(*bar) * 2);
    assert(bar);
    for (size_t i = 0; i < 2; i++) {
        bar[i] = malloc(sizeof(*bar[i]) * 4);
        assert(bar[i]);
    }

    h(bar);

    for (size_t i = 0; i < 2; i++) {
        free(bar[i]);
    }
    free(bar);
}
```

Siehe auch

Übergabe von Arrays an Funktionen

Arrays online lesen: https://riptutorial.com/de/c/topic/322/arrays

Kapitel 6: Atomik

Syntax

- `#ifdef __STDC_NO_ATOMICS__`
- `# error this implementation needs atomics`
- `#endif`
- `#include <stdatomic.h>`
- vorzeichenloser _Atomic Counter = ATOMIC_VAR_INIT (0);

Bemerkungen

Atomics als Teil der C-Sprache ist eine optionale Funktion, die seit C11 verfügbar ist.

Ihr Zweck ist es, einen rennfreien Zugriff auf Variablen zu gewährleisten, die von verschiedenen Threads gemeinsam genutzt werden. Ohne atomare Qualifizierung wäre der Status einer gemeinsam genutzten Variablen undefiniert, wenn zwei Threads gleichzeitig darauf zugreifen. Eine Inkrementierungsoperation (`++`) könnte beispielsweise in mehrere Assembler-Anweisungen, einen Lesevorgang, den Zusatz selbst und einen Speicherbefehl aufgeteilt werden. Wenn ein anderer Thread dieselbe Operation ausführen würde, könnten die beiden Befehlssequenzen miteinander verflochten werden und zu einem inkonsistenten Ergebnis führen.

- **Typen:** Alle Objekttypen mit Ausnahme von `_Atomic` können mit `_Atomic` qualifiziert `_Atomic` .

- **Operatoren:** Alle **Operatoren zum** Lesen, Ändern und Schreiben (z. B. `++` oder `*=`) sind garantiert atomar.

- **Operationen:** Es gibt einige andere Operationen, die als generische Funktionen des Typs angegeben sind, z. B. `atomic_compare_exchange` .

- **Threads: Der** Zugriff darauf wird garantiert nicht zu einem Datenrennen führen, wenn von verschiedenen Threads darauf zugegriffen wird.

- **Signalhandler:** Atomic-Typen werden als *sperrungsfrei bezeichnet,* wenn alle Operationen auf ihnen zustandslos sind. In diesem Fall können sie auch verwendet werden, um Zustandsänderungen zwischen dem normalen Steuerfluss und einem Signalhandler zu behandeln.

- Es gibt nur einen Datentyp, der garantiert `atomic_flag` : `atomic_flag` . Dies ist ein minimaler Typ, dessen Operationen dazu bestimmt sind, effiziente Test-and-Set-Hardwareanweisungen abzubilden.

Andere `mtx_t` zur Vermeidung von Race-Bedingungen sind in der Thread-Schnittstelle von C11 verfügbar, insbesondere ein Mutex-Typ `mtx_t` , um Threads gegenseitig vom Zugriff auf kritische Daten oder kritische Codeabschnitte auszuschließen. Wenn keine Atomik verfügbar ist, müssen diese verwendet werden, um Rassen zu verhindern.

Examples

Atomik und Operatoren

Auf atomare Variablen kann gleichzeitig zwischen verschiedenen Threads zugegriffen werden, ohne dass Race-Bedingungen erstellt werden.

```
/* a global static variable that is visible by all threads */
static unsigned _Atomic active = ATOMIC_VAR_INIT(0);


int myThread(void* a) {
  ++active;         // increment active race free
  // do something
  --active;         // decrement active race free
  return 0;
}
```

Alle lvalue-Operationen (Operationen, die das Objekt ändern), die für den Basistyp zulässig sind, sind zulässig und führen nicht zu Race-Bedingungen zwischen verschiedenen Threads, die auf sie zugreifen.

- Operationen an atomaren Objekten sind in der Regel um Größenordnungen langsamer als normale Rechenoperationen. Dies umfasst auch einfache Lade- oder Speicheroperationen. Sie sollten sie daher nur für kritische Aufgaben verwenden.
- Übliche arithmetische Operationen und Zuweisungen wie `a = a+1;` In der Tat gibt es drei Operationen auf `a` : zuerst eine Ladung, dann eine Addition und schließlich ein Geschäft. Dies ist *nicht* frei von Rennen. Nur die Operation `a += 1;` und `a++;` sind.

Atomik online lesen: https://riptutorial.com/de/c/topic/4924/atomik

Kapitel 7: Aufzählungen

Bemerkungen

Enumerationen bestehen aus dem Schlüsselwort `enum` und einem optionalen *Bezeichner,* gefolgt von einer *Enumeratorliste,* die von geschweiften Klammern eingeschlossen ist.

Ein *Bezeichner* ist vom Typ `int` .

Die *Enumerator-Liste* hat mindestens ein *Enumerator-* Element.

Einem *Enumerator* kann optional ein konstanter Ausdruck vom Typ `int` "zugewiesen" werden.

Ein *Enumerator* ist konstant und ist entweder mit einem `char` , einer vorzeichenbehafteten Ganzzahl oder einer vorzeichenlosen Ganzzahl kompatibel. Was immer verwendet wird, ist implementierungsdefiniert . In jedem Fall sollte der verwendete Typ alle für die betreffende Aufzählung definierten Werte darstellen können.

Wenn kein konstanter Ausdruck „zugeordnet“ zu einem *Enumerator* und es ist der 1. Eintrag in einer *Aufzählungsliste* es Wert nimmt `0` , sonst den Wert des vorherigen Eintrags dauert , in der *Aufzählungsliste* plus 1.

Die Verwendung mehrerer "Zuweisungen" kann dazu führen, dass verschiedene *Aufzähler* derselben Aufzählung dieselben Werte tragen.

Examples

Einfache Aufzählung

Eine Aufzählung ist ein benutzerdefinierter Datentyp, der aus Integralkonstanten besteht, und jeder Integralkonstante wird ein Name zugewiesen. Schlüsselwort `enum` verwendet Aufzählungsdatentyp zu definieren.

Wenn Sie `enum` anstelle von `int` oder `string/ char*` , erhöhen Sie die Überprüfung der Kompilierzeit und vermeiden Fehler, wenn ungültige Konstanten übergeben werden. Außerdem dokumentieren Sie, welche Werte zulässig sind.

Beispiel 1

```
enum color{ RED, GREEN, BLUE };

void printColor(enum color chosenColor)
{
    const char *color_name = "Invalid color";
    switch (chosenColor)
    {
       case RED:
```

```
         color_name = "RED";
         break;

       case GREEN:
        color_name = "GREEN";
        break;

       case BLUE:
        color_name = "BLUE";
        break;
    }
    printf("%s\n", color_name);
}
```

Mit einer wie folgt definierten Hauptfunktion:

```
int main(){
    enum color chosenColor;
    printf("Enter a number between 0 and 2");
    scanf("%d", (int*)&chosenColor);
    printColor(chosenColor);
    return 0;
}
```

C99

Beispiel 2

(In diesem Beispiel werden festgelegte Initialisierer verwendet, die seit C99 standardisiert sind.)

```
enum week{ MON, TUE, WED, THU, FRI, SAT, SUN };

static const char* const dow[] = {
  [MON] = "Mon", [TUE] = "Tue", [WED] = "Wed",
  [THU] = "Thu", [FRI] = "Fri", [SAT] = "Sat", [SUN] = "Sun" };

void printDayOfWeek(enum week day)
{
   printf("%s\n", dow[day]);
}
```

Dasselbe Beispiel mit der Bereichsüberprüfung:

```
enum week{ DOW_INVALID = -1,
  MON, TUE, WED, THU, FRI, SAT, SUN,
  DOW_MAX };

static const char* const dow[] = {
  [MON] = "Mon", [TUE] = "Tue", [WED] = "Wed",
  [THU] = "Thu", [FRI] = "Fri", [SAT] = "Sat", [SUN] = "Sun" };

void printDayOfWeek(enum week day)
{
   assert(day > DOW_INVALID && day < DOW_MAX);
   printf("%s\n", dow[day]);
}
```

Typedef-Enum

Es gibt verschiedene Möglichkeiten und Konventionen, um eine Aufzählung zu benennen. Die erste besteht darin, einen *Tag-Namen* direkt nach dem `enum` Schlüsselwort zu verwenden.

```
enum color
{
    RED,
    GREEN,
    BLUE
};
```

Diese Aufzählung muss dann immer mit dem Schlüsselwort *und* dem Tag wie folgt verwendet werden:

```
enum color chosenColor = RED;
```

Wenn wir `typedef` direkt beim Deklarieren der `enum` , können wir den Tag-Namen weglassen und dann den Typ ohne das `enum` Schlüsselwort verwenden:

```
typedef enum
{
    RED,
    GREEN,
    BLUE
} color;

color chosenColor = RED;
```

Im letzteren Fall können wir es jedoch nicht als `enum color` , da wir den Tag-Namen nicht in der Definition verwendet haben. Eine gängige Konvention ist die Verwendung beider, so dass derselbe Name mit oder ohne `enum` . Dies hat den besonderen Vorteil, dass es mit C ++ kompatibel ist

```
enum color                /* as in the first example */
{
    RED,
    GREEN,
    BLUE
};
typedef enum color color; /* also a typedef of same identifier */

color chosenColor  = RED;
enum color defaultColor = BLUE;
```

Funktion:

```
void printColor()
{
    if (chosenColor == RED)
    {
        printf("RED\n");
```

```
    }
    else if (chosenColor == GREEN)
    {
        printf("GREEN\n");
    }
    else if (chosenColor == BLUE)
    {
        printf("BLUE\n");
    }
}
```

Weitere `typedef` zu `typedef` Sie unter Typedef

Aufzählung mit doppeltem Wert

Ein Aufzählungswert muss keinesfalls eindeutig sein:

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h> /* for printf() */


enum Dupes
{
   Base, /* Takes 0 */
   One, /* Takes Base + 1 */
   Two, /* Takes One + 1 */
   Negative = -1,
   AnotherZero /* Takes Negative + 1 == 0, sigh */
};

int main(void)
{
  printf("Base = %d\n", Base);
  printf("One = %d\n", One);
  printf("Two = %d\n", Two);
  printf("Negative = %d\n", Negative);
  printf("AnotherZero = %d\n", AnotherZero);

  return EXIT_SUCCESS;
}
```

Das Beispiel druckt:

```
Base = 0
One = 1
Two = 2
Negative = -1
AnotherZero = 0
```

Aufzählungskonstante ohne Typenname

Aufzählungstypen können auch deklariert werden, ohne ihnen einen Namen zu geben:

```
  enum { buffersize = 256, };
  static unsigned char buffer [buffersize] = { 0 };
```

Dies ermöglicht uns, Zeitkonstanten vom Typ `int` zu definieren, die wie in diesem Beispiel als Arraylänge verwendet werden können.

Aufzählungen online lesen: https://riptutorial.com/de/c/topic/5460/aufzahlungen

Kapitel 8: Auswahlanweisungen

Examples

if () Anweisungen

Eine der einfachsten Methoden zur Steuerung des Programmflusses besteht in der Verwendung von `if` Auswahlanweisungen. Ob ein Codeblock ausgeführt werden soll oder nicht, kann durch diese Anweisung entschieden werden.

Die Syntax für `if` Auswahlanweisungen in C könnte folgendermaßen lauten:

```
if(cond)
{
  statement(s);  /*to be executed, on condition being true*/
}
```

Zum Beispiel,

```
if (a > 1) {
    puts("a is larger than 1");
}
```

Dabei ist `a > 1` eine *Bedingung* , die als `true` ausgewertet werden muss, um die Anweisungen im `if` Block auszuführen. In diesem Beispiel wird "a ist größer als 1" nur gedruckt, wenn `a > 1` wahr ist.

`if` Auswahlanweisungen können die umlaufenden Klammern `{` und `}` auslassen, wenn nur eine Anweisung im Block enthalten ist. Das obige Beispiel kann umgeschrieben werden

```
if (a > 1)
    puts("a is larger than 1");
```

Für die Ausführung mehrerer Anweisungen innerhalb eines Blocks müssen jedoch die geschweiften Klammern verwendet werden.

Die *Bedingung* für `if` kann mehrere Ausdrücke enthalten. `if` führt die Aktion nur aus, wenn das Endergebnis des Ausdrucks wahr ist.

Zum Beispiel

```
if ((a > 1) && (b > 1)) {
    puts("a is larger than 1");
    a++;
}
```

führt die `printf` und `a++` wenn **sowohl** `a` als auch `b` größer als `1` .

if () ... else Anweisungen und Syntax

Während `if` führt eine Aktion nur aus, wenn ihre Bedingung als `true` ausgewertet wird. `if` / `else` ermöglicht Ihnen die Angabe der verschiedenen Aktionen, wenn die Bedingung `true` und die Bedingung `false` .

Beispiel:

```
if (a > 1)
    puts("a is larger than 1");
else
    puts("a is not larger than 1");
```

Genau wie bei der `if` Anweisung, wenn der Block innerhalb von `if` oder `else` nur aus einer Anweisung besteht, können die Klammern weggelassen werden (dies wird jedoch nicht empfohlen, da dies leicht unwillkürlich zu Problemen führen kann). Wenn es jedoch mehr als eine Anweisung im `if` oder `else` Block gibt, müssen die Klammern für diesen bestimmten Block verwendet werden.

```
if (a > 1)
{
    puts("a is larger than 1");
    a--;
}
else
{
    puts("a is not larger than 1");
    a++;
}
```

switch () Anweisungen

`switch` Anweisungen sind nützlich, wenn Sie möchten, dass Ihr Programm je nach Wert einer bestimmten Testvariablen viele verschiedene Dinge ausführt.

Ein Beispiel für die Verwendung der `switch` Anweisung sieht folgendermaßen aus:

```
int a = 1;

switch (a) {
case 1:
    puts("a is 1");
    break;
case 2:
    puts("a is 2");
    break;
default:
    puts("a is neither 1 nor 2");
    break;
}
```

Dieses Beispiel ist äquivalent zu

```
int a = 1;

if (a == 1) {
    puts("a is 1");
} else if (a == 2) {
    puts("a is 2");
} else {
    puts("a is neither 1 nor 2");
}
```

Wenn der Wert von `a` bei Verwendung der `switch` Anweisung 1 ist, `a is 1` ausgegeben. Wenn der Wert von `a` 2 ist, `a is 2` gedruckt. Andernfalls wird `a is neither 1 nor 2` gedruckt.

`case n:` wird verwendet, um zu beschreiben, wo der Ausführungsfluss einspringt, wenn der an die `switch` Anweisung übergebene Wert *n ist* . *n* muss eine konstante Kompilierungszeit sein, und dasselbe *n* darf höchstens einmal in einer `switch` Anweisung vorhanden sein.

`default:` wird verwendet, um zu beschreiben, dass der Wert mit keiner der Optionen für den `case n:` übereinstimmt. Es `default` , in jede switch-Anweisung einen `default` einzufügen, um unerwartetes Verhalten abzufangen.

Eine `break;` Eine Anweisung ist erforderlich, um aus dem `switch` zu springen .

Hinweis: Wenn Sie versehentlich vergessen, eine `break` nach dem Ende eines `case` hinzuzufügen, geht der Compiler davon aus, dass Sie "durchfallen " möchten, und alle nachfolgenden Case-Anweisungen (sofern vorhanden) werden ausgeführt (sofern keine *Break*- Anweisung in gefunden wird alle nachfolgenden Fälle), unabhängig davon, ob die nachfolgenden Fallanweisungen übereinstimmen oder nicht. Diese spezielle Eigenschaft wird zum Implementieren von Duff's Device verwendet . Dieses Verhalten wird häufig als Fehler in der C-Sprachspezifikation betrachtet.

Unten ist ein Beispiel, das die Auswirkungen der Abwesenheit von `break;` :

```
int a = 1;

switch (a) {
case 1:
case 2:
    puts("a is 1 or 2");
case 3:
    puts("a is 1, 2 or 3");
    break;
default:
    puts("a is neither 1, 2 nor 3");
    break;
}
```

Wenn der Wert von `a` 1 oder 2 ist, werden `a is 1 or 2` und `a is 1, 2 or 3` gedruckt. Wenn `a` 3 ist, wird nur `a is 1, 2 or 3` gedruckt. Andernfalls wird `a is neither 1, 2 nor 3` gedruckt.

Beachten Sie, dass der `default` nicht erforderlich ist, insbesondere wenn der Wertebereich, den Sie im `switch` , zum Zeitpunkt des Kompilierens beendet und bekannt ist.

Das beste Beispiel ist die Verwendung eines `switch` in einer `enum` .

```
enum msg_type { ACK, PING, ERROR };
void f(enum msg_type t)
{
  switch (t) {
  case ACK:
    // do nothing
    break;
  case PING:
    // do something
    break;
  case ERROR:
    // do something else
    break;
  }
}
```

Dies hat mehrere Vorteile:

- Die meisten Compiler melden eine Warnung, wenn Sie keinen Wert behandeln (dies würde nicht gemeldet werden, wenn ein `default` vorliegt).
- Aus dem gleichen Grund werden Sie beim Hinzufügen eines neuen Werts zur `enum` über alle Stellen informiert, an denen Sie vergessen haben, den neuen Wert zu behandeln (bei einem `default` müssten Sie Ihren Code manuell nach solchen Fällen durchsuchen).
- Der Leser muss nicht herausfinden, "was durch die `default:` ausgeblendet wird", ob es andere `enum` oder ob es sich um einen Schutz für "nur für den Fall" handelt. Und wenn es andere `enum` , hat der Codierer absichtlich den `default` für sie verwendet oder gibt es einen Fehler, der eingeführt wurde, als er den Wert addierte?
- Die Verarbeitung jedes `enum` macht den Code selbsterklärend, da Sie sich nicht hinter einem Platzhalter verstecken können. Sie müssen jeden von ihnen explizit behandeln.

Trotzdem können Sie nicht verhindern, dass jemand bösartigen Code schreibt:

```
enum msg_type t = (enum msg_type)666; // I'm evil
```

Sie können also vor dem Wechsel eine zusätzliche Prüfung hinzufügen, um sie zu erkennen, wenn Sie sie wirklich brauchen.

```
void f(enum msg_type t)
{
   if (!is_msg_type_valid(t)) {
      // Handle this unlikely error
   }

   switch(t) {
    // Same code than before
   }
}
```

if () ... else Ladder Chaining zwei oder mehr if () ... else-Anweisungen

Während mit der `if ()... else` -Anweisung nur ein (Standard-) Verhalten definiert werden kann, das auftritt, wenn die Bedingung innerhalb von `if ()` nicht erfüllt ist, können zwei oder mehr `if () ... else` Anweisungen miteinander verkettet werden mehr Verhaltensweisen vor zur letzten gehen `else` Zweig , die als eine „default“, falls vorhanden.

Beispiel:

```
int a = ... /* initialise to some value. */

if (a >= 1)
{
    printf("a is greater than or equals 1.\n");
}
else if (a == 0) //we already know that a is smaller than 1
{
    printf("a equals 0.\n");
}
else /* a is smaller than 1 and not equals 0, hence: */
{
    printf("a is negative.\n");
}
```

Geschachtelte if () ... else VS if () .. else Ladder

Geschachtelte `if()...else` Anweisungen benötigen mehr Ausführungszeit (sie sind langsamer) im Vergleich zu einem `if()...else` Ladder, da die verschachtelten `if()...else` Anweisungen alle inneren Bedingungsbefehle der äußeren prüfen bedingte `if()` Anweisung ist erfüllt, wohingegen der `if()..else` Ladder den Bedingungstest beendet, sobald eine der `if()` oder `else if()` Bedingungsanweisungen wahr ist.

Eine `if()...else` Ladder:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int a, b, c;
  printf("\nEnter Three numbers = ");
  scanf("%d%d%d", &a, &b, &c);
  if ((a < b) && (a < c))
  {
    printf("\na = %d is the smallest.", a);
  }
  else if ((b < a) && (b < c))
  {
    printf("\nb = %d is the smallest.", b);
  }
  else if ((c < a) && (c < b))
  {
    printf("\nc = %d is the smallest.", c);
  }
  else
  {
    printf("\nImprove your coding logic");
  }
```

```
  return 0;
}
```

Ist im allgemeinen Fall besser als das entsprechende verschachtelte `if()...else` :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int a, b, c;
  printf("\nEnter Three numbers = ");
  scanf("%d%d%d", &a, &b, &c);
  if (a < b)
  {
    if (a < c)
      {
        printf("\na = %d is the smallest.", a);
      }
    else
      {
        printf("\nc = %d is the smallest.", c);
      }
  }
  else
  {
    if(b < c)
    {
      printf("\nb = %d is the smallest.", b);
    }
    else
    {
      printf("\nc = %d is the smallest.", c);
    }
  }
  return 0;
}
```

Auswahlanweisungen online lesen: https://riptutorial.com/de/c/topic/3073/auswahlanweisungen

Kapitel 9: Behauptung

Einführung

Eine ***Assertion*** ist ein Prädikat, dass die dargestellte Bedingung in dem Moment wahr sein muss, in dem die Software von der Assertion getroffen wird. Am häufigsten sind ***einfache Zusicherungen*** , die zur Ausführungszeit überprüft werden. ***Statische Zusicherungen*** werden jedoch zur Kompilierzeit geprüft.

Syntax

- behaupten (Ausdruck)
- static_assert (Ausdruck, Nachricht)
- _Static_assert (Ausdruck, Nachricht)

Parameter

Parameter	Einzelheiten
Ausdruck	Ausdruck des Skalartyps.
Botschaft	String-Literal, das in die Diagnosemeldung aufgenommen werden soll.

Bemerkungen

Sowohl `assert` und `static_assert` sind Makros in definierten `assert.h` .

Die Definition von `assert` hängt von der Makro `NDEBUG` , die nicht durch die Standardbibliothek definiert ist. Wenn `NDEBUG` definiert ist, `assert` ist ein No-op:

```
#ifdef NDEBUG
#  define assert(condition) ((void) 0)
#else
#  define assert(condition) /* implementation defined */
#endif
```

Es gibt unterschiedliche Meinungen darüber, ob `NDEBUG` immer für Produktionszusammenstellungen verwendet werden sollte.

- Das Pro-Lager argumentiert , dass `assert` Anrufe `abort` und Assertion - Meldungen sind nicht hilfreich für den Endverbraucher, so das Ergebnis Benutzer nicht hilfreich ist. Wenn Sie schwerwiegende Bedingungen zum Einchecken des Produktionscodes haben, sollten Sie normale `if/else` Bedingungen verwenden und `exit` oder `quick_exit` zum `quick_exit` des Programms verwenden. Im Gegensatz zum `abort` kann das Programm einige Bereinigungen

atexit (über Funktionen, die mit atexit oder at_quick_exit registriert at_quick_exit).
- Das Con-Camp argumentiert, dass assert Aufrufe niemals im Produktionscode ausgelöst werden sollten, aber wenn dies der Fall ist, bedeutet dies, dass etwas dramatisch falsch ist und das Programm sich schlechter benimmt, wenn die Ausführung fortgesetzt wird. Daher ist es besser, die Aussagen im Produktionscode zu aktivieren, denn wenn sie feuern, ist die Hölle bereits gebrochen.
- Eine andere Möglichkeit besteht darin, ein Selbstbrühsystem für Assertions zu verwenden, das die Prüfung immer vornimmt, jedoch Fehler zwischen Entwicklung (wo abort angebracht ist) und Produktion (bei "unerwartetem internem Fehler - wenden Sie sich an den technischen Support" an, möglicherweise angemessener).

static_assert erweitert sich zu _Static_assert , einem Schlüsselwort. Die Bedingung wird zur Kompilierzeit geprüft, daher muss die condition ein konstanter Ausdruck sein. Es ist nicht erforderlich, dass dies in der Entwicklung und in der Produktion anders gehandhabt wird.

Examples

Vorbedingung und Nachbedingung

Ein Anwendungsfall für die Behauptung ist Vorbedingung und Nachbedingung. Dies kann sehr nützlich sein, um die Invariante und das vertragliche Design zu erhalten. Für ein Beispiel ist eine Länge immer Null oder positiv, daher muss diese Funktion einen Nullwert oder einen positiven Wert zurückgeben.

```
#include <stdio.h>
/* Uncomment to disable `assert()` */
/* #define NDEBUG */
#include <assert.h>

int length2 (int *a, int count)
{
    int i, result = 0;

    /* Precondition: */
    /* NULL is an invalid vector */
    assert (a != NULL);
    /* Number of dimensions can not be negative.*/
    assert (count >= 0);

    /* Calculation */
    for (i = 0; i < count; ++i)
    {
        result = result + (a[i] * a[i]);
    }

    /* Postcondition: */
    /* Resulting length can not be negative. */
    assert (result >= 0);
    return result;
}

#define COUNT 3
```

```
int main (void)
{
    int a[COUNT] = {1, 2, 3};
    int *b = NULL;
    int r;
    r = length2 (a, COUNT);
    printf ("r = %i\n", r);
    r = length2 (b, COUNT);
    printf ("r = %i\n", r);
    return 0;
}
```

Einfache Assertion

Eine Assertion ist eine Aussage, mit der behauptet wird, dass eine Tatsache wahr sein muss, wenn diese Codezeile erreicht ist. Zusicherungen sind hilfreich, um sicherzustellen, dass die erwarteten Bedingungen erfüllt werden. Wenn die an eine Assertion übergebene Bedingung wahr ist, gibt es keine Aktion. Das Verhalten bei falschen Bedingungen hängt von Compilerflags ab. Wenn Assertions aktiviert sind, führt eine falsche Eingabe zum sofortigen Programmstopp. Wenn sie deaktiviert sind, wird keine Aktion ausgeführt. Es ist üblich, Assertions in internen Builds und Debug-Builds zu aktivieren und sie in Release-Builds zu deaktivieren, obwohl Assertions häufig in Release aktiviert werden. (Ob die Beendigung besser oder schlechter als Fehler ist, hängt vom Programm ab.) Assertions sollten nur zum Abfangen von internen Programmierfehlern verwendet werden, was normalerweise bedeutet, dass schlechte Parameter übergeben werden.

```
#include <stdio.h>
/* Uncomment to disable `assert()` */
/* #define NDEBUG */
#include <assert.h>

int main(void)
{
    int x = -1;
    assert(x >= 0);

    printf("x = %d\n", x);
    return 0;
}
```

Mögliche Ausgabe mit `NDEBUG` undefined:

```
a.out: main.c:9: main: Assertion `x >= 0' failed.
```

Mögliche Ausgabe mit `NDEBUG` definiert:

```
x = -1
```

Es ist `NDEBUG` , `NDEBUG` global zu definieren, sodass Sie Ihren Code mit allen Zusicherungen entweder `NDEBUG` oder ausschalten können. Eine einfache Möglichkeit, dies zu tun, ist die Definition von `NDEBUG` als Option für den Compiler oder in einem gemeinsam genutzten Konfigurationsheader (z. B. `config.h`).

Statische Assertion

C11

Statische Zusicherungen werden verwendet, um zu überprüfen, ob eine Bedingung erfüllt ist, wenn der Code kompiliert wird. Ist dies nicht der Fall, muss der Compiler eine Fehlermeldung ausgeben und den Kompiliervorgang stoppen.

Eine statische Zusicherung ist eine, die zur Kompilierzeit geprüft wird, nicht zur Laufzeit. Die Bedingung muss ein konstanter Ausdruck sein, und falls false, wird ein Compilerfehler ausgegeben. Das erste Argument, die geprüfte Bedingung, muss ein konstanter Ausdruck sein und das zweite ein Zeichenkettenliteral.

Im Gegensatz zu assert ist `_Static_assert` ein Schlüsselwort. Ein komfortables Makro `static_assert` ist in `<assert.h>` .

```
#include <assert.h>

enum {N = 5};
_Static_assert(N == 5, "N does not equal 5");
static_assert(N > 10, "N is not greater than 10");  /* compiler error */
```

C99

Vor C11 gab es keine direkte Unterstützung für statische Assertions. In C99 konnten statische Assertions jedoch mit Makros emuliert werden, die einen Kompilierungsfehler auslösen würden, wenn die Kompilierzeitbedingung falsch war. Im Gegensatz zu `_Static_assert` muss der zweite Parameter ein geeigneter Token-Name sein, damit ein Variablenname erstellt werden kann. Wenn die Assertion fehlschlägt, wird der Variablenname im Compiler-Fehler angezeigt, da diese Variable in einer syntaktisch falschen Array-Deklaration verwendet wurde.

```
#define STATIC_MSG(msg, l) STATIC_MSG2(msg, l)
#define STATIC_MSG2(msg,l) on_line_##l##__##msg
#define STATIC_ASSERT(x, msg) extern char STATIC_MSG(msg, __LINE__) [(x)?1:-1]

enum { N = 5 };
STATIC_ASSERT(N == 5, N_must_equal_5);
STATIC_ASSERT(N > 5, N_must_be_greater_than_5); /* compile error */
```

Vor C99 konnten Sie keine Variablen an beliebigen Stellen in einem Block deklarieren. Daher müssen Sie bei der Verwendung dieses Makros äußerst vorsichtig sein und sicherstellen, dass nur dort angezeigt wird, wo eine Variablendeklaration gültig ist.

Geltendmachung eines unerreichbaren Codes

Wenn während der Entwicklung bestimmte Codepfade von der Reichweite des Steuerungsflusses ferngehalten werden müssen, können Sie `assert(0)` um anzuzeigen, dass eine solche Bedingung fehlerhaft ist:

```
switch (color) {
```

```
    case COLOR_RED:
    case COLOR_GREEN:
    case COLOR_BLUE:
        break;

    default:
        assert(0);
}
```

Wenn das Argument des `assert()` -Makros den Wert false ergibt, schreibt das Makro Diagnoseinformationen in den Standardfehlerstrom und bricht das Programm ab. Diese Informationen enthalten die Datei- und Zeilennummer der `assert()` Anweisung und können beim Debuggen sehr hilfreich sein. Asserts können durch Definieren des Makros `NDEBUG` .

Eine andere Möglichkeit, ein Programm zu beenden, wenn ein Fehler auftritt, sind die Standardbibliotheksfunktionen `exit` , `quick_exit` oder `abort` . `exit` und `quick_exit` nehmen ein Argument an, das an Ihre Umgebung zurückgegeben werden kann. `abort()` (und damit `assert`) kann eine wirklich schwerwiegende Beendigung Ihres Programms sein, und bestimmte Bereinigungen, die andernfalls am Ende der Ausführung durchgeführt würden, werden möglicherweise nicht durchgeführt.

Der Hauptvorteil von `assert()` ist, dass Debugging-Informationen automatisch gedruckt werden. Das Aufrufen von `abort()` hat den Vorteil, dass es nicht wie ein Assert deaktiviert werden kann, aber möglicherweise keine Debugging-Informationen angezeigt werden. In einigen Situationen kann die Verwendung beider Konstrukte von Vorteil sein:

```
if (color == COLOR_RED || color == COLOR_GREEN) {
   ...
} else if (color == COLOR_BLUE) {
   ...
} else {
   assert(0), abort();
}
```

Wenn Assert `assert()` *aktiviert sind* , gibt der `assert()` Aufruf die Debug-Informationen aus und beendet das Programm. Die Ausführung erreicht niemals den `abort()` Aufruf. Wenn Asserts *deaktiviert sind* , macht der `assert()` Aufruf nichts und `abort()` wird aufgerufen. Dadurch wird sichergestellt, dass das Programm für diese Fehlerbedingung *immer* beendet wird. Durch Aktivieren und Deaktivieren wird nur die Auswirkung aktiviert, ob die Debug-Ausgabe gedruckt wird.

Sie sollten eine solche `assert` niemals im Produktionscode belassen, da die Debug-Informationen für Endbenutzer nicht hilfreich sind und der `abort` im Allgemeinen eine viel zu schwerwiegende Beendigung darstellt, die die für `exit` oder `quick_exit` installierten Cleanup-Handler `quick_exit` .

Bestätigen Sie Fehlermeldungen

Es gibt einen Trick, der neben einer Assertion eine Fehlermeldung anzeigen kann. Normalerweise würden Sie Code so schreiben

```
void f(void *p)
{
    assert(p != NULL);
    /* more code */
}
```

Wenn die Zusicherung fehlgeschlagen ist, würde eine Fehlermeldung ähneln

> Assertion fehlgeschlagen: p! = NULL, Datei main.c, Zeile 5

Sie können jedoch auch das logische AND (`&&`) verwenden, um eine Fehlermeldung zu geben

```
void f(void *p)
{
    assert(p != NULL && "function f: p cannot be NULL");
    /* more code */
}
```

Wenn die Zusicherung fehlschlägt, wird eine Fehlermeldung in etwa folgendermaßen angezeigt

> Assertion fehlgeschlagen: p! = NULL && "Funktion f: p kann nicht NULL sein", Datei main.c, Zeile 5

Der Grund, warum dies funktioniert, ist, dass ein Zeichenfolgenliteral immer den Wert ungleich Null (wahr) ergibt. Das Hinzufügen von `&& 1` zu einem booleschen Ausdruck hat keine Auswirkung. Das Hinzufügen von `&& "error message"` hat also keine Auswirkung, außer dass der Compiler den gesamten fehlgeschlagenen Ausdruck anzeigt.

Behauptung online lesen: https://riptutorial.com/de/c/topic/555/behauptung

Kapitel 10: Bemerkungen

Einführung

Kommentare werden verwendet, um der lesenden Person etwas mitzuteilen. Kommentare werden vom Compiler wie ein Leerzeichen behandelt und ändern nichts an der tatsächlichen Bedeutung des Codes. Es gibt zwei Syntaxen für Kommentare in C, das Original `/* */` und das etwas neuere `//` . Einige Dokumentationssysteme verwenden speziell formatierte Kommentare, um die Dokumentation für Code zu erstellen.

Syntax

- `/*...*/`
- `//...` (nur C99 und höher)

Examples

/ * * / abgegrenzte Kommentare

Ein Kommentar beginnt mit einem Schrägstrich gefolgt von einem Sternchen (`/*`) und endet, sobald ein Sternchen gefolgt von einem Schrägstrich (`*/`) erscheint. Alles, was sich zwischen diesen Zeichenkombinationen befindet, ist ein Kommentar und wird vom Compiler als Leerzeichen behandelt (grundsätzlich ignoriert).

```
/* this is a comment */
```

Der Kommentar oben ist ein einzeiliger Kommentar. Kommentare dieses `/*` -Typs können sich über mehrere Zeilen erstrecken:

```
/* this is a
multi-line
comment */
```

Obwohl es nicht unbedingt erforderlich ist, besteht eine übliche Stilkonvention mit mehrzeiligen Kommentaren darin, führende Zeilen und Sternchen in die Zeilen nach der ersten Zeile und die Zeichen `/*` und `*/` in neue Zeilen einzufügen, sodass sie alle in einer Reihe angeordnet sind:

```
/*
 * this is a
 * multi-line
 * comment
 */
```

Die zusätzlichen Sternchen haben keine funktionellen Auswirkungen auf den Kommentar, da keines der Zeichen einen entsprechenden Schrägstrich hat.

Diese `/*` Art der Kommentare können auf ihre eigene Linie verwendet werden, am Ende einer Codezeile oder sogar innerhalb von Codezeilen:

```
/* this comment is on its own line */
if (x && y) { /*this comment is at the end of a line */
    if ((complexCondition1) /* this comment is within a line of code */
            && (complexCondition2)) {
        /* this comment is within an if, on its own line */
    }
}
```

Kommentare können nicht verschachtelt werden. Dies liegt daran, dass nachfolgende `/*` (als Teil des Kommentars) ignoriert werden und das erste `*/` erreichtes Ende des Kommentars behandelt wird. Der Kommentar im folgenden Beispiel *wird nicht funktionieren* :

```
/* outer comment, means this is ignored => /* attempted inner comment */ <= ends the comment,
not this one => */
```

Informationen zum Kommentieren von Codeblöcken, die Kommentare dieses Typs enthalten, die andernfalls verschachtelt wären, finden Sie im Kommentarbeispiel unter Verwendung des Präprozessors

// abgegrenzte Kommentare

C99

Mit C99 wurden einzeilige Kommentare im C ++ - Stil eingeführt. Diese Art von Kommentar beginnt mit zwei Schrägstrichen und geht bis zum Zeilenende:

```
// this is a comment
```

Diese Art von Kommentar erlaubt keine mehrzeiligen Kommentare. Es ist jedoch möglich, einen Kommentarblock durch Hinzufügen mehrerer einzeiliger Kommentare nacheinander zu erstellen:

```
// each of these lines are a single-line comment
// note how each must start with
// the double forward-slash
```

Diese Art von Kommentar kann in einer eigenen Zeile oder am Ende einer Codezeile verwendet werden. Da sie jedoch *bis zum Ende der Zeile* laufen, dürfen sie *nicht* in einer Codezeile verwendet werden

```
// this comment is on its own line
if (x && y) { // this comment is at the end of a line
    // this comment is within an if, on its own line
}
```

Kommentieren mit dem Präprozessor

Große Teile des Codes kann auch sein „ kommentierte out“ mit den Präprozessordirektiven `#if 0` und `#endif` . Dies ist nützlich, wenn der Code mehrzeilige Kommentare enthält, die sonst nicht verschachtelt würden.

```
#if 0 /* Starts the "comment", anything from here on is removed by preprocessor */

/* A large amount of code with multi-line comments */
int foo()
{
    /* lots of code */
    ...

    /* ... some comment describing the if statement ... */
    if (someTest) {
        /* some more comments */
        return 1;
    }

    return 0;
}

#endif /* 0 */

/* code from here on is "uncommented" (included in compiled executable) */
...
```

Möglicher Fallstrick durch Trigraphen

C99

Beim Schreiben `//` Kommentare ist es möglich, einen Tippfehler zu machen, der die erwartete Operation beeinflusst. Wenn man schreibt:

```
int x = 20;  // Why did I do this??/
```

Das `/` am Ende war ein Tippfehler, wird nun aber in `\` interpretiert. Dies liegt daran, dass das `??/` ein Trigraph bildet.

Das `??/` Trigraph ist eigentlich eine lange Schreibweise für `\` , das das Linienfortsetzungssymbol ist. Dies bedeutet, dass der Compiler der Meinung ist, dass die nächste Zeile eine Fortsetzung der aktuellen Zeile ist, d. H. Eine Fortsetzung des Kommentars, der möglicherweise nicht wie beabsichtigt ist.

```
int foo = 20; // Start at 20 ??/
int bar = 0;

// The following will cause a compilation error (undeclared variable 'bar')
// because 'int bar = 0;' is part of the comment on the preceding line
bar += foo;
```

Bemerkungen online lesen: https://riptutorial.com/de/c/topic/10670/bemerkungen

Kapitel 11: Bitfelder

Einführung

Die meisten Variablen in C haben eine Größe, die eine ganzzahlige Anzahl von Bytes ist. Bitfelder sind Teil einer Struktur, die nicht unbedingt eine ganzzahlige Anzahl von Bytes belegt. Sie können eine beliebige Anzahl von Bits. Mehrere Bitfelder können in eine einzige Speichereinheit gepackt werden. Sie sind Teil von Standard C, aber es gibt viele Aspekte, die durch die Implementierung definiert werden. Sie sind einer der am wenigsten tragbaren Teile von C.

Syntax

- Typenbezeichnerkennung: Größe;

Parameter

Parameter	Beschreibung
Typbezeichner	`signed` , `unsigned` , `int` oder `_Bool`
Kennung	Der Name für dieses Feld in der Struktur
Größe	Die Anzahl der Bits, die für dieses Feld verwendet werden sollen

Bemerkungen

Die einzigen portablen Typen für Bitfelder sind `signed` , `unsigned` oder `_Bool` . Die Ebene `int` - Typ kann verwendet werden, aber der Standard sagt (§6.7.2¶5) *... für Bitfelder, es ist* die *Implementierung definiert* , *ob der Bezeichner* `int` *den gleichen Typ wie bezeichnet* `signed int` *oder vom gleichen Typ wie* `unsigned int` *.*

Andere ganzzahlige Typen können von einer bestimmten Implementierung zugelassen werden, deren Verwendung ist jedoch nicht portierbar.

Examples

Bitfelder

Ein einfaches Bitfeld kann verwendet werden, um Dinge zu beschreiben, an denen eine bestimmte Anzahl von Bits beteiligt ist.

```
struct encoderPosition {
   unsigned int encoderCounts : 23;
```

```
   unsigned int encoderTurns  : 4;
   unsigned int _reserved     : 5;
};
```

In diesem Beispiel betrachten wir einen Encoder mit 23 Bit einfacher Genauigkeit und 4 Bit, um Multiturn zu beschreiben. Bitfelder werden häufig verwendet, wenn eine Schnittstelle zu Hardware hergestellt wird, die Daten ausgibt, die einer bestimmten Anzahl von Bits zugeordnet sind. Ein anderes Beispiel könnte die Kommunikation mit einem FPGA sein, bei dem der FPGA Daten in 32-Bit-Abschnitten in Ihren Speicher schreibt, um Hardware-Lesevorgänge zu ermöglichen:

```
struct FPGAInfo {
    union {
        struct bits {
            unsigned int bulb1On  : 1;
            unsigned int bulb2On  : 1;
            unsigned int bulb1Off : 1;
            unsigned int bulb2Off : 1;
            unsigned int jetOn    : 1;
        };
        unsigned int data;
   };
};
```

Für dieses Beispiel haben wir ein häufig verwendetes Konstrukt gezeigt, um auf die Daten in ihren einzelnen Bits zugreifen zu können oder das Datenpaket als Ganzes zu schreiben (emulieren, was der FPGA tun könnte). Wir könnten dann auf die Bits wie folgt zugreifen:

```
FPGAInfo fInfo;
fInfo.data = 0xFF34F;
if (fInfo.bits.bulb1On) {
    printf("Bulb 1 is on\n");
}
```

Dies gilt, aber gemäß C99-Standard 6.7.2.1, Punkt 10:

> Die Reihenfolge der Zuordnung von Bitfeldern innerhalb einer Einheit (von hoher bis niedriger oder von niedriger nach hoher Ordnung) ist implementierungsdefiniert.

Wenn Sie Bitfelder auf diese Weise definieren, müssen Sie sich der Endianität bewusst sein. Daher kann es erforderlich sein, eine Präprozessor-Direktive zu verwenden, um die Endianness der Maschine zu überprüfen. Ein Beispiel dafür folgt:

```
typedef union {
    struct bits {
#if defined(WIN32) || defined(LITTLE_ENDIAN)
    uint8_t commFailure :1;
    uint8_t hardwareFailure :1;
    uint8_t _reserved :6;
#else
    uint8_t _reserved :6;
    uint8_t hardwareFailure :1;
    uint8_t commFailure :1;
#endif
    };
```

```
    uint8_t data;
} hardwareStatus;
```

Verwenden von Bitfeldern als kleine Ganzzahlen

```
#include <stdio.h>

int main(void)
{
    /* define a small bit-field that can hold values from 0 .. 7 */
    struct
    {
        unsigned int uint3: 3;
    } small;

    /* extract the right 3 bits from a value */
    unsigned int value = 255 - 2; /* Binary 11111101 */
    small.uint3 = value;          /* Binary      101 */
    printf("%d", small.uint3);

    /* This is in effect an infinite loop */
    for (small.uint3 = 0; small.uint3 < 8; small.uint3++)
    {
        printf("%d\n", small.uint3);
    }

    return 0;
}
```

Bitfeldausrichtung

Bitfelder geben die Möglichkeit, Strukturfelder zu deklarieren, die kleiner als die Zeichenbreite sind. Bitfelder werden mit einer Byte- oder Wortebenenmaske implementiert. Das folgende Beispiel ergibt eine Struktur von 8 Bytes.

```
struct C
{
    short s;            /* 2 bytes */
    char  c;            /* 1 byte */
    int   bit1 : 1;     /* 1 bit */
    int   nib  : 4;     /* 4 bits padded up to boundary of 8 bits. Thus 3 bits are padded */
    int   sept : 7;     /* 7 Bits septet, padded up to boundary of 32 bits. */
};
```

In den Kommentaren wird ein mögliches Layout beschrieben. Da der Standard besagt, dass *die Ausrichtung der adressierbaren Speichereinheit nicht angegeben ist* , sind auch andere Layouts möglich.

Ein unbenanntes Bitfeld kann eine beliebige Größe haben, sie kann jedoch nicht initialisiert oder referenziert werden.

Ein Bitfeld mit der Breite Null kann nicht benannt werden und richtet das nächste Feld an der durch den Datentyp des Bitfelds definierten Grenze aus. Dies wird durch Auffüllen von Bits zwischen den Bitfeldern erreicht.

Die Größe der Struktur 'A' beträgt 1 Byte.

```
struct A
{
    unsigned char c1 : 3;
    unsigned char c2 : 4;
    unsigned char c3 : 1;
};
```

In der Struktur B überspringt das erste unbenannte Bitfeld 2 Bits; Das Bitfeld mit der Breite Null nach `c2` bewirkt, dass `c3` an der char-Grenze beginnt (also werden 3 Bits zwischen `c2` und `c3` übersprungen. Nach `c4` gibt es 3 Auffüllbits. Daher beträgt die Größe der Struktur 2 Byte.

```
struct B
{
    unsigned char c1 : 1;
    unsigned char    : 2;    /* Skips 2 bits in the layout */
    unsigned char c2 : 2;
    unsigned char    : 0;    /* Causes padding up to next container boundary */
    unsigned char c3 : 4;
    unsigned char c4 : 1;
};
```

Wann sind Bitfelder nützlich?

Ein Bitfeld wird verwendet, um viele Variablen in einem Objekt zusammenzufassen, ähnlich einer Struktur. Dies ermöglicht einen geringeren Speicherbedarf und ist besonders in einer eingebetteten Umgebung nützlich.

```
 e.g. consider the following variables having the ranges as given below.
 a --> range 0 - 3
 b --> range 0 - 1
 c --> range 0 - 7
 d --> range 0 - 1
 e --> range 0 - 1
```

Wenn wir diese Variablen separat deklarieren, muss jede mindestens eine 8-Bit-Ganzzahl sein und der gesamte erforderliche Speicherplatz beträgt 5 Byte. Darüber hinaus verwenden die Variablen nicht den gesamten Bereich einer vorzeichenlosen 8-Bit-Ganzzahl (0-255). Hier können wir Bitfelder verwenden.

```
typedef struct {
   unsigned int a:2;
   unsigned int b:1;
   unsigned int c:3;
   unsigned int d:1;
   unsigned int e:1;
} bit_a;
```

Auf die Bitfelder in der Struktur wird wie auf jede andere Struktur zugegriffen. Der Programmierer muss darauf achten, dass die Variablen in den Bereich geschrieben werden. Wenn außerhalb des Bereichs ist das Verhalten undefiniert.

```
int main(void)
{
   bit_a bita_var;
   bita_var.a = 2;              // to write into element a
   printf ("%d",bita_var.a);    // to read from element a.
   return 0;
}
```

Oft möchte der Programmierer die Menge der Bitfelder auf Null setzen. Dies kann Element für Element erfolgen, aber es gibt eine zweite Methode. Erstellen Sie einfach eine Vereinigung der obigen Struktur mit einem vorzeichenlosen Typ, der größer oder gleich der Größe der Struktur ist. Dann kann der gesamte Satz von Bitfeldern auf Null gesetzt werden, indem diese vorzeichenlose Ganzzahl auf Null gesetzt wird.

```
typedef union {
    struct {
       unsigned int a:2;
       unsigned int b:1;
       unsigned int c:3;
       unsigned int d:1;
       unsigned int e:1;
    };
    uint8_t data;
} union_bit;
```

Die Verwendung ist wie folgt

```
int main(void)
{
   union_bit un_bit;
   un_bit.data = 0x00;        // clear the whole bit-field
   un_bit.a = 2;              // write into element a
   printf ("%d",un_bit.a);    // read from element a.
   return 0;
}
```

Zusammenfassend werden Bitfelder im Allgemeinen in Situationen mit eingeschränktem Speicher verwendet, in denen viele Variablen vorhanden sind, die begrenzte Bereiche annehmen können.

Don'ts für Bitfelder

1. Bitfelder, Zeiger auf Bitfelder und Funktionen, die Bitfelder zurückgeben, sind nicht zulässig.
2. Der Adressoperator (&) kann nicht auf Bitfeldmitglieder angewendet werden.
3. Der Datentyp eines Bitfelds muss breit genug sein, um die Größe des Felds zu enthalten.
4. Der Operator `sizeof()` kann nicht auf ein Bitfeld angewendet werden.
5. Es gibt keine Möglichkeit, eine `typedef` für ein Bitfeld isoliert zu erstellen (obwohl Sie sicherlich eine `typedef` für eine Struktur mit Bitfeldern erstellen können).

```
typedef struct mybitfield
{
    unsigned char c1 : 20;   /* incorrect, see point 3 */
    unsigned char c2 : 4;    /* correct */
    unsigned char c3 : 1;
```

```
    unsigned int x[10]: 5;    /* incorrect, see point 1 */
} A;

int SomeFunction(void)
{
    // Somewhere in the code
    A a = { … };
    printf("Address of a.c2 is %p\n", &a.c2);      /* incorrect, see point 2 */
    printf("Size of a.c2 is %zu\n", sizeof(a.c2)); /* incorrect, see point 4 */
}
```

Bitfelder online lesen: https://riptutorial.com/de/c/topic/1930/bitfelder

Kapitel 12: Boolean

Bemerkungen

Um den vordefinierten Typ `_Bool` und den Header `<stdbool.h>` , müssen Sie die C99 / C11-Versionen von C verwenden.

Um Compiler-Warnungen und möglicherweise Fehler zu vermeiden, sollten Sie das `typedef` / `define` Beispiel nur verwenden `define` wenn Sie C89 und frühere Versionen der Sprache verwenden.

Examples

Stdbool.h verwenden

C99

Mit der `stdbool.h` können Sie `bool` als booleschen Datentyp verwenden. `true` ergibt `1` und `false` ergibt `0` .

```
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    bool x = true;  /* equivalent to bool x = 1; */
    bool y = false; /* equivalent to bool y = 0; */
    if (x)  /* Functionally equivalent to if (x != 0) or if (x != false) */
    {
        puts("This will print!");
    }
    if (!y) /* Functionally equivalent to if (y == 0) or if (y == false) */
    {
        puts("This will also print!");
    }
}
```

`bool` ist nur eine schöne Schreibweise für den Datentyp `_Bool` . Es gibt spezielle Regeln, wenn Zahlen oder Zeiger in ihn umgewandelt werden.

#Define verwenden

C allen Versionen wird behandeln effektiv jeden Integer - Wert anders als `0` als `true` für Vergleichsoperator und den ganzzahligen Wert `0` als `false` . Wenn Sie nicht über `_Bool` oder `bool` als von C99 zur Verfügung, könnten Sie einen Booleschen Datentyp in C unter Verwendung simulieren `#define` Makros, und Sie können immer noch solche Dinge in Legacy - Code finden.

```
#include <stdio.h>

#define bool int
```

```
#define true 1
#define false 0

int main(void) {
    bool x = true;  /* Equivalent to int x = 1; */
    bool y = false; /* Equivalent to int y = 0; */
    if (x) /* Functionally equivalent to if (x != 0) or if (x != false) */
    {
        puts("This will print!");
    }
    if (!y) /* Functionally equivalent to if (y == 0) or if (y == false) */
    {
        puts("This will also print!");
    }
}
```

`<stdbool.h>` Sie dies nicht in neuem Code ein, da die Definition dieser Makros mit der modernen Verwendung von `<stdbool.h>` .

Verwendung des eingebauten (integrierten) Typs _Bool

C99

`_Bool` in der C-Standardversion C99 `_Bool` ist ebenfalls ein nativer C-Datentyp. Es kann die Werte `0` (für *falsch*) und `1` (für *wahr*) halten.

```
#include <stdio.h>

int main(void) {
    _Bool x = 1;
    _Bool y = 0;
    if(x) /* Equivalent to if (x == 1) */
    {
        puts("This will print!");
    }
    if (!y) /* Equivalent to if (y == 0) */
    {
        puts("This will also print!");
    }
}
```

`_Bool` ist ein Integer-Typ, hat jedoch spezielle Regeln für Konvertierungen von anderen Typen. Das Ergebnis ist analog zur Verwendung anderer Typen in `if` Ausdrücken . Im Folgenden

```
_Bool z = X;
```

- Wenn `X` einen arithmetischen Typ hat (ist eine beliebige Zahl), wird `z` zu `0` wenn `X == 0` . Ansonsten wird `z` zu `1` .
- Wenn `X` einen Zeigertyp hat, wird `z` zu `0` wenn `X` ein Nullzeiger ist, andernfalls `1` .

So verwenden schöner Schreibweisen `bool` , `false` und `true` Sie verwenden müssen `<stdbool.h>` .

Ganzzahlen und Zeiger in booleschen Ausdrücken.

Alle Ganzzahlen oder Zeiger können in einem Ausdruck verwendet werden, der als "Wahrheitswert" interpretiert wird.

```
int main(int argc, char* argv[]) {
  if (argc % 4) {
    puts("arguments number is not divisible by 4");
  } else {
    puts("argument number is divisible by 4");
  }
...
```

Der Ausdruck `argc % 4` wird ausgewertet und führt zu einem der Werte `0` , `1` , `2` oder `3` . Der erste, `0` ist der einzige Wert, der "false" ist und die Ausführung in den `else` Teil bringt. Alle anderen Werte sind "true" und gehen in den `if` Teil über.

```
double* A = malloc(n*sizeof *A);
if (!A) {
   perror("allocation problems");
   exit(EXIT_FAILURE);
}
```

Hier wird der Zeiger `A` ausgewertet und wenn es sich um einen Nullzeiger handelt, wird ein Fehler erkannt und das Programm beendet.

Viele Leute schreiben lieber etwas als `A == NULL` , aber wenn Sie solche Zeigervergleiche als Teil anderer komplizierter Ausdrücke haben, werden die Dinge schnell schwer zu lesen.

```
char const* s = ....;   /* some pointer that we receive */
if (s != NULL && s[0] != '\0' && isalpha(s[0])) {
   printf("this starts well, %c is alphabetic\n", s[0]);
}
```

Um dies zu überprüfen, müssen Sie einen komplizierten Code im Ausdruck scannen und die Präferenzen des Bedieners beachten.

```
char const* s = ....;   /* some pointer that we receive */
if (s && s[0] && isalpha(s[0])) {
   printf("this starts well, %c is alphabetic\n", s[0]);
}
```

ist relativ einfach zu erfassen: Wenn der Zeiger gültig ist, prüfen wir, ob das erste Zeichen ungleich Null ist, und prüfen, ob es sich um einen Buchstaben handelt.

Definieren eines Bool-Typs mit Typedef

In Anbetracht der Tatsache, dass die meisten Debugger keine # `#define` Makros kennen, aber `enum` überprüfen können, ist es möglicherweise wünschenswert, Folgendes zu tun:

```
#if __STDC_VERSION__ < 199900L
typedef enum { false, true } bool;
/* Modern C code might expect these to be macros. */
```

```
# ifndef bool
#  define bool bool
# endif
# ifndef true
#  define true true
# endif
# ifndef false
#  define false false
# endif
#else
# include <stdbool.h>
#endif

/* Somewhere later in the code ... */
bool b = true;
```

Dadurch können Compiler für ältere Versionen von C funktionieren, sie bleiben jedoch vorwärtskompatibel, wenn der Code mit einem modernen C-Compiler kompiliert wird.

Weitere Informationen zu `typedef` finden Sie unter Typedef. Weitere Informationen zu `enum` Sie unter Aufzählungen

Boolean online lesen: https://riptutorial.com/de/c/topic/3336/boolean

Kapitel 13: Dateien und E / A-Streams

Syntax

- #include <stdio.h> / * Fügen Sie dies ein, um einen der folgenden Abschnitte zu verwenden * /
- FILE * fopen (const char * pfad, const char * -Modus); / * Öffne einen Stream in der Datei unter *Pfad* mit dem angegebenen *Modus* * /
- FILE * freopen (const char * pfad, const char * Modus, FILE * stream); / * Einen vorhandenen Stream in der Datei unter *Pfad* im angegebenen *Modus erneut* öffnen. * /
- int fclose (FILE * stream); / * Einen geöffneten Stream schließen * /
- size_t fread (void * ptr, size_t size, size_t nmemb, FILE * stream); / * Höchstens *nmemb* Elemente der *Größe* Lesen jeder aus dem *Stream* - Bytes und schreibt sie in *PTR.* Gibt die Anzahl der gelesenen Elemente zurück. * /
- size_t fwrite (const void * ptr, size_t size, size_t nmemb, FILE * stream); / * Schreibe *nmemb* Elemente *size* Bytes jeweils von *PTR* in den *Stream.* Gibt die Anzahl der geschriebenen Elemente zurück. * /
- int fseek (FILE * stream, langer Offset, int woce); / * Der Cursor des Stroms Stellen zu *kompensieren,* bezogen auf die von *woher* gesagt Offset und liefert 0 , wenn es erfolgreich war. * /
- long ftell (FILE * stream); / * Gibt den Offset der aktuellen Cursorposition vom Anfang des Streams zurück. * /
- Zurückspulen (FILE * stream); / * Setzt die Cursorposition auf den Anfang der Datei. * /
- int fprintf (FILE * fout, const char * fmt, ...); / * Schreibt den printf-Formatstring in fout * /
- FILE * stdin; / * Standardeingangsstrom * /
- FILE * stdout; / * Standardausgabestrom * /
- FILE * stderr; / * Standardfehlerstrom * /

Parameter

Parameter	Einzelheiten
const char * Modus	Eine Zeichenfolge, die den Eröffnungsmodus des durch Dateien gesicherten Streams beschreibt. Siehe Anmerkungen zu möglichen Werten.
int woher	`SEEK_SET` vom Anfang der Datei aus gesetzt werden, `SEEK_END` vom Ende der Datei oder `SEEK_CUR` , um den aktuellen `SEEK_CUR` . Hinweis: `SEEK_END` ist nicht portierbar.

Bemerkungen

Moduszeichenfolgen:

`fopen()` in `fopen()` und `freopen()` können einen dieser Werte `freopen()` :

- `"r"` : Öffnet die Datei im Nur-Lese-Modus, wobei der Cursor auf den Anfang der Datei gesetzt ist.
- `"r+"` : Öffnet die Datei im Lese- und Schreibmodus, wobei der Cursor auf den Anfang der Datei gesetzt ist.
- `"w"` : Öffnen oder erstellen Sie die Datei im schreibgeschützten Modus, wobei der Inhalt auf 0 Byte gekürzt wird. Der Cursor steht auf den Anfang der Datei.
- `"w+"` : Öffnen oder erstellen Sie die Datei im Lese- / Schreibmodus, wobei der Inhalt auf 0 Byte gekürzt wird. Der Cursor steht auf den Anfang der Datei.
- `"a"` : Öffnen oder erstellen Sie die Datei im schreibgeschützten Modus, wobei sich der Cursor am Ende der Datei befindet.
- `"a+"` : Öffnen oder erstellen Sie die Datei im Lese- / Schreibmodus, wobei der Lese-Cursor am Anfang der Datei steht. Die Ausgabe wird jedoch *immer* an das Ende der Datei angehängt.

Für jeden dieser Dateimodi kann nach dem Anfangsbuchstaben ein `b` hinzugefügt werden (z. B. `"rb"` oder `"a+b"` oder `"ab+"`). Das `b` bedeutet, dass die Datei auf den Systemen, in denen ein Unterschied besteht, anstelle von Textdateien als Binärdateien behandelt werden sollte. Auf Unix-ähnlichen Systemen macht es keinen Unterschied. es ist wichtig auf Windows-Systemen. (Außerdem erlaubt Windows `fopen` ein explizites `t` anstelle von `b` , um "Textdatei" anzugeben - und zahlreiche andere plattformspezifische Optionen.)

C11

- `"wx"` : Erstellen Sie eine Textdatei im schreibgeschützten Modus. *Die Datei existiert möglicherweise nicht* .
- `"wbx"` : `"wbx"` eine Binärdatei im schreibgeschützten Modus. *Die Datei existiert möglicherweise nicht* .

Das `x` , falls vorhanden, das letzte Zeichen in der Moduszeichenfolge sein.

Examples

Öffnen Sie und schreiben Sie in die Datei

```
#include <stdio.h>   /* for perror(), fopen(), fputs() and fclose() */
#include <stdlib.h>  /* for the EXIT_* macros */

int main(int argc, char **argv)
{
    int e = EXIT_SUCCESS;

    /* Get path from argument to main else default to output.txt */
    char *path = (argc > 1) ? argv[1] : "output.txt";

    /* Open file for writing and obtain file pointer */
    FILE *file = fopen(path, "w");

    /* Print error message and exit if fopen() failed */
```

```
    if (!file)
    {
        perror(path);
        return EXIT_FAILURE;
    }

    /* Writes text to file. Unlike puts(), fputs() does not add a new-line. */
    if (fputs("Output in file.\n", file) == EOF)
    {
        perror(path);
        e = EXIT_FAILURE;
    }

    /* Close file */
    if (fclose(file))
    {
        perror(path);
        return EXIT_FAILURE;
    }
    return e;
}
```

Dieses Programm öffnet die Datei mit dem Namen, der im Argument angegeben ist, in main, wobei `output.txt` wenn kein Argument angegeben wird. Wenn bereits eine Datei mit demselben Namen vorhanden ist, wird deren Inhalt verworfen und die Datei wird als neue leere Datei behandelt. Wenn die Dateien noch nicht vorhanden sind, werden sie vom `fopen()` Aufruf erstellt.

Wenn der Aufruf von `fopen()` aus irgendeinem Grund fehlschlägt, gibt er einen `NULL` Wert zurück und legt den globalen Wert der `errno` Variablen fest. Dies bedeutet , dass das Programm den Rückgabewert nach dem Test kann `fopen()` Aufruf und Nutzung `perror()` , wenn `fopen()` fehlschlägt.

Wenn der Aufruf von `fopen()` erfolgreich ist, wird ein gültiger `FILE` Zeiger zurückgegeben. Dieser Zeiger kann dann verwendet werden, um auf diese Datei zu verweisen, bis `fclose()` für sie aufgerufen wird.

Die Funktion `fputs()` schreibt den angegebenen Text in die geöffnete Datei und ersetzt den vorherigen Inhalt der Datei. Ähnlich wie bei `fopen()` setzt die Funktion `fputs()` auch den `errno` Wert, falls dieser fehlschlägt. In diesem Fall gibt die Funktion `EOF` , um den Fehler anzuzeigen (andernfalls einen nicht negativen Wert).

Die Funktion `fclose()` alle Puffer, schließt die Datei und gibt den Speicher frei, auf den `FILE *` . Der Rückgabewert gibt die Fertigstellung genau wie bei `fputs()` (obwohl bei Erfolg '0' zurückgegeben wird). Im `errno` wird auch der `errno` Wert gesetzt.

fprintf

Sie können `fprintf` für eine Datei genauso wie für eine Konsole mit `printf` . Zum Beispiel, um die Gewinne, Verluste und Bindungen, die Sie schreiben, im Auge zu behalten

```
/* saves wins, losses and, ties */
void savewlt(FILE *fout, int wins, int losses, int ties)
{
```

```
    fprintf(fout, "Wins: %d\nTies: %d\nLosses: %d\n", wins, ties, losses);
}
```

Eine Randnotiz: Einige Systeme (infamous, Windows) verwenden nicht das, was die meisten Programmierer als "normale" Leitungsenden bezeichnen würden. Während UNIX-ähnliche Systeme \ n verwenden, um Zeilen zu beenden, verwendet Windows ein Zeichenpaar: \ r (Wagenrücklauf) und \ n (Zeilenvorschub). Diese Sequenz wird allgemein als CRLF bezeichnet. Bei der Verwendung von C müssen Sie sich jedoch nicht um diese stark plattformabhängigen Details kümmern. Ein AC-Compiler ist erforderlich, um jede Instanz von \ n in die korrekte Endung der Plattformzeile zu konvertieren. Ein Windows-Compiler würde also \ n in \ r \ n konvertieren, aber ein UNIX-Compiler würde es beibehalten.

Prozess ausführen

```
#include <stdio.h>

void print_all(FILE *stream)
{
    int c;
    while ((c = getc(stream)) != EOF)
        putchar(c);
}
int main(void)
{
    FILE *stream;

    /* call netstat command. netstat is available for Windows and Linux */
    if ((stream = popen("netstat", "r")) == NULL)
        return 1;

    print_all(stream);
    pclose(stream);
    return 0;
}
```

Dieses Programm führt einen Prozess (`netstat`) über `popen()` und liest alle Standardausgaben aus dem Prozess und gibt diese an die Standardausgabe zurück.

Hinweis: `popen()` ist nicht in der Standard-C-Bibliothek vorhanden , sondern ist ein Teil von POSIX C).

Zeilen mit getline () aus einer Datei holen

Die POSIX C-Bibliothek definiert die Funktion `getline()` . Diese Funktion weist einen Puffer für den Zeileninhalt zu und gibt die neue Zeile, die Anzahl der Zeichen in der Zeile und die Größe des Puffers zurück.

Beispielprogramm, das jede Zeile aus `example.txt` :

```
#include <stdlib.h>
#include <stdio.h>
```

```
#define FILENAME "example.txt"

int main(void)
{
  /* Open the file for reading */
  char *line_buf = NULL;
  size_t line_buf_size = 0;
  int line_count = 0;
  ssize_t line_size;
  FILE *fp = fopen(FILENAME, "r");
  if (!fp)
  {
    fprintf(stderr, "Error opening file '%s'\n", FILENAME);
    return EXIT_FAILURE;
  }

  /* Get the first line of the file. */
  line_size = getline(&line_buf, &line_buf_size, fp);

  /* Loop through until we are done with the file. */
  while (line_size >= 0)
  {
    /* Increment our line count */
    line_count++;

    /* Show the line details */
    printf("line[%06d]: chars=%06zd, buf size=%06zu, contents: %s", line_count,
        line_size, line_buf_size, line_buf);

    /* Get the next line */
    line_size = getline(&line_buf, &line_buf_size, fp);
  }

  /* Free the allocated line buffer */
  free(line_buf);
  line_buf = NULL;

  /* Close the file now that we are done with it */
  fclose(fp);

  return EXIT_SUCCESS;
}
```

Eingabedatei `example.txt`

```
This is a file
  which has
multiple lines
    with various indentation,
blank lines



a really long line to show that getline() will reallocate the line buffer if the length of a
line is too long to fit in the buffer it has been given,
  and punctuation at the end of the lines.
```

Ausgabe

```
line[000001]: chars=000015, buf size=000016, contents: This is a file
line[000002]: chars=000012, buf size=000016, contents:   which has
line[000003]: chars=000015, buf size=000016, contents: multiple lines
line[000004]: chars=000030, buf size=000032, contents:     with various indentation,
line[000005]: chars=000012, buf size=000032, contents: blank lines
line[000006]: chars=000001, buf size=000032, contents:
line[000007]: chars=000001, buf size=000032, contents:
line[000008]: chars=000001, buf size=000032, contents:
line[000009]: chars=000150, buf size=000160, contents: a really long line to show that
getline() will reallocate the line buffer if the length of a line is too long to fit in the
buffer it has been given,
line[000010]: chars=000042, buf size=000160, contents:  and punctuation at the end of the
lines.
line[000011]: chars=000001, buf size=000160, contents:
```

Im Beispiel wird `getline()` zunächst ohne zugewiesenen Puffer aufgerufen. Während dieses ersten Aufrufs weist `getline()` einen Puffer zu, liest die erste Zeile und platziert den Inhalt der Zeile im neuen Puffer. Bei nachfolgenden Aufrufen aktualisiert `getline()` denselben Puffer und `getline()` den Puffer nur dann neu zu, wenn er nicht mehr groß genug ist, um in die gesamte Zeile zu passen. Der temporäre Puffer wird dann freigegeben, wenn die Datei fertig ist.

Eine weitere Option ist `getdelim()` . Dies ist dasselbe wie `getline()` außer dass Sie das Zeilenendezeichen angeben. Dies ist nur erforderlich, wenn das letzte Zeichen der Zeile für Ihren Dateityp nicht '\ n' ist. `getline()` funktioniert auch mit Windows-Textdateien, da mit der Multibyte-Zeilenendung (`"\r\n")` '\ n'` noch das letzte Zeichen der Zeile ist.

Beispielimplementierung von `getline()`

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <stdint.h>


#if !(defined _POSIX_C_SOURCE)
typedef long int ssize_t;
#endif

/* Only include our version of getline() if the POSIX version isn't available. */

#if !(defined _POSIX_C_SOURCE) || _POSIX_C_SOURCE < 200809L

#if !(defined SSIZE_MAX)
#define SSIZE_MAX (SIZE_MAX >> 1)
#endif

ssize_t getline(char **pline_buf, size_t *pn, FILE *fin)
{
  const size_t INITALLOC = 16;
  const size_t ALLOCSTEP = 16;
  size_t num_read = 0;

  /* First check that none of our input pointers are NULL. */
```

```
if ((NULL == pline_buf) || (NULL == pn) || (NULL == fin))
{
  errno = EINVAL;
  return -1;
}

/* If output buffer is NULL, then allocate a buffer. */
if (NULL == *pline_buf)
{
  *pline_buf = malloc(INITALLOC);
  if (NULL == *pline_buf)
  {
    /* Can't allocate memory. */
    return -1;
  }
  else
  {
    /* Note how big the buffer is at this time. */
    *pn = INITALLOC;
  }
}

/* Step through the file, pulling characters until either a newline or EOF. */

{
  int c;
  while (EOF != (c = getc(fin)))
  {
    /* Note we read a character. */
    num_read++;

    /* Reallocate the buffer if we need more room */
    if (num_read >= *pn)
    {
      size_t n_realloc = *pn + ALLOCSTEP;
      char * tmp = realloc(*pline_buf, n_realloc + 1); /* +1 for the trailing NUL. */
      if (NULL != tmp)
      {
        /* Use the new buffer and note the new buffer size. */
        *pline_buf = tmp;
        *pn = n_realloc;
      }
      else
      {
        /* Exit with error and let the caller free the buffer. */
        return -1;
      }

      /* Test for overflow. */
      if (SSIZE_MAX < *pn)
      {
        errno = ERANGE;
        return -1;
      }
    }

    /* Add the character to the buffer. */
    (*pline_buf)[num_read - 1] = (char) c;

    /* Break from the loop if we hit the ending character. */
    if (c == '\n')
```

```
      {
        break;
      }
    }

    /* Note if we hit EOF. */
    if (EOF == c)
    {
      errno = 0;
      return -1;
    }
  }

  /* Terminate the string by suffixing NUL. */
  (*pline_buf)[num_read] = '\0';

  return (ssize_t) num_read;
}

#endif
```

Öffnen und schreiben Sie in eine Binärdatei

```
#include <stdlib.h>
#include <stdio.h>


int main(void)
{
   result = EXIT_SUCCESS;

   char file_name[] = "outbut.bin";
   char str[] = "This is a binary file example";
   FILE * fp = fopen(file_name, "wb");

   if (fp == NULL)  /* If an error occurs during the file creation */
   {
     result = EXIT_FAILURE;
     fprintf(stderr, "fopen() failed for '%s'\n", file_name);
   }
   else
   {
     size_t element_size = sizeof *str;
     size_t elements_to_write = sizeof str;

     /* Writes str (_including_ the NUL-terminator) to the binary file. */
     size_t elements_written = fwrite(str, element_size, elements_to_write, fp);
     if (elements_written != elements_to_write)
     {
       result = EXIT_FAILURE;
       /* This works for >=c99 only, else the z length modifier is unknown. */
       fprintf(stderr, "fwrite() failed: wrote only %zu out of %zu elements.\n",
         elements_written, elements_to_write);
       /* Use this for <c99: *
       fprintf(stderr, "fwrite() failed: wrote only %lu out of %lu elements.\n",
         (unsigned long) elements_written, (unsigned long) elements_to_write);
        */
     }
```

```
        fclose(fp);
    }

    return result;
}
```

Dieses Programm erstellt und schreibt Text in binärer Form über die Funktion `fwrite` in die Datei `output.bin` .

Wenn bereits eine Datei mit demselben Namen vorhanden ist, wird deren Inhalt verworfen und die Datei wird als neue leere Datei behandelt.

Ein binärer Stream ist eine geordnete Folge von Zeichen, die interne Daten transparent aufzeichnen kann. In diesem Modus werden Bytes ohne Interpretation zwischen dem Programm und der Datei geschrieben.

Um Ganzzahlen portabel schreiben zu können, muss bekannt sein, ob das Dateiformat sie im Big- oder Little-Endian-Format und in der Größe (normalerweise 16, 32 oder 64 Bit) erwartet. Bitverschiebung und Maskierung können dann zum Schreiben der Bytes in der richtigen Reihenfolge verwendet werden. Für Ganzzahlen in C ist nicht garantiert, dass sie eine Zweierkomplementdarstellung haben (obwohl dies bei fast allen Implementierungen der Fall ist). Zum Glück einer Umwandlung in unsigned garantiert Zweier - Komplement zu verwenden. Der Code zum Schreiben einer vorzeichenbehafteten Ganzzahl in eine Binärdatei ist daher etwas überraschend.

```
/* write a 16-bit little endian integer */
int fput16le(int x, FILE *fp)
{
    unsigned int rep = x;
    int e1, e2;

    e1 = fputc(rep & 0xFF, fp);
    e2 = fputc((rep >> 8) & 0xFF, fp);

    if(e1 == EOF || e2 == EOF)
        return EOF;
    return 0;
}
```

Die anderen Funktionen folgen dem gleichen Muster mit geringfügigen Änderungen für Größe und Byte-Reihenfolge.

fscanf ()

Nehmen wir an, wir haben eine Textdatei und möchten alle Wörter in dieser Datei lesen, um einige Anforderungen zu erfüllen.

Datei.txt :

```
This is just
a test file
to be used by fscanf()
```

Dies ist die Hauptfunktion:

```
#include <stdlib.h>
#include <stdio.h>

void printAllWords(FILE *);

int main(void)
{
  FILE *fp;

  if ((fp = fopen("file.txt", "r")) == NULL) {
      perror("Error opening file");
      exit(EXIT_FAILURE);
  }

  printAllWords(fp);

  fclose(fp);

  return EXIT_SUCCESS;
}

void printAllWords(FILE * fp)
{
    char tmp[20];
    int i = 1;

    while (fscanf(fp, "%19s", tmp) != EOF) {
        printf("Word %d: %s\n", i, tmp);
        i++;
    }
}
```

Die Ausgabe wird sein:

```
Word 1: This
Word 2: is
Word 3: just
Word 4: a
Word 5: test
Word 6: file
Word 7: to
Word 8: be
Word 9: used
Word 10: by
Word 11: fscanf()
```

Zeilen aus einer Datei lesen

Der Header `stdio.h` definiert die Funktion `fgets()` . Diese Funktion liest eine Zeile aus einem Stream und speichert sie in einer angegebenen Zeichenfolge. Die Funktion beendet das Lesen von Text aus dem Stream, wenn entweder `n - 1` Zeichen gelesen werden, das Zeilenvorschubzeichen (`'\n'`) gelesen wird oder das Dateiende (EOF) erreicht wird.

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>

#define MAX_LINE_LENGTH 80

int main(int argc, char **argv)
{
    char *path;
    char line[MAX_LINE_LENGTH] = {0};
    unsigned int line_count = 0;

    if (argc < 1)
        return EXIT_FAILURE;
    path = argv[1];

    /* Open file */
    FILE *file = fopen(path, "r");

    if (!file)
    {
        perror(path);
        return EXIT_FAILURE;
    }

    /* Get each line until there are none left */
    while (fgets(line, MAX_LINE_LENGTH, file))
    {
        /* Print each line */
        printf("line[%06d]: %s", ++line_count, line);

        /* Add a trailing newline to lines that don't already have one */
        if (line[strlen(line) - 1] != '\n')
            printf("\n");
    }

    /* Close file */
    if (fclose(file))
    {
        return EXIT_FAILURE;
        perror(path);
    }
}
```

Aufruf des Programms mit einem Argument, bei dem es sich um einen Pfad zu einer Datei handelt, die den folgenden Text enthält:

```
This is a file
  which has
multiple lines
    with various indentation,
blank lines



a really long line to show that the line will be counted as two lines if the length of a line
is too long to fit in the buffer it has been given,
  and punctuation at the end of the lines.
```

Ergibt folgende Ausgabe:

```
line[000001]: This is a file
line[000002]:   which has
line[000003]: multiple lines
line[000004]:     with various indentation,
line[000005]: blank lines
line[000006]:
line[000007]:
line[000008]:
line[000009]: a really long line to show that the line will be counted as two lines if the le
line[000010]: ngth of a line is too long to fit in the buffer it has been given,
line[000011]:  and punctuation at the end of the lines.
line[000012]:
```

Dieses sehr einfache Beispiel erlaubt eine feste maximale Zeilenlänge, so dass längere Zeilen effektiv als zwei Zeilen gezählt werden. Die Funktion `fgets()` setzt `fgets()` , dass der aufrufende Code den Speicher bereitstellt, der als Ziel für die gelesene Zeile verwendet werden soll.

POSIX stellt die Funktion `getline()` Verfügung, die stattdessen intern Speicher `getline()` , um den Puffer nach Bedarf für eine Zeile beliebiger Länge zu vergrößern (sofern ausreichend Speicher vorhanden ist).

Dateien und E / A-Streams online lesen: https://riptutorial.com/de/c/topic/507/dateien-und-e---a-streams

Kapitel 14: Datentypen

Bemerkungen

- Während `char` 1 Byte lang sein muss, muss 1 Byte **nicht** 8 Bit sein (oft auch als *Oktett bezeichnet*), obwohl die meisten modernen Computerplattformen es als 8 Bit definieren. Die Implementierung der Anzahl von Bits pro `char` wird durch die vorgesehenen `CHAR_BIT` Makro, definiert in `<limits.h>` . POSIX erfordert, dass 1 Byte aus 8 Bit besteht.
- Typen mit fester Breite sollten nur spärlich verwendet werden. Die integrierten Typen von C sind so konzipiert, dass sie in jeder Architektur selbstverständlich sind. Die Typen mit fester Breite sollten nur verwendet werden, wenn Sie explizit eine ganzzahlige Größe benötigen (zum Beispiel für Netzwerke).

Examples

Integer-Typen und Konstanten

Signed Integer kann dieser Typen sein (der `int` nach `short` oder `long` ist optional):

```
signed char c = 127; /* required to be 1 byte, see remarks for further information. */
signed short int si = 32767; /* required to be at least 16 bits. */
signed int i = 32767; /* required to be at least 16 bits */
signed long int li = 2147483647; /* required to be at least 32 bits. */
```

C99

```
signed long long int li = 2147483647; /* required to be at least 64 bits */
```

Jeder dieser vorzeichenbehafteten Integer-Typen hat eine vorzeichenlose Version.

```
unsigned int i = 65535;
unsigned short = 2767;
unsigned char = 255;
```

Bei allen Typen außer `char` die `signed` Version angenommen, wenn der `signed` oder `unsigned` Teil ausgelassen wird. Der Typ `char` stellt einen dritten Zeichentyp dar, der sich von `signed char` und `unsigned char` und die Signiertheit (oder nicht) hängt von der Plattform ab.

Verschiedene Typen von Integer-Konstanten (im C-Jargon als *Literale* bezeichnet) können auf der Grundlage ihres Präfixes oder Suffixes mit unterschiedlichen Basen und unterschiedlicher Breite geschrieben werden.

```
/* the following variables are initialized to the same value: */
int d = 42;   /* decimal constant (base10) */
int o = 052;  /* octal constant (base8) */
int x = 0xaf; /* hexadecimal constants (base16) */
int X = 0XAf; /* (letters 'a' through 'f' (case insensitive) represent 10 through 15) */
```

Dezimalkonstanten sind immer `signed` . Hexadezimale Konstanten beginnen mit `0x` oder `0X` und `0X` beginnen nur mit einer `0` . Die beiden letzteren sind `signed` oder `unsigned` abhängig davon, ob der Wert in den signierten Typ passt oder nicht.

```
/* suffixes to describe width and signedness : */
long int i = 0x32; /* no suffix represent int, or long int */
unsigned int ui = 65535u; /* u or U represent unsigned int, or long int */
long int li = 65536l; /* l or L represent long int */
```

Ohne Suffix hat die Konstante den ersten Typ, der zu ihrem Wert passt, d. `INT_MAX` Eine Dezimalkonstante, die größer ist als `INT_MAX` wenn möglich vom Typ `long` oder ansonsten `long long` .

Die Header-Datei `<limits.h>` beschreibt die Grenzen von Ganzzahlen wie folgt. Ihre implementierungsdefinierten Werte müssen gleich oder größer sein als die unten gezeigten, mit demselben Vorzeichen.

Makro	Art	Wert
`CHAR_BIT`	kleinstes Objekt, das kein Bitfeld (Byte) ist	8
`SCHAR_MIN`	`signed char`	-127 / - (2^7 - 1)
`SCHAR_MAX`	`signed char`	+127 / 2^7 - 1
`UCHAR_MAX`	`unsigned char`	255/2^8 - 1
`CHAR_MIN`	`char`	siehe unten
`CHAR_MAX`	`char`	siehe unten
`SHRT_MIN`	`short int`	-32767 / - (2^{15} - 1)
`SHRT_MAX`	`short int`	+32767 / 2^{15} - 1
`USHRT_MAX`	`unsigned short int`	65535/2^{16} - 1
`INT_MIN`	`int`	-32767 / - (2^{15} - 1)
`INT_MAX`	`int`	+32767 / 2^{15} - 1
`UINT_MAX`	`unsigned int`	65535/2^{16} - 1
`LONG_MIN`	`long int`	-2147483647 / - (2^{31} - 1)
`LONG_MAX`	`long int`	+2147483647 / 2^{31} - 1
`ULONG_MAX`	`unsigned long int`	4294967295/2^{32} - 1

C99

Makro	Art	Wert
LLONG_MIN	long long int	-9223372036854775807 / - (2 ^{63} - 1)
LLONG_MAX	long long int	+9223372036854775807 / 2 ^{63} - 1
ULLONG_MAX	unsigned long long int	18446744073709551615/2 ^{64} - 1

Wenn der Wert von einem Objekt vom Typ `char` Zeichen erstreckt , wenn in einem Ausdruck verwendet, der Wert von `CHAR_MIN` soll das gleiche sein wie die der `SCHAR_MIN` und der Wert von `CHAR_MAX` ist die gleich wie das sein `SCHAR_MAX` . Wenn der Wert eines Objekt vom Typ `char` nicht Vorzeichen erweitern , wenn in einem Ausdruck verwendet, der Wert von `CHAR_MIN` sein soll 0 und der Wert von `CHAR_MAX` ist die gleich wie das sein `UCHAR_MAX` .

C99

Der C99-Standard fügte einen neuen Header `<stdint.h>` , der Definitionen für Ganzzahlen mit fester Breite enthält. Eine ausführlichere Erklärung finden Sie im Beispiel für die Ganzzahl mit fester Breite.

String Literals

Ein String-Literal in C ist eine Folge von Zeichen, die durch eine Literal-Null abgeschlossen wird.

```
char* str = "hello, world"; /* string literal */

/* string literals can be used to initialize arrays */
char a1[] = "abc"; /* a1 is char[4] holding {'a','b','c','\0'} */
char a2[4] = "abc"; /* same as a1 */
char a3[3] = "abc"; /* a1 is char[3] holding {'a','b','c'}, missing the '\0' */
```

String-Literale sind **nicht modifizierbar** (und können tatsächlich in einen Nur-Lese-Speicher gestellt werden, z. B. .rodata). Der Versuch, ihre Werte zu ändern, führt zu undefiniertem Verhalten.

```
char* s = "foobar";
s[0] = 'F'; /* undefined behaviour */

/* it's good practice to denote string literals as such, by using `const` */
char const* s1 = "foobar";
s1[0] = 'F'; /* compiler error! */
```

Mehrere String-Literale werden zur Kompilierzeit verkettet, sodass Sie Konstrukte wie diese schreiben können.

C99

```
/* only two narrow or two wide string literals may be concatenated */
char* s = "Hello, " "World";
```

C99

```
/* since C99, more than two can be concatenated */
/* concatenation is implementation defined */
char* s1 = "Hello" ", " "World";

/* common usages are concatenations of format strings */
char* fmt = "%" PRId16; /* PRId16 macro since C99 */
```

Zeichenkettenliterale unterstützen wie Zeichenkonstanten verschiedene Zeichensätze.

```
/* normal string literal, of type char[] */
char* s1 = "abc";

/* wide character string literal, of type wchar_t[] */
wchar_t* s2 = L"abc";
```

C11

```
/* UTF-8 string literal, of type char[] */
char* s3 = u8"abc";

/* 16-bit wide string literal, of type char16_t[] */
char16_t* s4 = u"abc";

/* 32-bit wide string literal, of type char32_t[] */
char32_t* s5 = U"abc";
```

Integer-Typen mit fester Breite (seit C99)

C99

Der Header `<stdint.h>` enthält mehrere Integer-Typdefinitionen mit fester Breite. Diese Typen sind *optional* und werden nur bereitgestellt, wenn die Plattform über einen ganzzahligen Typ mit der entsprechenden Breite verfügt und der entsprechende vorzeichenbehaftete Typ eine Zweierkomplement-Darstellung mit negativen Werten hat.

In den Anmerkungen finden Sie Verwendungshinweise für Typen mit fester Breite.

```
/* commonly used types include */
uint32_t u32 = 32; /* exactly 32-bits wide */

uint8_t u8 = 255;  /* exactly 8-bits wide */

int64_t i64 = -65  /* exactly 64 bit in two's complement representation */
```

Fließkommakonstanten

Die C-Sprache hat drei obligatorische reelle Fließkommatypen: `float` , `double` und `long double` .

```
float f = 0.314f;         /* suffix f or F denotes type float */
double d = 0.314;         /* no suffix denotes double */
long double ld = 0.314l; /* suffix l or L denotes long double */

/* the different parts of a floating point definition are optional */
```

```
double x = 1.; /* valid, fractional part is optional */
double y = .1; /* valid, whole-number part is optional */

/* they can also defined in scientific notation */
double sd = 1.2e3; /* decimal fraction 1.2 is scaled by 10^3, that is 1200.0 */
```

Der Header `<float.h>` definiert verschiedene Grenzwerte für Gleitkommaoperationen.

Gleitkomma-Arithmetik ist implementierungsdefiniert. Die meisten modernen Plattformen (Arm, x86, x86_64, MIPS) verwenden jedoch Fließkommaoperationen nach IEEE 754 .

C verfügt außerdem über drei optionale komplexe Fließkommatypen, die von den oben genannten abgeleitet werden.

Erklärungen interpretieren

Eine charakteristische syntaktische Besonderheit von C besteht darin, dass Deklarationen die Verwendung des deklarierten Objekts wie in einem normalen Ausdruck widerspiegeln.

Die folgenden Operatoren mit identischer Priorität und Assoziativität werden in Deklaratoren wiederverwendet, nämlich:

- der unäre `*` "Dereferenzierungs" -Operator, der einen Zeiger bezeichnet;
- der binäre `[]` "Array Subscription" -Operator, der ein Array bezeichnet;
- der (1 + n) -ary `()` "Funktionsaufruf" -Operator, der eine Funktion bezeichnet;
- die `()` Gruppierung Klammern, die den Vorrang und die Assoziativität der übrigen aufgelisteten Operatoren überschreiben.

Die obigen drei Operatoren haben die folgende Priorität und Assoziativität:

Operator	Relativer Vorrang	Assoziativität
`[]` (Array-Abonnement)	1	Links nach rechts
`()` (Funktionsaufruf)	1	Links nach rechts
`*` (Dereferenzierung)	2	Rechts nach links

Bei der Interpretation von Deklarationen muss man vom Bezeichner nach außen gehen und die benachbarten Operatoren in der richtigen Reihenfolge gemäß der obigen Tabelle anwenden. Jede Anwendung eines Operators kann durch die folgenden englischen Wörter ersetzt werden:

Ausdruck	Deutung
`thing[X]`	ein Array von Größe `X` von ...
`thing(t1, t2, t3)`	eine Funktion, die `t1` , `t2` , `t3` und zurückgibt ...
`*thing`	ein Zeiger auf ...

Daraus folgt, dass der Beginn der englischen Interpretation immer mit dem Bezeichner beginnt und mit dem Typ endet, der auf der linken Seite der Deklaration steht.

Beispiele

```
char *names[20];
```

`[]` hat Vorrang vor `*` , die Interpretation lautet also: `names` ist ein Array der Größe 20 eines Zeigers auf `char` .

```
char (*place)[10];
```

Im Falle der Verwendung von Klammern zum Überschreiben der Rangfolge wird das `*` zuerst angewendet: `place` ist ein Zeiger auf ein Array der Größe 10 von `char` .

```
int fn(long, short);
```

Es gibt keinen Vorrang, über den Sie sich Gedanken machen müssen: `fn` ist eine Funktion, die `long` , `short` und `int` .

```
int *fn(void);
```

`()` Wird zuerst angewendet: `fn` ist eine Funktion, die `void` und einen Zeiger auf `int` .

```
int (*fp)(void);
```

Überschreiben der Priorität von `()` : `fp` ist ein Zeiger auf eine Funktion, die `void` und `int` .

```
int arr[5][8];
```

Mehrdimensionale Arrays bilden keine Ausnahme von der Regel. Die Operatoren `[]` werden in der Reihenfolge von links nach rechts entsprechend der Assoziativität in der Tabelle angewendet: `arr` ist ein Array der Größe 5 eines Arrays der Größe 8 von `int` .

```
int **ptr;
```

Die beiden Dereferenzierungsoperatoren haben gleiche Priorität, sodass die Assoziativität wirksam wird. Die Operatoren werden in einer Reihenfolge von rechts nach links angewendet: `ptr` ist ein Zeiger auf einen Zeiger auf ein `int` .

Mehrere Erklärungen

Das Komma kann als Trennzeichen verwendet werden (* nicht * wie der Komma-Operator), um mehrere Deklarationen innerhalb einer einzelnen Anweisung zu begrenzen. Die folgende Anweisung enthält fünf Deklarationen:

```
int fn(void), *ptr, (*fp)(int), arr[10][20], num;
```

Die deklarierten Objekte im obigen Beispiel sind:

- `fn` : eine Funktion, die `void` und `int` ;
- `ptr` : ein Zeiger auf ein `int` ;
- `fp` : ein Zeiger auf eine Funktion, die `int` und `int` ;
- `arr` : ein Array der Größe 10 eines Arrays der Größe 20 von `int` ;
- `num` : `int` .

Alternative Interpretation

Da Deklarationen gespiegelt werden, kann eine Deklaration auch in Bezug auf die Operatoren, die auf das Objekt angewendet werden könnten, und den endgültigen resultierenden Typ dieses Ausdrucks interpretiert werden. Der Typ, der auf der linken Seite steht, ist das Endergebnis, das sich nach Anwendung aller Operatoren ergibt.

```
/*
 * Subscripting "arr" and dereferencing it yields a "char" result.
 * Particularly: *arr[5] is of type "char".
 */
char *arr[20];

/*
 * Calling "fn" yields an "int" result.
 * Particularly: fn('b') is of type "int".
 */
int fn(char);

/*
 * Dereferencing "fp" and then calling it yields an "int" result.
 * Particularly: (*fp)() is of type "int".
 */
int (*fp)(void);

/*
 * Subscripting "strings" twice and dereferencing it yields a "char" result.
 * Particularly: *strings[5][15] is of type "char"
 */
char *strings[10][20];
```

Datentypen online lesen: https://riptutorial.com/de/c/topic/309/datentypen

Kapitel 15: Deklaration vs. Definition

Bemerkungen

Quelle: Was ist der Unterschied zwischen einer Definition und einer Deklaration?

Quelle (für schwache und starke Symbole): https://www.amazon.com/Computer-Systems-Programmierer-Perspektive-2nd/dp/0136108040/

Examples

Erklärung und Definition verstehen

Eine Deklaration führt einen Bezeichner ein und beschreibt seinen Typ, sei es Typ, Objekt oder Funktion. Eine Deklaration ist das, was der Compiler benötigt, um Verweise auf diesen Bezeichner zu akzeptieren. Dies sind Deklarationen:

```
extern int bar;
extern int g(int, int);
double f(int, double); /* extern can be omitted for function declarations */
double h1();           /* declaration without prototype */
double h2();           /* ditto                         */
```

Eine Definition instantiiert / implementiert diese Kennung. Es ist das, was der Linker benötigt, um Verweise auf diese Entitäten zu verlinken. Dies sind Definitionen, die den obigen Deklarationen entsprechen:

```
int bar;
int g(int lhs, int rhs) {return lhs*rhs;}
double f(int i, double d) {return i+d;}
double h1(int a, int b) {return -1.5;}
double h2() {}  /* prototype is implied in definition, same as double h2(void) */
```

Anstelle einer Deklaration kann eine Definition verwendet werden.

Es muss jedoch genau einmal definiert werden. Wenn Sie vergessen, etwas zu definieren, das irgendwo deklariert und referenziert wurde, weiß der Linker nicht, worauf er Bezug nehmen soll, und er beschwert sich über fehlende Symbole. Wenn Sie etwas mehr als einmal definieren, kann der Linker nicht wissen, auf welche Definitionen Verweise verweist und sich über duplizierte Symbole beschwert.

Ausnahme:

```
extern int i = 0;  /* defines i */
extern int j;  /* declares j */
```

Diese Ausnahme kann mit Konzepten von "Starken Symbolen vs. Schwachen Symbolen" (aus

Linkersicht) erklärt werden. Weitere Informationen finden Sie hier (Bild 22).

```
/* All are definitions. */
struct S { int a; int b; };             /* defines S */
struct X {                              /* defines X */
    int x;                              /* defines non-static data member x */
};
struct X anX;                                  /* defines anX */
```

Deklaration vs. Definition online lesen: https://riptutorial.com/de/c/topic/3104/deklaration-vs--definition

Kapitel 16: Einschränkungen

Bemerkungen

Einschränkungen sind ein Begriff, der in allen vorhandenen C-Spezifikationen verwendet wird (kürzlich ISO-IEC 9899-2011). Sie sind einer der drei Teile der in Abschnitt 6 der Norm beschriebenen Sprache (neben Syntax und Semantik).

ISO-IEC 9899-2011 definiert eine Einschränkung als:

> Einschränkung, entweder syntaktisch oder semantisch, durch die die Exposition von Sprachelementen interpretiert werden soll

(Bitte beachten Sie auch, dass in Bezug auf den C-Standard eine "Laufzeitbeschränkung" keine Art von Einschränkung ist und weitgehend andere Regeln hat.)

Mit anderen Worten beschreibt eine Einschränkung eine Regel der Sprache, die ein sonst syntaktisch gültiges Programm illegal machen würde. In dieser Hinsicht ähneln Einschränkungen gewissermaßen undefiniertes Verhalten, jedes Programm, das ihnen nicht folgt, ist nicht in Bezug auf die C-Sprache definiert.

Einschränkungen dagegen haben einen sehr signifikanten Unterschied zu undefinierten Verhalten. Es ist nämlich eine Implementierung erforderlich, um während der Übersetzungsphase (Teil der Kompilierung) eine Diagnosemeldung bereitzustellen, wenn eine Einschränkung verletzt wird. Diese Nachricht kann eine Warnung sein oder die Kompilierung stoppen.

Examples

Doppelte Variablennamen im selben Bereich

Ein Beispiel für eine Einschränkung, wie sie im C-Standard zum Ausdruck kommt, besteht darin, dass zwei Variablen mit demselben Namen in einem Gültigkeitsbereich [1)] deklariert sind, zum Beispiel:

```
void foo(int bar)
{
    int var;
    double var;
}
```

Dieser Code verstößt gegen die Einschränkung und muss zur Kompilierzeit eine Diagnosemeldung erzeugen. Dies ist sehr nützlich im Vergleich zu undefiniertem Verhalten, da der Entwickler vor der Ausführung des Programms über das Problem informiert wird und möglicherweise irgendetwas tut.

Einschränkungen sind daher tendenziell Fehler, die zum Zeitpunkt des Kompilierens leicht erkannt werden können. Probleme, die zu undefiniertem Verhalten führen, aber zum Zeitpunkt des

Kompilierens schwer oder nicht zu erkennen wären, sind daher keine Einschränkungen.

---

1) genaue Formulierung:

C99

Wenn ein Bezeichner keine Verknüpfung hat, darf es nicht mehr als eine Deklaration des Bezeichners (in einem Deklarator oder Typbezeichner) mit demselben Gültigkeitsbereich und demselben Namensraum geben, mit Ausnahme der in 6.7.2.3 angegebenen Tags.

Unäre arithmetische Operatoren

Die unären `+` und `-` Operatoren können nur für arithmetische Typen verwendet werden. Wenn Sie beispielsweise versuchen, sie für eine Struktur zu verwenden, erzeugt das Programm eine Diagnose, z.

```
struct foo
{
    bool bar;
};

void baz(void)
{
   struct foo testStruct;
   -testStruct; /* This breaks the constraint so must produce a diagnostic */
}
```

Einschränkungen online lesen: https://riptutorial.com/de/c/topic/7397/einschrankungen

Kapitel 17: Erklärungen

Bemerkungen

Die Deklaration eines Bezeichners, der sich auf ein Objekt oder eine Funktion bezieht, wird häufig kurz als Deklaration eines Objekts oder einer Funktion bezeichnet.

Examples

Aufruf einer Funktion aus einer anderen C-Datei

foo.h

```
#ifndef FOO_DOT_H    /* This is an "include guard" */
#define FOO_DOT_H    /* prevents the file from being included twice. */
                     /* Including a header file twice causes all kinds */
                     /* of interesting problems.*/

/**
 * This is a function declaration.
 * It tells the compiler that the function exists somewhere.
 */
void foo(int id, char *name);

#endif /* FOO_DOT_H */
```

foo.c

```
#include "foo.h"    /* Always include the header file that declares something
                     * in the C file that defines it. This makes sure that the
                     * declaration and definition are always in-sync.  Put this
                     * header first in foo.c to ensure the header is self-contained.
                     */
#include <stdio.h>

/**
 * This is the function definition.
 * It is the actual body of the function which was declared elsewhere.
 */
void foo(int id, char *name)
{
    fprintf(stderr, "foo(%d, \"%s\");\n", id, name);
    /* This will print how foo was called to stderr - standard error.
     * e.g., foo(42, "Hi!") will print `foo(42, "Hi!")`
     */
}
```

Haupt c

```
#include "foo.h"
```

```
int main(void)
{
    foo(42, "bar");
    return 0;
}
```

Kompilieren und Verknüpfen

Zuerst *kompilieren* wir `foo.c` und `main.c` in *Objektdateien* . Hier verwenden wir den `gcc` Compiler. Ihr Compiler hat möglicherweise einen anderen Namen und benötigt andere Optionen.

```
$ gcc -Wall -c foo.c
$ gcc -Wall -c main.c
```

Jetzt verknüpfen wir sie miteinander, um unsere endgültige ausführbare Datei zu erstellen:

```
$ gcc -o testprogram foo.o main.o
```

Verwendung einer globalen Variablen

Von der Verwendung globaler Variablen wird generell abgeraten. Dies macht Ihr Programm schwieriger zu verstehen und schwieriger zu debuggen. Aber manchmal ist die Verwendung einer globalen Variablen akzeptabel.

global.h

```
#ifndef GLOBAL_DOT_H    /* This is an "include guard" */
#define GLOBAL_DOT_H

/**
 * This tells the compiler that g_myglobal exists somewhere.
 * Without "extern", this would create a new variable named
 * g_myglobal in _every file_ that included it. Don't miss this!
 */
extern int g_myglobal; /* _Declare_ g_myglobal, that is promise it will be _defined_ by
                        * some module. */

#endif /* GLOBAL_DOT_H */
```

global.c

```
#include "global.h" /* Always include the header file that declares something
                     * in the C file that defines it. This makes sure that the
                     * declaration and definition are always in-sync.
                     */

int g_myglobal;     /* _Define_ my_global. As living in global scope it gets initialised to 0
                     * on program start-up. */
```

Haupt c

```
#include "global.h"
```

```
int main(void)
{
    g_myglobal = 42;
    return 0;
}
```

Siehe auch Wie verwende ich `extern` , um Variablen zwischen Quelldateien zu teilen?

Globale Konstanten verwenden

Header können verwendet werden, um global verwendete schreibgeschützte Ressourcen, wie zum Beispiel Stringtabellen, zu deklarieren.

Deklarieren Sie diese in einem separaten Header, der in jeder Datei (" *Translation Unit* ") enthalten ist, die sie verwenden möchte. Es ist praktisch, den gleichen Header zu verwenden, um eine zugehörige Aufzählung zu definieren, um alle String-Ressourcen zu identifizieren:

resources.h:

```
#ifndef RESOURCES_H
#define RESOURCES_H

typedef enum { /* Define a type describing the possible valid resource IDs. */
  RESOURCE_UNDEFINED = -1, /* To be used to initialise any EnumResourceID typed variable to be

                              marked as "not in use", "not in list", "undefined", wtf.
                              Will say un-initialised on application level, not on language
level. Initialised uninitialised, so to say ;-)
                              Its like NULL for pointers ;-)*/
  RESOURCE_UNKNOWN = 0,    /* To be used if the application uses some resource ID,
                              for which we do not have a table entry defined, a fall back in
                              case we _need_ to display something, but do not find anything
                              appropriate. */

  /* The following identify the resources we have defined: */
  RESOURCE_OK,
  RESOURCE_CANCEL,
  RESOURCE_ABORT,
  /* Insert more here. */

  RESOURCE_MAX /* The maximum number of resources defined. */
} EnumResourceID;


extern const char * const resources[RESOURCE_MAX]; /* Declare, promise to anybody who includes

                                      this, that at linkage-time this symbol will be around.
                                      The 1st const guarantees the strings will not change,
                                      the 2nd const guarantees the string-table entries
                                      will never suddenly point somewhere else as set during
                                      initialisation. */
#endif
```

Um tatsächlich die Ressourcen zu definieren, die eine zugehörige .c-Datei erstellt haben, ist dies eine weitere Übersetzungseinheit, die die tatsächlichen Instanzen der in der zugehörigen Header-

Datei (.h) deklarierten Dateien enthält:

resources.c:

```
#include "resources.h" /* To make sure clashes between declaration and definition are
                          recognised by the compiler include the declaring header into
                          the implementing, defining translation unit (.c file).

/* Define the resources. Keep the promise made in resources.h. */
const char * const resources[RESOURCE_MAX] = {
  "<unknown>",
  "OK",
  "Cancel",
  "Abort"
};
```

Ein Programm, das dies verwendet, könnte folgendermaßen aussehen:

Haupt c:

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h>

#include "resources.h"


int main(void)
{
  EnumResourceID resource_id = RESOURCE_UNDEFINED;

  while ((++resource_id) < RESOURCE_MAX)
  {
    printf("resource ID: %d, resource: '%s'\n", resource_id, resources[resource_id]);
  }

  return EXIT_SUCCESS;
}
```

Kompilieren Sie die drei obigen Dateien mithilfe von GCC und verknüpfen Sie sie, um beispielsweise die Programmdatei `main` zu werden.

```
gcc -Wall -Wextra -pedantic -Wconversion -g  main.c resources.c -o main
```

(Verwenden Sie diese `-Wall -Wextra -pedantic -Wconversion` , um den Compiler wirklich wählerisch zu machen, damit Sie nichts verpassen, bevor Sie den Code in SO veröffentlichen, sagen Sie der Welt, oder lohnt es sich, ihn in der Produktion einzusetzen).

Führen Sie das erstellte Programm aus:

```
$ ./main
```

Und bekomme:

```
resource ID: 0, resource: '<unknown>'
resource ID: 1, resource: 'OK'
resource ID: 2, resource: 'Cancel'
resource ID: 3, resource: 'Abort'
```

Einführung

Beispiele für Deklarationen sind:

```
int a; /* declaring single identifier of type int */
```

Die obige Deklaration deklariert einen einzelnen Bezeichner mit dem Namen `a` der sich auf ein Objekt mit dem Typ `int` bezieht.

```
int a1, b1; /* declaring 2 identifiers of type int */
```

Die zweite Deklaration deklariert 2 Bezeichner mit den Namen `a1` und `b1` die sich auf andere Objekte beziehen, jedoch mit demselben `int` Typ.

Im Grunde funktioniert das folgendermaßen: Zuerst geben Sie einen Typ ein , dann schreiben Sie einen einzelnen oder mehrere durch Kommas getrennte Ausdrücke (`,` **die an dieser Stelle nicht ausgewertet werden) und die sonst als Deklaratoren bezeichnet werden diesem Kontext**). Beim Schreiben solcher Ausdrücke dürfen Sie nur die Operatoren Indirection (`*`), Funktionsaufruf (`( )`) oder Subskription (oder Array-Indizierung - `[ ]`) auf einen Bezeichner anwenden (Sie können auch keine Operatoren verwenden). Der verwendete Bezeichner muss im aktuellen Bereich nicht sichtbar sein. Einige Beispiele:

```
/* 1 */ int /* 2 */ (*z) /* 3 */ , /* 4 */ *x , /* 5 */ **c /* 6 */ ;
```

#	Beschreibung
1	Der Name des ganzzahligen Typs.
2	Nicht ausgewerteter Ausdruck, der die Indirektion auf einen Bezeichner anwendet, `z` .
3	Wir haben ein Komma, das darauf hinweist, dass in derselben Deklaration ein weiterer Ausdruck folgt.
4	Nicht ausgewerteter Ausdruck, der die Indirektion auf einen anderen Bezeichner `x` anwendet.
5	Nicht ausgewerteter Ausdruck, der Indirektion auf den Wert des Ausdrucks `(*c)` anwendet.
6	Ende der Erklärung

Beachten Sie, dass vor dieser Deklaration keine der oben genannten Bezeichner sichtbar waren und die verwendeten Ausdrücke daher nicht gültig waren.

Nach jedem solchen Ausdruck wird der darin verwendete Bezeichner in den aktuellen Bereich eingefügt. (Wenn der Bezeichner ihm eine Verknüpfung zugewiesen hat, kann er auch mit derselben Art von Verknüpfung neu deklariert werden, sodass sich beide Bezeichner auf dasselbe Objekt oder dieselbe Funktion beziehen.)

Zusätzlich kann das Gleichheitszeichen (`=`) zur Initialisierung verwendet werden. Wenn auf einen nicht ausgewerteten Ausdruck (Deklarator) innerhalb der Deklaration `=` gefolgt wird, sagen wir, dass der eingeführte Bezeichner ebenfalls initialisiert wird. Nach dem `=` -Zeichen können wir noch einmal einen Ausdruck setzen, aber dieses Mal wird es ausgewertet und sein Wert wird als Anfang für das deklarierte Objekt verwendet.

Beispiele:

```
int l = 90; /* the same as: */

int l; l = 90; /* if it the declaration of l was in block scope */

int c = 2, b[c]; /* ok, equivalent to: */

int c = 2; int b[c];
```

Später in Ihrem Code können Sie genau den gleichen Ausdruck aus dem Deklarationsteil des neu eingeführten Bezeichners schreiben. Dadurch erhalten Sie ein Objekt des Typs, der zu Beginn der Deklaration angegeben wurde. Dabei wird davon ausgegangen, dass Sie allen Zugriff auf gültige Werte zugewiesen haben Objekte im Weg. Beispiele:

```
void f()
{
    int b2; /* you should be able to write later in your code b2
            which will directly refer to the integer object
            that b2 identifies */

    b2 = 2; /* assign a value to b2 */

    printf("%d", b2); /*ok - should print 2*/

    int *b3; /* you should be able to write later in your code *b3 */

    b3 = &b2; /* assign valid pointer value to b3 */

    printf("%d", *b3); /* ok - should print 2 */

    int **b4; /* you should be able to write later in your code **b4 */

    b4 = &b3;

    printf("%d", **b4); /* ok - should print 2 */

    void (*p)(); /* you should be able to write later in your code (*p)() */

    p = &f; /* assign a valid pointer value */

    (*p)(); /* ok - calls function f by retrieving the
            pointer value inside p -    p
            and dereferencing it -      *p
```

```
            resulting in a function
            which is then called -      (*p)() -

            it is not *p() because else first the () operator is
            applied to p and then the resulting void object is
            dereferenced which is not what we want here */
}
```

Die Deklaration von `b3` gibt an, dass Sie den `b3` Wert möglicherweise als Mittelwert für den Zugriff auf ein ganzzahliges Objekt verwenden können.

Um Indirection (`*`) auf `b3` anzuwenden, müssen Sie natürlich auch einen korrekten Wert speichern (siehe Zeiger für weitere Informationen). Sie sollten auch einen Wert in ein Objekt speichern , zuerst bevor Sie es abrufen (können Sie mehr über dieses Problem finden Sie hier). Wir haben das alles in den obigen Beispielen gemacht.

```
int a3(); /* you should be able to call a3 */
```

Dies sagt dem Compiler, dass Sie versuchen, `a3` aufzurufen. In diesem Fall bezieht sich `a3` auf die Funktion anstelle eines Objekts. Ein Unterschied zwischen Objekt und Funktion besteht darin, dass Funktionen immer eine Art Verknüpfung aufweisen. Beispiele:

```
void f1()
{
    {
        int f2(); /* 1 refers to some function f2 */
    }

    {
        int f2(); /* refers to the exact same function f2 as (1) */
    }
}
```

In dem obigen Beispiel beziehen sich die 2 Deklarationen auf dieselbe Funktion `f2` , während sie, wenn sie Objekte deklarieren würden, in diesem Zusammenhang (mit zwei verschiedenen Blockbereichen) zwei unterschiedliche Objekte haben würden.

```
int (*a3)(); /* you should be able to apply indirection to `a3` and then call it */
```

Nun scheint es kompliziert zu werden, aber wenn Sie die Vorrangigkeit der Operatoren kennen, haben Sie 0 Probleme beim Lesen der obigen Deklaration. Die Klammern werden benötigt, da der Operator `*` eine geringere Priorität als der `( )` .

Im Falle der Verwendung des Indexoperators ist der resultierende Ausdruck nach der Deklaration nicht wirklich gültig, da der darin verwendete Index (der Wert innerhalb von `[` und `]`) immer 1 über dem maximal zulässigen Wert für dieses Objekt / diese Funktion liegt.

```
int a4[5]; /* here a4 shouldn't be accessed using the index 5 later on */
```

Es sollte jedoch für alle anderen Indizes unter 5 verfügbar sein. Beispiele:

```
a4[0], a4[1]; a4[4];
```

`a4[5]` führt zu UB. Weitere Informationen zu Arrays finden Sie hier .

```
int (*a5)[5](); /* here a4 could be applied indirection
                indexed up to (but not including) 5
                and called */
```

Leider ist die Deklaration von `a5` , obwohl syntaktisch möglich, durch den aktuellen Standard verboten.

Typedef

Typedefs sind Deklarationen, bei denen das Schlüsselwort `typedef` vor und vor dem Typ steht. Z.B:

```
typedef int (*(*t0)())[5];
```

(*Sie können das typedef auch technisch nach dem Typ setzen - wie hier* *`int typedef (*(*t0)())[5];`*

Die obigen Deklarationen deklarieren einen Bezeichner für einen Typedef-Namen. Sie können es später so verwenden:

```
t0 pf;
```

Was hat den gleichen Effekt wie das Schreiben:

```
int (*(*pf)())[5];
```

Wie Sie sehen, "speichert" der Typedef-Name die Deklaration als einen Typ, der später für andere Deklarationen verwendet wird. Auf diese Weise können Sie einige Tastatureingaben speichern. Auch als Deklaration mit `typedef` sind Sie immer noch eine Deklaration. Sie sind nicht nur auf das obige Beispiel beschränkt:

```
t0 (*pf1);
```

Ist das gleiche wie:

```
int (*(**pf1)())[5];
```

Verwenden der Rechts-Links- oder Spiralregel zum Entschlüsseln der C-Deklaration

Die "Rechts-Links" -Regel ist eine völlig reguläre Regel zum Entschlüsseln von C-Deklarationen. Es kann auch nützlich sein, um sie zu erstellen.

Lesen Sie die Symbole, wenn Sie auf sie in der Deklaration stoßen ...

```
*   as "pointer to"          - always on the left side
[]  as "array of"            - always on the right side
()  as "function returning"  - always on the right side
```

So wenden Sie die Regel an

SCHRITT 1

Identifizierer suchen Dies ist dein Ausgangspunkt. Dann sagen Sie sich: "Identifier is". Sie haben mit Ihrer Erklärung begonnen.

SCHRITT 2

Schauen Sie sich die Symbole auf der rechten Seite des Bezeichners an. Wenn Sie beispielsweise `()` dort finden, wissen Sie, dass dies die Deklaration einer Funktion ist. Sie hätten also *"Kennung gibt Funktion zurück"* . Wenn Sie dort ein `[]` haben, würden Sie sagen *"Bezeichner ist Array von"* . Fahren Sie nach rechts fort, bis Ihnen die Symbole ausgehen ODER drücken Sie eine rechte Klammer `)` . (Wenn Sie eine linke Klammer treffen `(` , das ist der Beginn eines `()` Symbol, auch wenn es Zeug ist in den Klammern. Mehr dazu weiter unten.)

SCHRITT 3

Schauen Sie sich die Symbole links neben dem Bezeichner an. Wenn es sich nicht um eines unserer Symbole handelt (sagen Sie etwas wie "int"), sagen Sie es einfach. Andernfalls übersetzen Sie es mit Hilfe der obigen Tabelle ins Englische. Fahren Sie weiter nach links, bis Ihnen die Symbole ausgehen ODER Sie eine linke Klammer `(` .

Wiederholen Sie die Schritte 2 und 3, bis Sie Ihre Deklaration erstellt haben.

Hier sind einige Beispiele:

```
int *p[];
```

Zuerst Identifizierer finden:

```
int *p[];
     ^
```

"p ist"

Bewegen Sie sich jetzt nach rechts, bis keine Symbole oder die rechte Klammer getroffen ist.

```
int *p[];
      ^^
```

"p ist Array von"

Kann sich nicht mehr nach rechts bewegen (außerhalb von Symbolen), also nach links gehen und suchen:

```
int *p[];
    ^
```

"p ist ein Array von Zeiger auf"

Gehe nach links und finde:

```
int *p[];
^^^
```

"p ist ein Array von Zeiger auf int".

(oder *"p ist ein Array, in dem jedes Element vom Typ Zeiger auf int ist"*)

Ein anderes Beispiel:

```
int *(*func())();
```

Identifizierer suchen

```
int *(*func())();
       ^^^^
```

"func is"

Nach rechts bewegen.

```
int *(*func())();
           ^^
```

"Funktion ist Rückkehr"

Ich kann mich wegen der rechten Klammer nicht mehr nach rechts bewegen, also nach links gehen.

```
int *(*func())();
      ^
```

"func ist Funktion, die einen Zeiger auf" zurückgibt.

Ich kann mich wegen der linken Klammer nicht mehr nach links bewegen, also weiter nach rechts.

```
int *(*func())();
              ^^
```

"func ist Funktion, die Zeiger auf Funktion zurückgibt"

Ich kann mich nicht mehr nach rechts bewegen, weil wir keine Symbole mehr haben. Gehen Sie also nach links.

```
int *(*func())();
    ^
```

"func ist Funktion, die den Zeiger auf die Funktion zurückgibt.

Und schließlich weiter nach links, denn rechts ist nichts mehr.

```
int *(*func())();
^^^
```

msgstr "func ist eine Funktion, die einen Zeiger auf eine Funktion zurückgibt und einen Zeiger auf int."

Wie Sie sehen, kann diese Regel sehr nützlich sein. Sie können es auch verwenden, um sich während der Erstellung von Deklarationen zu überprüfen und einen Hinweis darauf zu geben, wo das nächste Symbol platziert werden soll und ob Klammern erforderlich sind.

Einige Deklarationen sehen viel komplizierter aus, als sie aufgrund von Arraygrößen und Argumentlisten in Prototypform sind. Wenn Sie `[3]` , wird dies als *"Array (Größe 3) von ..."* gelesen. Wenn Sie `(char *,int)` , wird dies als * "Funktion gelesen, die erwartet (char *, int) und zurückgibt ..."* .

Hier ist ein Spaß:

```
int (*(*fun_one)(char *,double))[9][20];
```

Ich werde nicht durch die einzelnen Schritte gehen, um diese zu entschlüsseln.

* "fun_one ist ein Zeiger auf eine Funktion, die einen *Zeiger auf das Array (Größe 9) des Arrays (Größe 20) von int* erwartet *.*

Wie Sie sehen, ist es nicht so kompliziert, wenn Sie die Arraygrößen und Argumentlisten loswerden:

```
int (*(*fun_one)())[][];
```

Sie können es auf diese Weise entschlüsseln und später die Array-Größen und Argumentlisten eingeben.

Einige abschließende Worte:

Es ist durchaus möglich, mit dieser Regel illegale Erklärungen abzugeben, daher ist ein gewisses Wissen darüber erforderlich, was in C legal ist. Wenn zum Beispiel das Obige gewesen wäre:

```
int *((*fun_one)())[][];
```

es hätte gelesen *"fun_one ist ein Zeiger auf eine Funktion, die ein Array des Zeigerarrays auf int zurückgibt"* . Da eine Funktion kein Array, sondern nur einen Zeiger auf ein Array zurückgeben kann, ist diese Deklaration ungültig.

Zu den illegalen Kombinationen gehören:

```
[]() - cannot have an array of functions
()() - cannot have a function that returns a function
()[] - cannot have a function that returns an array
```

In allen oben genannten Fällen benötigen Sie eine Reihe von Klammern, um ein linkes `*` -Symbol zwischen diesen Symbolen `()` und `[]` rechts zu binden, damit die Deklaration zulässig ist.

Hier einige weitere Beispiele:

Rechtliches

```
int i;               an int
int *p;              an int pointer (ptr to an int)
int a[];             an array of ints
int f();             a function returning an int
int **pp;            a pointer to an int pointer (ptr to a ptr to an int)
int (*pa)[];         a pointer to an array of ints
int (*pf)();         a pointer to a function returning an int
int *ap[];           an array of int pointers (array of ptrs to ints)
int aa[][];          an array of arrays of ints
int *fp();           a function returning an int pointer
int ***ppp;          a pointer to a pointer to an int pointer
int (**ppa)[];       a pointer to a pointer to an array of ints
int (**ppf)();       a pointer to a pointer to a function returning an int
int *(*pap)[];       a pointer to an array of int pointers
int (*paa)[][];      a pointer to an array of arrays of ints
int *(*pfp)();       a pointer to a function returning an int pointer
int **app[];         an array of pointers to int pointers
int (*apa[])[];      an array of pointers to arrays of ints
int (*apf[])();      an array of pointers to functions returning an int
int *aap[][];        an array of arrays of int pointers
int aaa[][][];       an array of arrays of arrays of int
int **fpp();         a function returning a pointer to an int pointer
int (*fpa())[];      a function returning a pointer to an array of ints
int (*fpf())();      a function returning a pointer to a function returning an int
```

Illegal

```
int af[]();          an array of functions returning an int
int fa()[];          a function returning an array of ints
int ff()();          a function returning a function returning an int
int (*pfa)()[];      a pointer to a function returning an array of ints
int aaf[][]();       an array of arrays of functions returning an int
int (*paf)[]();      a pointer to a an array of functions returning an int
int (*pff)()();      a pointer to a function returning a function returning an int
int *afp[]();        an array of functions returning int pointers
int afa[]()[];       an array of functions returning an array of ints
int aff[]()();       an array of functions returning functions returning an int
int *fap()[];        a function returning an array of int pointers
```

```
int faa()[][];        a function returning an array of arrays of ints
int faf()[]();        a function returning an array of functions returning an int
int *ffp()();         a function returning a function returning an int pointer
```

Quelle: http://ieng9.ucsd.edu/~cs30x/rt_lt.rule.html

Erklärungen online lesen: https://riptutorial.com/de/c/topic/3729/erklarungen

Kapitel 18: Erstellen Sie Header-Dateien und fügen Sie sie ein

Einführung

In modernen C sind Header-Dateien entscheidende Werkzeuge, die korrekt entworfen und verwendet werden müssen. Sie ermöglichen es dem Compiler, unabhängig kompilierte Teile eines Programms zu überprüfen.

Header deklarieren Typen, Funktionen, Makros usw., die von den Verbrauchern einer Reihe von Einrichtungen benötigt werden. Der gesamte Code, der eine dieser Funktionen verwendet, enthält den Header. Der gesamte Code, der diese Funktionen definiert, enthält den Header. Dadurch kann der Compiler prüfen, ob die Verwendungen und Definitionen übereinstimmen.

Examples

Einführung

Es gibt eine Reihe von Richtlinien, die beim Erstellen und Verwenden von Header-Dateien in einem C-Projekt zu beachten sind:

- Idemopotenz

  Wenn eine Headerdatei mehrmals in einer Übersetzungseinheit (TU) enthalten ist, sollte sie die Builds nicht beschädigen.

- Selbstbeherrschung

  Wenn Sie die in einer Header-Datei deklarierten Einrichtungen benötigen, müssen Sie keine anderen Header explizit angeben.

- Minimalität

  Sie sollten keine Informationen aus einem Header entfernen können, ohne dass Builds fehlschlagen.

- Einschließen, was Sie verwenden (IWYU)

  Für C ++ mehr Bedeutung als für C, aber auch für C wichtig. Wenn der Code in der TU (nennen wir es `code.c`) direkt die Eigenschaften verwendet , die durch einen Header deklariert (nennen wir es `"headerA.h"`), dann `code.c` sollte `#include "headerA.h"` direkt, auch wenn die TU umfasst Ein weiterer Header (nennen Sie es `"headerB.h"`), der momentan `"headerA.h"` .

Gelegentlich gibt es gute Gründe, eine oder mehrere dieser Richtlinien zu brechen. Sie sollten jedoch beide wissen, dass Sie die Regel brechen, und sich der Konsequenzen dessen bewusst

sein, bevor Sie sie brechen.

Idempotenz

Wenn eine bestimmte Header-Datei mehr als einmal in einer Übersetzungseinheit (TU) enthalten ist, sollten keine Kompilierungsprobleme auftreten. Dies wird als "Idempotenz" bezeichnet. Ihre Header sollten idempotent sein. Stellen Sie sich vor, wie schwierig das Leben wäre, wenn Sie dafür sorgen müssten, dass `#include <stdio.h>` nur einmal aufgenommen wurde.

Es gibt zwei Möglichkeiten, um Idempotenz zu erreichen: Header Guards und die Direktive `#pragma once` .

Kopfschutz

Kopfschutzvorrichtungen sind einfach und zuverlässig und entsprechen dem C-Standard. Die ersten Nicht-Kommentarzeilen in einer Header-Datei sollten folgende Form haben:

```
#ifndef UNIQUE_ID_FOR_HEADER
#define UNIQUE_ID_FOR_HEADER
```

Die letzte Nichtkommentarzeile sollte `#endif` , optional mit einem Kommentar:

```
#endif /* UNIQUE_ID_FOR_HEADER */
```

Der gesamte Betriebscode, einschließlich anderer `#include` Direktiven, sollte zwischen diesen Zeilen stehen.

Jeder Name muss eindeutig sein. Häufig wird ein Namensschema wie `HEADER_H_INCLUDED` verwendet. Ein älterer Code verwendet ein Symbol, das als Header Guard definiert ist (z. B. `#ifndef BUFSIZ` in `<stdio.h>`), aber es ist nicht so zuverlässig wie ein eindeutiger Name.

Eine Option wäre die Verwendung eines generierten MD5-Hashes (oder eines anderen) für den Header-Guard-Namen. Sie sollten es vermeiden, die Schemata zu emulieren, die von Systemköpfen verwendet werden, die häufig für die Implementierung reservierte Namen verwenden - Namen, die mit einem Unterstrich beginnen, gefolgt von einem weiteren Unterstrich oder einem Großbuchstaben.

Die `#pragma once` Direktive

Alternativ unterstützen einige Compiler die Direktive `#pragma once` , die dieselbe Wirkung wie die drei Zeilen für Header-Guards hat.

```
#pragma once
```

Zu den Compilern, die `#pragma once` gehören MS Visual Studio und GCC und Clang. Wenn die Portabilität jedoch ein Problem darstellt, sollten Kopfschutzvorrichtungen oder beides verwendet werden. Moderne Compiler (diejenigen, die C89 oder höher unterstützen) müssen unangemessen

ignorierte Pragmas ignorieren ("Ein solches Pragma, das von der Implementierung nicht erkannt wird, wird ignoriert"), aber alte Versionen von GCC waren nicht nachsichtig.

Selbstbeherrschung

Moderne Header sollten in sich geschlossen sein, was bedeutet, dass ein Programm, das die von `header.h` definierten `header.h` verwenden muss, diesen Header (`#include "header.h"`) enthalten kann und sich nicht darum kümmern muss, ob andere Header zuerst `#include "header.h"` müssen.

Empfehlung: Header-Dateien sollten in sich geschlossen sein.

Historische Regeln

Historisch war dies ein leicht kontroverses Thema.

Nach einem weiteren Jahrtausend gaben die AT & T Indian Hill C-Standards für Stil und Kodierung Folgendes an:

> Header-Dateien sollten nicht verschachtelt sein. Der Prolog für eine Header-Datei sollte daher beschreiben, welche anderen Header `#include` d sein müssen, damit der Header funktionsfähig ist. In extremen Fällen, in denen eine große Anzahl von Header-Dateien in mehreren verschiedenen Quelldateien enthalten ist, ist es zulässig, alle gängigen `#include` Dateien in einer Include-Datei zu speichern.

Dies ist der Gegensatz von Selbstbeherrschung.

Moderne Regeln

Seitdem tendiert die Meinung jedoch in die entgegengesetzte Richtung. Wenn eine Quelldatei die von einem Header `header.h` deklarierten `header.h` , kann der Programmierer `header.h` schreiben:

```
#include "header.h"
```

und (vorbehaltlich der Einstellung der richtigen Suchpfade in der Befehlszeile) werden alle erforderlichen Kopfzeilen von `header.h` ohne dass weitere Kopfzeilen in die Quelldatei `header.h` werden müssen.

Dies bietet eine bessere Modularität für den Quellcode. Sie schützt die Quelle auch vor dem Rätsel "Vermutung, warum dieser Header hinzugefügt wurde", das entsteht, nachdem der Code für ein oder zwei Jahrzehnte geändert und gehackt wurde.

Der Kodierungsstandard der NASA Goddard Space Flight Center (GSFC) für C ist einer der moderneren Standards - er ist jedoch etwas schwer zu finden. Es besagt, dass Header in sich abgeschlossen sein sollten. Es bietet auch eine einfache Möglichkeit, um sicherzustellen, dass Header in sich geschlossen sind: Die Implementierungsdatei für den Header sollte den Header als

ersten Header enthalten. Wenn es nicht eigenständig ist, wird der Code nicht kompiliert.

Die Begründung des GSFC beinhaltet:

> § 2.1.1.1 Header-Include-Begründung
>
> Dieser Standard erfordert, dass der Header einer Unit `#include` Anweisungen für alle anderen Header enthält, die für den Unit-Header erforderlich sind. Durch das Platzieren von `#include` für den Einheitenheader im Einheitenkörper kann der Compiler überprüfen, ob der Header alle erforderlichen `#include` Anweisungen enthält.
>
> Ein alternatives Design, das in dieser Norm nicht zulässig ist, erlaubt keine `#include` Anweisungen in Headern. Alle `#includes` sind in den Body-Dateien. Unit-Header-Dateien müssen dann #ifdef-Anweisungen enthalten, die überprüfen, ob die erforderlichen Header in der richtigen Reihenfolge enthalten sind.
>
> Ein Vorteil des alternativen Designs besteht darin, dass die `#include` Liste in der Hauptdatei genau die Abhängigkeitsliste ist, die in einem Makefile benötigt wird. Diese Liste wird vom Compiler geprüft. Bei der Standardausführung muss ein Werkzeug zum Generieren der Abhängigkeitsliste verwendet werden. Alle von der Branche empfohlenen Entwicklungsumgebungen bieten jedoch ein solches Werkzeug.
>
> Ein Hauptnachteil des alternativen Designs besteht darin, dass bei Änderungen der erforderlichen Header-Liste einer Einheit jede Datei, die diese Unit verwendet, bearbeitet werden muss, um die `#include` Anweisungsliste zu aktualisieren. Außerdem kann sich die erforderliche Headerliste für eine Compiler-Bibliothekseinheit auf verschiedenen Zielen unterscheiden.
>
> Ein weiterer Nachteil des alternativen Designs besteht darin, dass die Header-Dateien der Compiler-Bibliothek und andere Dateien von Drittanbietern geändert werden müssen, um die erforderlichen `#ifdef` Anweisungen hinzuzufügen.

Selbstbeherrschung bedeutet also:

- Wenn für einen Header `header.h` ein neuer verschachtelter Header `extra.h` , müssen Sie nicht jede Quelldatei, die `header.h` verwendet, `header.h` , ob Sie `extra.h` hinzufügen `extra.h` .
- Wenn ein Header `header.h` nicht länger einen bestimmten Header `notneeded.h` , müssen Sie nicht jede Quelldatei, die `header.h` verwendet, `header.h` , ob `notneeded.h` sicher entfernt werden kann .
- Sie müssen nicht die richtige Reihenfolge für das Einfügen der erforderlichen Header festlegen (dies erfordert eine topologische Sortierung, um die Aufgabe ordnungsgemäß auszuführen).

Selbstbeherrschung prüfen

Siehe Verknüpfen mit einer statischen Bibliothek für ein Skript- `chkhdr` , mit dem die Idempotenz und die Selbstbeschränkung einer Header-Datei `chkhdr` können.

Minimalität

Header sind ein wichtiger Mechanismus zur Überprüfung der Konsistenz, sollten jedoch so klein wie möglich sein. Das bedeutet insbesondere, dass ein Header keine anderen Header enthalten sollte, nur weil die Implementierungsdatei die anderen Header benötigt. Ein Header sollte nur die Header enthalten, die für einen Verbraucher der beschriebenen Dienste erforderlich sind.

Beispielsweise sollte ein Projektheader `<stdio.h>` nicht enthalten, es sei denn, eine der Funktionsschnittstellen verwendet den Typ `FILE *` (oder einen der anderen, nur in `<stdio.h>` definierten Typen). Wenn ein Interface `size_t` , ist der kleinste Header `<stddef.h>` . Wenn ein anderer Header enthalten ist, der `size_t` definiert, ist es nicht erforderlich, auch `<stddef.h>` .

Wenn die Kopfzeilen minimal sind, wird auch die Übersetzungszeit auf ein Minimum reduziert.

Es ist möglich, Header zu entwickeln, deren einziger Zweck darin besteht, viele andere Header einzubeziehen. Dies stellt sich auf lange Sicht selten als eine gute Idee heraus, da nur wenige Quelldateien alle von den Kopfzeilen beschriebenen Einrichtungen benötigen. Beispielsweise könnte ein `<standard-ch>` entworfen werden, der alle Standard-C-Header enthält - mit Vorsicht, da einige Header nicht immer vorhanden sind. Allerdings verwenden nur wenige Programme die Funktionen von `<locale.h>` oder `<tgmath.h>` .

- Siehe auch So verknüpfen Sie mehrere Implementierungsdateien in C?

Einschließen, was Sie verwenden (IWYU)

Das Google Include Your You Use- Projekt oder IWYU stellt sicher, dass die Quelldateien alle im Code verwendeten Header enthalten.

Angenommen , eine Quelldatei `source.c` einen Header enthält `arbitrary.h` die zufälligerweise wiederum enthält `freeloader.h` , aber die Quelldatei auch explizit und unabhängig verwendet die Einrichtungen von `freeloader.h` . Alles ist gut zu beginnen. Dann wird eines Tages `arbitrary.h` geändert, so dass seine Kunden die Einrichtungen von `freeloader.h` nicht mehr benötigen. Plötzlich hört `source.c` Kompilieren auf, weil es die IWYU-Kriterien nicht erfüllt. Da der Code in `source.c` ausdrücklich die Einrichtungen verwendet `freeloader.h` , hätte es enthalten , was es verwendet - es sollte eine explizite gewesen `#include "freeloader.h"` in der Quelle zu. (Idempotenz hätte sichergestellt, dass es kein Problem gab.)

Die IWYU-Philosophie maximiert die Wahrscheinlichkeit, dass Code trotz vernünftiger Änderungen an Schnittstellen weiterhin kompiliert wird. Wenn Ihr Code eine Funktion aufruft, die später von der veröffentlichten Schnittstelle entfernt wird, kann keine Vorbereitung ausreichend sein, damit Änderungen nicht erforderlich werden. Aus diesem Grund werden Änderungen an APIs nach Möglichkeit vermieden, und es gibt Abwertungszyklen für mehrere Releases usw.

Dies ist ein besonderes Problem in C ++, da Standardkopfzeilen sich gegenseitig einschließen dürfen. Die `file.cpp` könnte einen Header `header1.h` , der auf einer Plattform einen anderen Header `header2.h` . `file.cpp` könnte auch die Möglichkeiten von `header2.h` . Dies wäre kein Problem zunächst sein - würde der Code kompiliert , da `header1.h` enthält `header2.h` . Auf einer anderen Plattform oder einem Upgrade der aktuellen Plattform könnte `header1.h` überarbeitet werden,

sodass diese `header2.h` `file.cpp` nicht mehr enthält. `header2.h` diesem `file.cpp` würde `file.cpp` das Kompilieren beenden.

IWYU würde das Problem erkennen und empfehlen, `header2.h` direkt in `file.cpp` . Dies würde sicherstellen, dass es weiterhin kompiliert wird. Analoge Überlegungen gelten auch für C-Code.

Notation und Verschiedenes

Der C-Standard sagt, dass es sehr wenig Unterschiede zwischen den Schreibweisen `#include <header.h>` und `#include "header.h"` .

> [`#include <header.h>`] durchsucht eine Sequenz implementierungsdefinierter Stellen nach einem Header, der durch die angegebene Sequenz zwischen den Trennzeichen `<` und `>` eindeutig identifiziert wird, und bewirkt, dass diese Direktive durch den gesamten Inhalt des Headers ersetzt wird. Wie die Bereiche angegeben werden oder wie der Header identifiziert wird, ist durch die Implementierung definiert.
>
> [`#include "header.h"`] bewirkt, dass diese Direktive durch den gesamten Inhalt der Quelldatei ersetzt wird, die in der angegebenen Reihenfolge zwischen den Trennzeichen `"…"` . Die benannte Quelldatei wird implementierungsdefiniert gesucht. Wenn diese Suche nicht unterstützt wird oder die Suche fehlschlägt, wird die Direktive erneut verarbeitet, als ob sie [`#include <header.h>`] lesen würde.

Daher kann die doppelt zitierte Form an mehr Stellen aussehen als in der eckigen Klammer. In der Norm ist beispielhaft festgelegt, dass die Standardheader in spitzen Klammern enthalten sein sollten, obwohl die Kompilierung funktioniert, wenn Sie stattdessen doppelte Anführungszeichen verwenden. In ähnlicher Weise verwenden Standards wie POSIX das spitze Klammerformat - und das sollten Sie auch. Reservieren Sie doppelte Anführungszeichen für vom Projekt definierte Header. Bei extern definierten Kopfzeilen (einschließlich Kopfzeilen aus anderen Projekten, auf die sich Ihr Projekt stützt) ist die Notation der spitzen Klammern am besten geeignet.

Beachten Sie, dass zwischen `#include` und dem Header ein Leerzeichen stehen sollte, obwohl die Compiler dort kein Leerzeichen akzeptieren. Räume sind billig.

Einige Projekte verwenden eine Notation wie:

```
#include <openssl/ssl.h>
#include <sys/stat.h>
#include <linux/kernel.h>
```

Sie sollten überlegen, ob Sie diese Namespace-Steuerung in Ihrem Projekt verwenden möchten (dies ist wahrscheinlich eine gute Idee). Sie sollten sich von den Namen fernhalten, die in vorhandenen Projekten verwendet werden (insbesondere wären sowohl `sys` als auch `linux` eine schlechte Wahl).

Wenn Sie dies verwenden, sollte Ihr Code bei der Verwendung der Notation vorsichtig und konsistent sein.

Verwenden Sie nicht `#include "../include/header.h"` .

Header-Dateien sollten selten, wenn überhaupt Variablen definiert werden. Obwohl Sie globale Variablen auf ein Minimum beschränken, müssen Sie sie, wenn Sie eine globale Variable benötigen, in einem Header deklarieren und in einer geeigneten Quelldatei definieren. Diese Quelldatei enthält den Header, um die Deklaration und Definition zu überprüfen und alle Quelldateien, die die Variable verwenden, verwenden den Header, um sie zu deklarieren.

Folgerung: Sie deklarieren keine globalen Variablen in einer Quelldatei - eine Quelldatei enthält nur Definitionen.

Header-Dateien sollten selten `static` Funktionen deklarieren, mit der bemerkenswerten Ausnahme von `static inline` Funktionen, die in Headern definiert werden, wenn die Funktion in mehr als einer Quelldatei benötigt wird.

- Quelldateien definieren globale Variablen und globale Funktionen.
- Quelldateien erklären nicht das Vorhandensein globaler Variablen oder Funktionen. Sie enthalten den Header, der die Variable oder Funktion deklariert.
- Header-Dateien deklarieren globale Variablen und Funktionen (und Typen und anderes unterstützendes Material).
- Header-Dateien definieren keine Variablen oder Funktionen außer (`static`) `inline` Funktionen.

Querverweise

- Wo sollen Funktionen in C dokumentiert werden?
- Liste der Standardheaderdateien in C und C ++
- Ist `inline` ohne `static` oder `extern` in C99 überhaupt nützlich?
- Wie verwende ich `extern` , um Variablen zwischen Quelldateien zu teilen?
- Was sind die Vorteile eines relativen Pfads wie `"../include/header.h"` für einen Header?
- Optimierung der Header-Inklusion
- Soll ich jeden Header angeben?

Erstellen Sie Header-Dateien und fügen Sie sie ein online lesen: https://riptutorial.com/de/c/topic/6257/erstellen-sie-header-dateien-und-fugen-sie-sie-ein

Kapitel 19: Fehlerbehandlung

Syntax

- #include <errno.h>
- int errno; / * Implementierung definiert * /
- #include <string.h>
- char * strerror (int errnum);
- #include <stdio.h>
- void perror (const Zeichen * s);

Bemerkungen

`errno` dass `errno` nicht unbedingt eine Variable ist, sondern dass die Syntax nur ein Hinweis darauf ist, wie sie deklariert werden *könnte* . Auf vielen modernen Systemen mit Thread-Schnittstellen ist `errno` ein Makro, das sich in ein Objekt auflöst, das lokal für den aktuellen Thread ist.

Examples

errno

Wenn eine Standardbibliotheksfunktion fehlschlägt, wird für errno häufig der entsprechende Fehlercode festgelegt. Der C-Standard erfordert, dass mindestens 3 Werte für das Errno eingestellt werden:

Wert	Bedeutung
EDOM	Domänenfehler
ERANGE	Bereichsfehler
EILSEQ	Unzulässige Multi-Byte-Zeichenfolge

Strerror

Wenn `perror` nicht flexibel genug ist, erhalten Sie eine vom Benutzer lesbare Fehlerbeschreibung, indem Sie `strerror` von `<string.h>` .

```
int main(int argc, char *argv[])
{
    FILE *fout;
    int last_error = 0;

    if ((fout = fopen(argv[1], "w")) == NULL) {
        last_error = errno;
```

```
        /* reset errno and continue */
        errno = 0;
    }

    /* do some processing and try opening the file differently, then */


    if (last_error) {
        fprintf(stderr, "fopen: Could not open %s for writing: %s",
                argv[1], strerror(last_error));
        fputs("Cross fingers and continue", stderr);
    }

    /* do some other processing */

    return EXIT_SUCCESS;
}
```

perror

Rufen Sie `perror` aus <stdio.h> auf, um eine vom Benutzer lesbare Fehlermeldung an `stderr` `perror` .

```
int main(int argc, char *argv[])
{
   FILE *fout;

   if ((fout = fopen(argv[1], "w")) == NULL) {
      perror("fopen: Could not open file for writing");
      return EXIT_FAILURE;
   }
return EXIT_SUCCESS;
}
```

Dadurch wird eine Fehlermeldung zum aktuellen Wert von `errno` .

Fehlerbehandlung online lesen: https://riptutorial.com/de/c/topic/2486/fehlerbehandlung

Kapitel 20: Folgepunkte

Bemerkungen

Internationale Norm ISO / IEC 9899: 201x Programmiersprachen - C

> Auf ein flüchtiges Objekt zuzugreifen, ein Objekt zu ändern, eine Datei zu ändern oder eine Funktion aufzurufen, die eine dieser Operationen *ausführt* , sind alles *Nebeneffekte* , die den Status der Ausführungsumgebung beeinflussen.
>
> Das Vorhandensein eines *Sequenzpunktes* zwischen der Auswertung der Ausdrücke A und B impliziert, dass jede mit A verbundene Wertberechnung und Nebenwirkung vor jeder mit B verbundenen Wertberechnung und Nebenwirkung sequenziert wird.

Hier ist die vollständige Liste der Sequenzpunkte aus Anhang C des Online-Vorentwurfs der Veröffentlichung von 2011 für die C-Standardsprache:

> **Folgepunkte**
>
> 1 Im Folgenden werden die in 5.1.2.3 beschriebenen Sequenzpunkte beschrieben:
>
> - Zwischen den Auswertungen des Funktionsbezeichners und den tatsächlichen Argumenten in einem Funktionsaufruf und dem tatsächlichen Aufruf. (6.5.2.2).
> - Zwischen den Bewertungen des ersten und des zweiten Operanden der folgenden Operatoren: logisches AND `&&` (6.5.13); logisches ODER `||` (6.5.14); comma `,` (6.5.17).
> - Zwischen den Bewertungen des ersten Operanden des Bedingten `? :` Operator und welcher der zweite und dritte Operand ausgewertet wird (6.5.15).
> - Das Ende eines vollständigen Deklarators: Deklaratoren (6.7.6);
> - Zwischen der Auswertung eines vollständigen Ausdrucks und dem nächsten zu bewertenden vollen Ausdruck. Die folgenden Ausdrücke sind vollständige Ausdrücke: ein Initialisierer, der nicht Teil eines zusammengesetzten Literal (6.7.9) ist; der Ausdruck in einer Ausdrucksanweisung (6.8.3); der steuernde Ausdruck einer Auswahlanweisung (`if` oder `switch`) (6.8.4); die Steuerung der Expression eines `while` oder `do` Anweisung (6.8.5); jeder der (optionalen) Ausdrücke einer `for` Anweisung (6.8.5.3); der (optionale) Ausdruck in einer `return` Anweisung (6.8.6.4).
> - Unmittelbar bevor eine Bibliotheksfunktion zurückkehrt (7.1.4).
> - Nach den Aktionen, die jedem formatierten Eingabe- / Ausgabefunktionsumwandlungsbezeichner zugeordnet sind (7.21.6, 7.29.2).
> - Unmittelbar vor und unmittelbar nach jedem Aufruf einer Vergleichsfunktion sowie zwischen jedem Aufruf einer Vergleichsfunktion und jeder Bewegung der Objekte, die als Argumente an diesen Aufruf übergeben werden (7.22.5).

Examples

Sequenzierte Ausdrücke

Die folgenden Ausdrücke werden *sequenziert* :

```
a && b
a || b
a , b
a ? b : c
for ( a ; b ; c ) { ... }
```

In allen Fällen wird der Ausdruck `a` vollständig ausgewertet und *alle Nebenwirkungen werden angewendet,* bevor entweder `b` oder `c` ausgewertet werden. Im vierten Fall wird nur eine von `b` oder `c` ausgewertet. Im letzten Fall wird `b` vollständig ausgewertet und alle Nebenwirkungen werden angewendet, bevor `c` ausgewertet wird.

In allen Fällen wird die Bewertung von Ausdruck `a` *vor* den Bewertungen von `b` oder `c` *sequenziert* (alternativ werden die Bewertungen von `b` und `c` *nach* der Bewertung von `a`) *sequenziert* .

So mögen Ausdrücke

```
x++ && x++
x++ ? x++ : y++
(x = f()) && x != 0
for ( x = 0; x < 10; x++ ) { ... }
y = (x++, x++);
```

gut definiertes Verhalten haben.

Nicht aufeinanderfolgende Ausdrücke

C11

Die folgenden Ausdrücke sind nicht *aufeinanderfolgend* :

```
a + b;
a - b;
a * b;
a / b;
a % b;
a & b;
a | b;
```

In den obigen Beispielen kann der Ausdruck `a` vor oder nach dem Ausdruck `b` bewertet werden, `b` kann vor `a` ausgewertet werden, oder sie können sogar gemischt werden, wenn sie mehreren Anweisungen entsprechen.

Eine ähnliche Regel gilt für Funktionsaufrufe:

```
f(a, b);
```

Hier nicht nur `a` und `b` sind unsequenced (dh die `,` Operator in einem Funktionsaufruf *keinen* Sequenzpunkt erzeugen) , sondern auch `f` , der Ausdruck, der die Funktion bestimmt , die aufgerufen werden soll.

Nebenwirkungen können unmittelbar nach der Bewertung angewendet oder auf einen späteren Zeitpunkt verschoben werden.

Ausdrücke wie

```
x++ & x++;
f(x++, x++); /* the ',' in a function call is *not* the same as the comma operator */
x++ * x++;
a[i] = i++;
```

oder

```
x++ & x;
f(x++, x);
x++ * x;
a[i++] = i;
```

wird *unbestimmtes Verhalten ergeben,* weil

- Eine Änderung eines Objekts und jeglicher anderer Zugriff darauf muss sequenziert werden
- Die Reihenfolge der Bewertung und die Reihenfolge, in der die *Nebenwirkungen* [1] angewendet werden, ist nicht festgelegt.

---

1 Alle Änderungen im Status der Ausführungsumgebung.

Unbestimmte sequenzierte Ausdrücke

Funktionsaufrufe als `f(a)` implizieren immer einen Sequenzpunkt zwischen der Auswertung der Argumente und dem Bezeichner (hier `f` und `a`) und dem tatsächlichen Aufruf. Wenn zwei solcher Aufrufe nicht sequenziell sind, werden die beiden Funktionsaufrufe unbestimmt sequenziert, dh einer wird vor dem anderen ausgeführt, und die Reihenfolge ist nicht spezifiziert.

```
unsigned counter = 0;

unsingned account(void) {
   return counter++;
}

int main(void) {
   printf("the order is %u %u\n", account(), account());
}
```

Diese implizite zweifache Änderung des `counter` während der Auswertung der `printf` Argumente ist gültig. Wir wissen nur nicht, welcher der Aufrufe zuerst kommt. Da die Bestellung nicht

spezifiziert ist, kann sie variieren und ist nicht abhängig. Der Ausdruck könnte also lauten:

> die Reihenfolge ist 0 1

oder

> Die Reihenfolge ist 1 0

Die analoge Aussage zu dem obigen ohne Zwischenfunktionsaufruf

```
printf("the order is %u %u\n", counter++, counter++); // undefined behavior
```

hat ein undefiniertes Verhalten, da es keinen Sequenzpunkt zwischen den beiden Modifikationen des `counter` .

Folgepunkte online lesen: https://riptutorial.com/de/c/topic/1275/folgepunkte

Kapitel 21: Formatierte Eingabe / Ausgabe

Examples

Den Wert eines Zeigers auf ein Objekt drucken

Um den Wert eines Zeigers auf ein Objekt zu drucken (im Gegensatz zu einem Funktionszeiger), verwenden Sie den `p` Konvertierungsspezifizierer. Es ist definiert, nur `void` Zeiger auszudrucken. Um den Wert eines nicht `void` Zeigers auszudrucken, muss er explizit ("casted *") in `void*` konvertiert werden.

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h>  /* for printf() */

int main(void)
{
  int i;
  int * p = &i;

  printf("The address of i is %p.\n", (void*) p);

  return EXIT_SUCCESS;
}
```

C99

Verwenden von `<inttypes.h>` und `uintptr_t`

Eine andere Möglichkeit, Zeiger in C99 oder höher zu drucken, verwendet den Typ `uintptr_t` und die Makros aus `<inttypes.h>` :

```
#include <inttypes.h> /* for uintptr_t and PRIXPTR */
#include <stdio.h>    /* for printf() */

int main(void)
{
  int  i;
  int *p = &i;

  printf("The address of i is 0x%" PRIXPTR ".\n", (uintptr_t)p);

  return 0;
}
```

Theoretisch gibt es möglicherweise keinen Integer-Typ, der einen in eine Integer-Zahl konvertierten Zeiger enthalten kann (daher ist der Typ `uintptr_t` möglicherweise nicht vorhanden). In der Praxis existiert es. Zeiger auf Funktionen müssen nicht in den Typ `uintptr_t` konvertierbar sein - auch wenn sie meistens konvertierbar sind.

Wenn der Typ `uintptr_t` vorhanden ist, gilt auch der Typ `intptr_t` . Es ist jedoch nicht klar, warum

Sie Adressen jemals als signierte Ganzzahlen behandeln möchten.

K & R C89

Vorgeschichte:

Während der K & R-C-Zeiten vor C89 gab es weder einen Typ `void*` (noch Header `<stdlib.h>`) noch Prototypen und daher keine `int main(void)` Notation `int main(void)` Notation). Daher wurde der Zeiger in ein `long unsigned int` und mit der `long unsigned int lx` Längenmodifizierer / Konvertierungsspezifizierer.

Das folgende Beispiel dient nur zu Informationszwecken. Heutzutage handelt es sich um ungültigen Code, der das berüchtigte undefinierte Verhalten sehr wohl provozieren könnte.

```
#include <stdio.h> /* optional in pre-standard C - for printf() */

int main()
{
  int  i;
  int *p = &i;

  printf("The address of i is 0x%lx.\n", (long unsigned) p);

  return 0;
}
```

Drucken der Differenz der Werte zweier Zeiger auf ein Objekt

Das Abziehen der Werte von zwei Zeigern auf ein Objekt führt zu einer vorzeichenbehafteten Ganzzahl * 1 . Es würde also *mindestens mit* dem `d` Konvertierungsspezifizierer gedruckt.

Um sicherzustellen, dass ein Typ breit genug ist, um eine solche "Zeigerdifferenz" aufzunehmen, da C99 `<stddef.h>` den Typ `ptrdiff_t` definiert. Um ein `ptrdiff_t` zu drucken, verwenden Sie den Modifikator `t` length.

C99

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h> /* for printf() */
#include <stddef.h> /* for ptrdiff_t */


int main(void)
{
  int a[2];
  int * p1 = &a[0], * p2 = &a[1];
  ptrdiff_t pd = p2 - p1;

  printf("p1 = %p\n", (void*) p1);
  printf("p2 = %p\n", (void*) p2);
  printf("p2 - p1 = %td\n", pd);

  return EXIT_SUCCESS;
```

```
}
```

Das Ergebnis könnte so aussehen:

```
p1 = 0x7fff6679f430
p2 = 0x7fff6679f434
p2 - p1 = 1
```

Bitte beachten Sie, dass der sich ergebende Wert der Differenz durch die Größe des Typs skaliert wird, auf den der Zeiger, auf den hier subtrahiert wird, ein `int` Wert ist. Die Größe eines `int` für dieses Beispiel ist 4.

---

* [1] Wenn die zwei zu subtrahierenden Zeiger nicht auf dasselbe Objekt zeigen, ist das Verhalten undefiniert.

Konvertierungsspezifikationen für den Druck

Conversion Specifier	Art des Arguments	Beschreibung
`i` , `d`	int	druckt dezimal
`u`	unsigned int	druckt dezimal
`o`	unsigned int	druckt oktal
`x`	unsigned int	druckt hexadezimal und in Kleinbuchstaben
`X`	unsigned int	druckt hexadezimal in Großbuchstaben
`f`	doppelt	druckt Float mit einer Standardgenauigkeit von 6, wenn keine Genauigkeit angegeben ist (Kleinbuchstaben werden für spezielle Zahlen `nan` und `inf` oder `infinity`)
`F`	doppelt	druckt Float mit einer Standardgenauigkeit von 6, wenn keine Genauigkeit angegeben ist (Großbuchstaben werden für die Sondernummern `NAN` und `INF` oder `INFINITY`)
`e`	doppelt	druckt Float mit einer Standardgenauigkeit von 6, wenn keine Genauigkeit angegeben wird, unter Verwendung der wissenschaftlichen Notation mit Mantisse / Exponent; Exponent- und Sondernummern für Kleinbuchstaben
`E`	doppelt	druckt Float mit einer Standardgenauigkeit von 6, wenn keine Genauigkeit angegeben wird, unter Verwendung der wissenschaftlichen Notation mit Mantisse / Exponent; Exponent- und Sondernummern für Großbuchstaben

Conversion Specifier	Art des Arguments	Beschreibung
`g`	doppelt	verwendet entweder `f` oder `e` *[siehe unten]*
`G`	doppelt	verwendet entweder `F` oder `E` *[siehe unten]*
`a`	doppelt	druckt hexadezimal und in Kleinbuchstaben
`A`	doppelt	druckt hexadezimal in Großbuchstaben
`c`	verkohlen	druckt ein einzelnes Zeichen
`s`	verkohlen*	gibt eine Zeichenfolge bis zu einem `NUL` Terminator aus oder wird auf die durch die Genauigkeit angegebene Länge gekürzt, sofern angegeben
`p`	Leere*	druckt einen `void` Zeigerwert; Ein nicht `void` pointer sollte explizit in `void*` umgewandelt werden ("cast"); Zeiger nur auf Objekt, **nicht** Funktionszeiger
`%`	n / a	gibt das Zeichen `%`
`n`	int *	***schreibe*** die Anzahl der Bytes, die bisher gedruckt wurden, in das `int` auf das gezeigt wird.

Beachten Sie, dass `%hhn` auf `%n` angewendet werden können (z. B. bedeutet `%hhn` , dass *ein nachfolgender `n` Konvertierungsspezifizierer auf einen Zeiger auf ein `signed char` Argument* gemäß ISO / IEC 9899: 2011 (§7.21.6.1 ¶7 angewendet wird).

Beachten Sie, dass die Fließkomma-Konvertierungen aufgrund von Standard-Promotion-Regeln für die Typen `float` und `double` gelten *. Die Standardargument-Promotions werden für nachfolgende Argumente ausgeführt.*) Funktionen wie `printf()` werden also immer nur mit `double` , auch wenn die referenzierte Variable vom Typ `float` .

Bei den Formaten `g` und `G` ist die Wahl zwischen der Notation `e` und `f` (oder `E` und `F`) im C-Standard und in der POSIX-Spezifikation für `printf()` dokumentiert:

> Das Doppelargument, das eine Gleitkommazahl darstellt, wird abhängig von dem umgesetzten Wert und der Genauigkeit in den Stil `f` oder `e` (oder im Fall eines `G` Konvertierungsspezifizierers in `F` oder `E`) konvertiert. Sei `P` gleich der Genauigkeit, falls nicht Null, 6, wenn die Genauigkeit ausgelassen wird, oder 1, wenn die Genauigkeit Null ist. Dann, wenn eine Konvertierung mit Stil `E` einen Exponenten von `X` :
>
> - Wenn P> X> = -4, muss die Konvertierung mit dem Stil `f` (oder `F`) und der Genauigkeit `P - (X+1)` .
> - Andernfalls erfolgt die Konvertierung mit dem Stil `e` (oder `E`) und der Genauigkeit `P - 1` .

Wenn das Flag '#' nicht verwendet wird, werden abschließende Nullen aus dem gebrochenen Teil des Ergebnisses entfernt und das Dezimalzeichen wird entfernt, wenn kein verbleibender Bruchteil vorhanden ist.

Die Funktion printf ()

Die Funktion `printf()` ist das wichtigste Werkzeug, das zum Drucken von Text an die Konsole in C verwendet wird, indem `<stdio.h>` `printf()` wird.

```
printf("Hello world!");
// Hello world!
```

Normale, unformatierte Zeichen-Arrays können selbst gedruckt werden, indem sie direkt zwischen den Klammern platziert werden.

```
printf("%d is the answer to life, the universe, and everything.", 42);
// 42 is the answer to life, the universe, and everything.

int x = 3;
char y = 'Z';
char* z = "Example";
printf("Int: %d, Char: %c, String: %s", x, y, z);
// Int: 3, Char: Z, String: Example
```

Alternativ können ganze Zahlen, Gleitkommazahlen, Zeichen und mehr mit dem Escape-Zeichen `%` gedruckt werden, gefolgt von einem Zeichen oder einer Zeichenfolge, die das *Format angibt (Formatbezeichner)* .

Alle zusätzlichen Argumente für die Funktion `printf()` werden durch Kommas getrennt. Diese Argumente sollten in derselben Reihenfolge wie die Formatbezeichner stehen. Zusätzliche Argumente werden ignoriert, während falsch eingegebene Argumente oder fehlende Argumente Fehler oder undefiniertes Verhalten verursachen. Jedes Argument kann entweder ein Literalwert oder eine Variable sein.

Nach erfolgreicher Ausführung wird die Anzahl der gedruckten Zeichen mit dem Typ `int` . Andernfalls liefert ein Fehler einen negativen Wert.

Längenmodifikatoren

Die Standards C99 und C11 geben die folgenden Längenmodifizierer für `printf()` . ihre Bedeutungen sind:

Modifikator	Ändert	Gilt für
hh	d, i, o, u, x oder X	`char` , `signed char` oder `unsigned char`
h	d, i, o, u, x oder X	`short int` oder `unsigned short int`
l	d, i, o, u, x oder X	`long int` oder `unsigned long int`

Modifikator	Ändert	Gilt für
l	a, A, e, E, f, F, g oder G	`double` (zur Kompatibilität mit `scanf()` ; undefiniert in C90)
ll	d, i, o, u, x oder X	`long long int` oder `unsigned long long int`
j	d, i, o, u, x oder X	`intmax_t` oder `uintmax_t`
z	d, i, o, u, x oder X	`size_t` oder der entsprechende signierte Typ (`ssize_t` in POSIX)
t	d, i, o, u, x oder X	`ptrdiff_t` oder der entsprechende Integer-Typ ohne Vorzeichen
L	a, A, e, E, f, F, g oder G	`long double`

Wenn ein Längenmodifikator mit einem anderen als dem oben angegebenen Konvertierungsspezifizierer angezeigt wird, ist das Verhalten undefiniert.

Microsoft gibt einige andere Längenmodifizierer an und unterstützt `hh` , `j` , `z` oder `t` nicht explizit.

Modifikator	Ändert	Gilt für
I32	d, i, o, x oder X	`__int32`
I32	o, u, x oder X	`unsigned __int32`
I64	d, i, o, x oder X	`__int64`
I64	o, u, x oder X	`unsigned __int64`
ich	d, i, o, x oder X	`ptrdiff_t` (`__int32` auf 32-Bit-Plattformen, `__int64` auf 64-Bit-Plattformen)
ich	o, u, x oder X	`size_t` (`unsigned __int32` auf 32-Bit-Plattformen `unsigned __int64` auf 64-Bit-Plattformen `unsigned __int64`)
l oder l	a, A, e, E, f, g oder G	`long double` (In Visual C ++ ist, obwohl `long double` ein eindeutiger Typ ist, er dieselbe interne Darstellung wie `double` .)
l oder w	c oder c	Breites Zeichen mit Funktionen `printf` und `wprintf` . (Ein `wC` `lc` , `lC` , `wc` oder `wC` ist gleichbedeutend mit `c` in `printf` Funktionen und mit `c` in `wprintf` Funktionen.)

Modifikator	Ändert	Gilt für
l oder w	s, S oder Z	`wprintf` Funktionen `printf` und `wprintf` . (Ein `ls` , `lS` , `ws` oder `wS` Typbezeichner ist gleichbedeutend mit `S` in `printf` Funktionen und mit `s` in `wprintf` Funktionen.)

Beachten Sie, dass die `C` , `S` und `Z` Konvertierungsspezifizierer und die `I32` `I` , `I32` , `I64` und `w` Microsoft-Erweiterungen sind. Die Behandlung von `l` als Modifikator für `long double` anstelle von `double` unterscheidet sich vom Standard, obwohl Sie den Unterschied kaum erkennen können, es sei denn, `long double` hat eine andere Darstellung als `double` .

Formatflags drucken

Der C-Standard (auch C11 und C99) definiert die folgenden Flags für `printf()` :

Flagge	Konvertierungen	Bedeutung
`-`	alles	Das Ergebnis der Konvertierung ist innerhalb des Feldes linksbündig. Die Konvertierung ist rechtsbündig, wenn dieses Flag nicht angegeben ist.
`+`	signiert numerisch	Das Ergebnis einer signierten Konvertierung beginnt immer mit einem Zeichen ('+' oder '-'). Die Konvertierung darf nur dann mit einem Vorzeichen beginnen, wenn ein negativer Wert konvertiert wird, wenn dieses Flag nicht angegeben ist.
`<space>`	signiert numerisch	Ist das erste Zeichen einer signierten Konvertierung kein Vorzeichen oder führt eine signierte Konvertierung zu keinen Zeichen, wird dem Ergebnis ein `<space>` vorangestellt. Das bedeutet, wenn das `<space>` und das ' `+` ' Flag beide erscheinen, wird das `<space>` Flag ignoriert.
`#`	alles	Gibt an, dass der Wert in ein alternatives Formular konvertiert werden soll. Für `o` Umwandlung, so hat er die Genauigkeit zu erhöhen, wenn und nur wenn notwendig, um die erste Ziffer des Ergebnisses zu zwingen , eine Null zu sein (wenn der Wert und Präzision beide 0 ist, wird eine einzelne 0 gedruckt). Für `x` oder `X` Konvertierungsbezeichner wird einem Ergebnis ungleich Null ein `0x` (oder `0X`) vorangestellt. Für `a` , `A` , `e` , `E` , `f` , `F` , `g` , und `G` Konvertierungsspezifizierer, wird das Ergebnis immer ein Radix - Zeichen enthalten, auch wenn keine Ziffern stellen den radix Zeichen folgen. Ohne dieses Flag erscheint im Ergebnis dieser Konvertierungen nur ein Radix-Zeichen, wenn eine Ziffer darauf folgt. Bei `g` und `G` Konvertierungsspezifizierern werden nachfolgende Nullen nicht wie üblich aus dem Ergebnis entfernt. Bei anderen Konvertierungsspezifizierern ist das Verhalten undefiniert.

Flagge	Konvertierungen	Bedeutung
`0`	numerisch	Für d, i, o, u, x, a, a, e, f, g, g und G-Konvertierungsspezifizierer werden führende Nullen (nach beliebigen Zeichen- oder Basisangaben) zum Auffüllen auf das Feld verwendet Breite anstelle von Leerzeichen, außer bei der Konvertierung einer Unendlichkeit oder eines NaN. Wenn die Flag '0' und '-' beide erscheinen, wird das Flag '0' ignoriert. Wenn für die Konvertierungskennungen d, i, o, u, x und X eine Genauigkeit angegeben wird, wird das Flag '0' ignoriert. Wenn die Flags '0' und `<apostrophe>` angezeigt werden, werden die Gruppierungszeichen vor dem Auffüllen mit Nullen eingefügt. Bei anderen Konvertierungen ist das Verhalten undefiniert.

Diese Flags werden auch von Microsoft mit den gleichen Bedeutungen unterstützt.

Die POSIX-Spezifikation für `printf()` fügt hinzu:

Flagge	Konvertierungen	Bedeutung
`'`	i, d, u, f, F, g, G	Der ganzzahlige Teil des Ergebnisses einer Dezimalkonvertierung wird mit Gruppierungszeichen von Tausenden formatiert. Bei anderen Konvertierungen ist das Verhalten undefiniert. Das nicht monetäre Gruppierungszeichen wird verwendet.

Formatierte Eingabe / Ausgabe online lesen: https://riptutorial.com/de/c/topic/3750/formatierte-eingabe---ausgabe

Kapitel 22: Frameworks testen

Einführung

Viele Entwickler verwenden Komponententests, um zu überprüfen, ob ihre Software wie erwartet funktioniert. Unit-Tests prüfen kleine Einheiten von größeren Software-Einheiten und stellen sicher, dass die Ausgaben den Erwartungen entsprechen. Test-Frameworks erleichtern das Testen von Einheiten, indem sie Auf- / Abbau-Services anbieten und die Tests koordinieren.

Für C stehen viele Unit-Test-Frameworks zur Verfügung. Beispielsweise ist Unity ein reines C-Framework. Häufig verwenden Leute C ++ - Testframeworks, um C-Code zu testen. Es gibt auch viele C ++ - Testframeworks.

Bemerkungen

Test Harness:

TDD - Test Driven Development:

Doppelmechanismen in C testen:

1. Link-Time-Substitution
2. Funktionszeigerersetzung
3. Preprozessor-Ersetzung
4. Kombinierte Linkzeit- und Funktionszeigerersetzung

Hinweis zu C ++ - Testframeworks, die in C verwendet werden: Die Verwendung von C ++ - Frameworks zum Testen eines C-Programms ist üblich, wie hier erläutert.

Examples

CppUTest

CppUTest ist ein xUnit -Style-Framework für Unit-Tests von C und C ++. Es ist in C ++ geschrieben und zielt auf Portabilität und Einfachheit beim Design ab. Es unterstützt die Erkennung von Speicherlecks, das Erstellen von Mocks und das Ausführen seiner Tests zusammen mit dem Google-Test. Kommt mit Hilfsskripten und Beispielprojekten für Visual Studio und Eclipse CDT.

```
#include <CppUTest/CommandLineTestRunner.h>
#include <CppUTest/TestHarness.h>


TEST_GROUP(Foo_Group) {}

TEST(Foo_Group, Foo_TestOne) {}
```

```
/* Test runner may be provided options, such
   as to enable colored output, to run only a
   specific test or a group of tests, etc. This
   will return the number of failed tests. */

int main(int argc, char ** argv)
{
    RUN_ALL_TESTS(argc, argv);
}
```

Eine Testgruppe kann eine `setup()` und eine `teardown()` -Methode haben. Die `setup` Methode wird vor jedem Test und die `teardown()` -Methode danach aufgerufen. Beide sind optional und können unabhängig voneinander weggelassen werden. Andere Methoden und Variablen können auch innerhalb einer Gruppe deklariert werden und stehen allen Tests dieser Gruppe zur Verfügung.

```
TEST_GROUP(Foo_Group)
{
    size_t data_bytes = 128;
    void * data;

    void setup()
    {
        data = malloc(data_bytes);
    }

    void teardown()
    {
        free(data);
    }

    void clear()
    {
        memset(data, 0, data_bytes);
    }
}
```

Unity Test Framework

Unity ist ein xUnit- Test-Framework für Komponententests C. Es ist vollständig in C geschrieben und ist portabel, schnell, einfach, ausdrucksstark und erweiterbar. Es ist so konzipiert, dass es sich insbesondere für Unit-Tests für eingebettete Systeme eignet.

Ein einfacher Testfall, der den Rückgabewert einer Funktion überprüft, könnte folgendermaßen aussehen

```
void test_FunctionUnderTest_should_ReturnFive(void)
{
    TEST_ASSERT_EQUAL_INT( 5, FunctionUnderTest() );
}
```

Eine vollständige Testdatei könnte folgendermaßen aussehen:

```
#include "unity.h"
#include "UnitUnderTest.h" /* The unit to be tested. */
```

```
void setUp (void) {} /* Is run before every test, put unit init calls here. */
void tearDown (void) {} /* Is run after every test, put unit clean-up calls here. */

void test_TheFirst(void)
{
    TEST_IGNORE_MESSAGE("Hello world!"); /* Ignore this test but print a message. */
}

int main (void)
{
    UNITY_BEGIN();
    RUN_TEST(test_TheFirst); /* Run the test. */
    return UNITY_END();
}
```

Unity enthält einige Beispielprojekte, Makefiles und einige Ruby-Rake-Skripts, die das Erstellen längerer Testdateien erleichtern.

CMocka

CMocka ist ein elegantes Komponententest- Framework für C mit Unterstützung für Scheinobjekte . Es benötigt nur die Standard-C-Bibliothek, arbeitet mit einer Reihe von Computerplattformen (einschließlich Embedded) und mit verschiedenen Compilern. Es enthält ein Tutorial zum Testen mit Mocks, API-Dokumentation und eine Vielzahl von Beispielen .

```
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>

void null_test_success (void ** state) {}

void null_test_fail (void ** state)
{
    assert_true (0);
}

/* These functions will be used to initialize
   and clean resources up after each test run */
int setup (void ** state)
{
    return 0;
}

int teardown (void ** state)
{
    return 0;
}


int main (void)
{
    const struct CMUnitTest tests [] =
    {
        cmocka_unit_test (null_test_success),
        cmocka_unit_test (null_test_fail),
```

```
    };

    /* If setup and teardown functions are not
       needed, then NULL may be passed instead */

    int count_fail_tests =
        cmocka_run_group_tests (tests, setup, teardown);

    return count_fail_tests;
}
```

Frameworks testen online lesen: https://riptutorial.com/de/c/topic/6779/frameworks-testen

Kapitel 23: Funktionsparameter

Bemerkungen

In C ist es üblich, Rückgabewerte zu verwenden, um auf auftretende Fehler hinzuweisen. und um Daten durch die Verwendung von übergebenen Zeigern zurückzugeben. Dies kann aus mehreren Gründen erfolgen. Dazu gehört, dass auf dem Heap kein Speicher zugeordnet werden muss oder dass die statische Zuordnung an dem Punkt verwendet wird, an dem die Funktion aufgerufen wird.

Examples

Verwenden von Zeigerparametern, um mehrere Werte zurückzugeben

Ein übliches Muster in C zum einfachen Nachahmen mehrerer Werte aus einer Funktion ist die Verwendung von Zeigern.

```
#include <stdio.h>

void Get( int* c , double* d )
{
    *c = 72;
    *d = 175.0;
}

int main(void)
{
    int a = 0;
    double b = 0.0;

    Get( &a , &b );

    printf("a: %d, b: %f\n", a , b );

    return 0;
}
```

Übergabe von Arrays an Funktionen

```
int getListOfFriends(size_t size, int friend_indexes[]) {
  size_t i = 0;
  for (; i < size; i++) {
    friend_indexes[i] = i;
  }
}
```

C99 C11

```
/* Type "void" and VLAs ("int friend_indexes[static size]") require C99 at least.
   In C11 VLAs are optional. */
void getListOfFriends(size_t size, int friend_indexes[static size]) {
```

```
  size_t i = 0;
  for (; i < size; i++) {
    friend_indexes[i] = 1;
  }
}
```

Hier fordert die `static` innerhalb des `[]` des Funktionsparameters an, dass das Argument-Array mindestens so viele Elemente enthalten muss, wie angegeben (dh `size`). Um der Lage sein , diese Funktion verwenden wir , um sicherzustellen haben , dass die `size` Parameter kommt vor dem Array - Parameter in der Liste.

Verwenden Sie `getListOfFriends()` wie `getListOfFriends()` :

```
#define LIST_SIZE (50)

int main(void) {
  size_t size_of_list = LIST_SIZE;
  int friends_indexes[size_of_list];

  getListOfFriends(size_of_list, friend_indexes); /* friend_indexes decays to a pointer to the
                                                     address of its 1st element:
                                                                      &friend_indexes[0] */

  /* Here friend_indexes carries: {0, 1, 2, ..., 49}; */

  return 0;
}
```

Siehe auch

Übergeben von mehrdimensionalen Arrays an eine Funktion

Parameter werden per Wert übergeben

In C werden alle Funktionsparameter als Wert übergeben. Wenn Sie also ändern, was in Aufgerufsfunktionen übergeben wird, hat dies keine Auswirkungen auf die lokalen Variablen der Anruferfunktionen.

```
#include <stdio.h>

void modify(int v) {
    printf("modify 1: %d\n", v); /* 0 is printed */
    v = 42;
    printf("modify 2: %d\n", v); /* 42 is printed */
}

int main(void) {
    int v = 0;
    printf("main 1: %d\n", v); /* 0 is printed */
    modify(v);
    printf("main 2: %d\n", v); /* 0 is printed, not 42 */
    return 0;
}
```

Sie können Zeiger verwenden, um die lokalen Variablen der aufrufenden Funktionen durch Aufräumfunktionen zu ändern. Beachten Sie, dass dies nicht als *Referenz übergeben wird,* sondern die Zeigerwerte , die auf die lokalen Variablen zeigen, übergeben werden.

```
#include <stdio.h>

void modify(int* v) {
    printf("modify 1: %d\n", *v); /* 0 is printed */
    *v = 42;
    printf("modify 2: %d\n", *v); /* 42 is printed */
}

int main(void) {
    int v = 0;
    printf("main 1: %d\n", v); /* 0 is printed */
    modify(&v);
    printf("main 2: %d\n", v); /* 42 is printed */
    return 0;
}
```

Das Zurückgeben der Adresse einer lokalen Variablen an den Angerufenen führt jedoch zu undefiniertem Verhalten. Siehe Dereferenzieren eines Zeigers auf eine Variable, deren Lebensdauer überschritten wird .

Reihenfolge der Funktionsparameterausführung

Die Reihenfolge der Ausführung von Parametern ist in der C-Programmierung nicht festgelegt. Hier kann es von links nach rechts oder von rechts nach links ausgeführt werden. Die Reihenfolge hängt von der Implementierung ab.

```
#include <stdio.h>

void function(int a, int b)
{
    printf("%d %d\n", a, b);
}

int main(void)
{
    int a = 1;
    function(a++, ++a);
    return 0;
}
```

Beispiel für eine Funktion, die eine Struktur zurückgibt, die Werte mit Fehlercodes enthält

Bei den meisten Beispielen einer Funktion, die einen Wert zurückgibt, muss ein Zeiger als eines der Argumente angegeben werden, damit die Funktion den Wert, auf den gezeigt wird, ähnlich wie im Folgenden ändern kann. Der tatsächliche Rückgabewert der Funktion ist normalerweise ein Typ wie ein `int` , um den Status des Ergebnisses anzuzeigen, unabhängig davon, ob es funktioniert hat oder nicht.

```
int func (int *pIvalue)
{
    int iRetStatus = 0;             /* Default status is no change */

    if (*pIvalue > 10) {
        *pIvalue = *pIvalue * 45;   /* Modify the value pointed to */
        iRetStatus = 1;             /* indicate value was changed */
    }

    return iRetStatus;              /* Return an error code */
}
```

Sie können jedoch auch eine `struct` als Rückgabewert verwenden, mit der Sie sowohl einen Fehlerstatus als auch andere Werte zurückgeben können. Zum Beispiel.

```
typedef struct {
    int    iStat;      /* Return status */
    int    iValue;     /* Return value */
}  RetValue;

RetValue func (int iValue)
{
    RetValue iRetStatus = {0, iValue};

    if (iValue > 10) {
        iRetStatus.iValue = iValue * 45;
        iRetStatus.iStat = 1;
    }

    return iRetStatus;
}
```

Diese Funktion kann dann wie im folgenden Beispiel verwendet werden.

```
int usingFunc (int iValue)
{
    RetValue iRet = func (iValue);

    if (iRet.iStat == 1) {
        /* do things with iRet.iValue, the returned value */
    }
    return 0;
}
```

Oder es könnte wie folgt verwendet werden.

```
int usingFunc (int iValue)
{
    RetValue iRet;

    if ( (iRet = func (iValue)).iStat == 1 ) {
        /* do things with iRet.iValue, the returned value */
    }
    return 0;
}
```

Funktionsparameter online lesen: https://riptutorial.com/de/c/topic/1006/funktionsparameter

Kapitel 24: Funktionszeiger

Einführung

Funktionszeiger sind Zeiger, die auf Funktionen statt auf Datentypen verweisen. Sie können verwendet werden, um Variabilität in der aufzurufenden Funktion zur Laufzeit zu ermöglichen.

Syntax

- returnType (* name) (Parameter)
- typedef returnType (* name) (Parameter)
- typedef returnType Name (Parameter);
  Name Name;
- typedef returnType Name (Parameter);
  typedef Name * NamePtr;

Examples

Funktionszeiger zuweisen

```
#include <stdio.h>

/* increment: take number, increment it by one, and return it */
int increment(int i)
{
    printf("increment %d by 1\n", i);
    return i + 1;
}

/* decrement: take number, decrement it by one, and return it */
int decrement(int i)
{
    printf("decrement %d by 1\n", i);
    return i - 1;
}

int main(void)
{
    int num = 0;          /* declare number to increment */
    int (*fp)(int);       /* declare a function pointer */

    fp = &increment;      /* set function pointer to increment function */
    num = (*fp)(num);     /* increment num */
    num = (*fp)(num);     /* increment num a second time */

    fp = &decrement;      /* set function pointer to decrement function */
    num = (*fp)(num);     /* decrement num */
    printf("num is now: %d\n", num);
```

```
    return 0;
}
```

Funktionszeiger von einer Funktion zurückgeben

```
#include <stdio.h>

enum Op
{
  ADD = '+',
  SUB = '-',
};


/* add: add a and b, return result */
int add(int a, int b)
{
    return a + b;
}

/* sub: subtract b from a, return result */
int sub(int a, int b)
{
    return a - b;
}

/* getmath: return the appropriate math function */
int (*getmath(enum Op op))(int,int)
{
    switch (op)
    {
        case ADD:
            return &add;
        case SUB:
            return &sub;
        default:
            return NULL;
    }
}

int main(void)
{
    int a, b, c;
    int (*fp)(int,int);

    fp = getmath(ADD);

    a = 1, b = 2;
    c = (*fp)(a, b);
    printf("%d + %d = %d\n", a, b, c);
    return 0;
}
```

Best Practices

Mit typedef

Es kann nützlich sein, ein `typedef` anstatt den Funktionszeiger jedes Mal von Hand zu deklarieren.

Die Syntax zum Deklarieren eines `typedef` für einen Funktionszeiger lautet:

```
typedef returnType (*name)(parameters);
```

Beispiel:

Positiv, dass wir eine Funktion, `sort` , haben, die einen Funktionszeiger auf einen Funktionsvergleich erwartet `compare` so dass:

> compare - Eine Vergleichsfunktion für zwei Elemente, die einer Sortierfunktion zugeführt werden soll.
>
> Es wird erwartet, dass "compare" 0 zurückgibt, wenn die beiden Elemente als gleich betrachtet werden, ein positiver Wert, wenn das erste übergebene Element in gewissem Sinne "größer" ist als das letzte Element, und ansonsten gibt die Funktion einen negativen Wert zurück (was bedeutet, dass das erste Element ist.) "weniger" als das letztere).

Ohne ein `typedef` würden wir einen Funktionszeiger auf folgende Weise als Argument an eine Funktion übergeben:

```
void sort(int (*compare)(const void *elem1, const void *elem2)) {
    /* inside of this block, the function is named "compare" */
}
```

Mit einem `typedef` wir schreiben:

```
typedef int (*compare_func)(const void *, const void *);
```

und dann könnten wir die Funktionssignatur von `sort` in ändern:

```
void sort(compare_func func) {
    /* In this block the function is named "func" */
}
```

beide `sort` würden jede Funktion des Formulars akzeptieren

```
int compare(const void *arg1, const void *arg2) {
    /* Note that the variable names do not have to be "elem1" and "elem2" */
}
```

Funktionszeiger sind die einzige Stelle, an der Sie die Zeiger-Eigenschaft des Typs `typedef struct something_struct *something_type` Versuchen Sie nicht, Typen wie `typedef struct something_struct *something_type` zu definieren. Dies gilt auch für eine Struktur mit Mitgliedern, auf die nicht direkt von API-Aufrufern zugegriffen werden soll, z. B. der Typ stdio.h `FILE` (der, wie Sie jetzt feststellen

werden, kein Zeiger ist).

Kontext-Zeiger nehmen

Ein Funktionszeiger sollte fast immer eine vom Benutzer angegebene Leere * als Kontextzeiger verwenden.

Beispiel

```
/* function minimiser, details unimportant */
double findminimum( double (*fptr)(double x, double y, void *ctx), void *ctx)
{
    ...
    /* repeatedly make calls like this */
    temp = (*fptr)(testx, testy, ctx);
}

/* the function we are minimising, sums two cubics */
double *cubics(double x, double y, void *ctx)
{
    double *coeffsx = ctx;
    double *coeffsy = coeffx + 4;

    return coeffsx[0] * x * x * x + coeffsx[1] * x * x + coeffsx[2] * x + coeffsx[3] +
           coeffsy[0] * y * y * y + coeffsy[1] * y * y + coeffsy[2] * y + coeffsy[3];

}

void caller()
{
    /* context, the coefficients of the cubics */
    double coeffs[8] = {1, 2, 3, 4, 5, 6, 7, 8};
    double min;

    min = findminimum(cubics, coeffs);
}
```

Die Verwendung des Kontextzeigers bedeutet, dass die zusätzlichen Parameter nicht fest in der Funktion, auf die gezeigt wird, codiert werden müssen, oder dass sie die Verwendung von Globalen erfordern.

Die Bibliotheksfunktion `qsort()` folgt dieser Regel nicht und für triviale Vergleichsfunktionen kann man oft ohne Kontext auskommen. Für etwas komplizierter wird jedoch der Kontextzeiger wesentlich.

Siehe auch

Funktionszeiger

Einführung

Genau wie `char` und `int` ist eine Funktion ein grundlegendes Merkmal von C. Als solches können Sie einen Zeiger auf einen deklarieren: Dies bedeutet, dass Sie die zu übergebende *Funktion an* eine andere Funktion übergeben können, um ihre Aufgabe zu erfüllen. Wenn Sie beispielsweise über eine `graph()` Funktion verfügen, die eine Grafik anzeigt, können Sie die zu graphierende *Funktion an* `graph()` .

```
// A couple of external definitions to make the example clearer
extern unsigned int screenWidth;
extern void plotXY(double x, double y);

// The graph() function.
// Pass in the bounds: the minimum and maximum X and Y that should be plotted.
// Also pass in the actual function to plot.
void graph(double minX, double minY,
           double maxX, double maxY,
           ???? *fn) {             // See below for syntax

    double stepX = (maxX - minX) / screenWidth;
    for (double x=minX; x<maxX; x+=stepX) {

        double y = fn(x);          // Get y for this x by calling passed-in fn()

        if (minY<=y && y<maxY) {
            plotXY(x, y);          // Plot calculated point
        } // if
    } for
} // graph(minX, minY, maxX, maxY, fn)
```

Verwendungszweck

Der obige Code zeigt also die von Ihnen übergebene Funktion an - sofern diese Funktion bestimmte Kriterien erfüllt: nämlich, dass Sie ein `double` und ein `double` . Es gibt viele Funktionen wie `sin()` , `cos()` , `tan()` , `exp()` usw. - aber es gibt viele, die es nicht gibt, wie zB `graph()` selbst!

Syntax

Wie legen Sie also fest, welche Funktionen Sie in `graph()` und welche nicht. Die herkömmliche Methode ist die Verwendung einer Syntax, die möglicherweise nicht leicht zu lesen oder zu verstehen ist:

```
double (*fn)(double); // fn is a pointer-to-function that takes a double and returns one
```

Das obige Problem besteht darin, dass zwei Dinge gleichzeitig versucht werden: die Struktur der Funktion und die Tatsache, dass es sich um einen Zeiger handelt. Teilen Sie also die beiden Definitionen auf! Durch die Verwendung von `typedef` kann jedoch eine bessere Syntax (besser lesbar und verständlicher) erreicht werden.

Mnemonik zum Schreiben von Funktionszeigern

Alle C-Funktionen sind tatsächlich Zeiger auf eine Stelle im Programmspeicher, an der Code vorhanden ist. Ein Funktionszeiger dient hauptsächlich dazu, anderen Funktionen einen "Rückruf" zu geben (oder Klassen und Objekte zu simulieren).

Die Syntax einer Funktion, wie weiter unten auf dieser Seite definiert, lautet:

returnType (* name) (Parameter)

Eine Mnemonik zum Schreiben einer Funktionszeigerdefinition ist die folgende Prozedur:

1. Beginnen Sie mit dem Schreiben einer normalen Funktionsdeklaration: `returnType name(parameters)`
2. Umbrechen des Funktionsnamens mit der `returnType (*name)(parameters)` : `returnType (*name)(parameters)`

Grundlagen

So wie Sie einen Zeiger auf **int** , **char** , **float** , **array / string** , **struct** usw. haben können, können Sie auch einen Zeiger auf eine Funktion haben.

Beim Deklarieren des Zeigers werden der *Rückgabewert der Funktion* , der *Name der Funktion* und der *Typ der empfangenen Argumente / Parameter verwendet* .

Angenommen, Sie haben die folgende Funktion deklariert und initialisiert:

```
int addInt(int n, int m){
    return n+m;
}
```

Sie können einen Zeiger auf diese Funktion deklarieren und initialisieren:

```
int (*functionPtrAdd)(int, int) = addInt; // or &addInt - the & is optional
```

Wenn Sie eine Void-Funktion haben, könnte dies folgendermaßen aussehen:

```
void Print(void){
    printf("look ma' - no hands, only pointers!\n");
}
```

Dann würde der Zeiger darauf deklarieren:

```
void (*functionPtrPrint)(void) = Print;
```

Für den Zugriff auf die Funktion selbst muss der Zeiger dereferenziert werden:

```
sum = (*functionPtrAdd)(2, 3); //will assign 5 to sum
(*functionPtrPrint)(); //will print the text in Print function
```

Wie in den fortgeschritteneren Beispielen in diesem Dokument zu sehen ist, kann das Deklarieren eines Zeigers auf eine Funktion unübersichtlich werden, wenn der Funktion mehr als ein paar

Parameter übergeben werden. Wenn Sie einige Zeiger auf Funktionen mit identischer "Struktur" haben (gleiche Art von Rückgabewert und gleiche Art von Parametern), **empfiehlt** es sich, den **typedef-** Befehl zu verwenden, um Ihnen die Eingabe zu ersparen und den Code deutlicher zu machen:

```
typedef int (*ptrInt)(int, int);

int Add(int i, int j){
    return i+j;
}

int Multiply(int i, int j){
    return i*j;
}

int main()
{
    ptrInt ptr1 = Add;
    ptrInt ptr2 = Multiply;

    printf("%d\n", (*ptr1)(2,3)); //will print 5
    printf("%d\n", (*ptr2)(2,3)); //will print 6
    return 0;
}
```

Sie können auch ein **Array von Funktionszeigern** erstellen. Wenn alle Zeiger dieselbe "Struktur" haben:

```
int (*array[2]) (int x, int y); // can hold 2 function pointers
array[0] = Add;
array[1] = Multiply;
```

Hier und hier erfahren Sie mehr.

Es ist auch möglich, ein Array von Funktionszeigern verschiedener Typen zu definieren. Dies erfordert jedoch ein Casting, wenn Sie auf die jeweilige Funktion zugreifen möchten. Sie können erfahren Sie mehr hier .

Funktionszeiger online lesen: https://riptutorial.com/de/c/topic/250/funktionszeiger

Kapitel 25: Generische Auswahl

Syntax

- _Generic (Zuweisungsausdruck, generische Assoc-Liste)

Parameter

Parameter	Einzelheiten
generische assoc-liste	generic-association **ODER** generic-assoc-liste, generic-association
generische Vereinigung	Typname: Zuweisungsausdruck **ODER** Standard: Zuweisungsausdruck

Bemerkungen

1. Alle `_Generic` werden während der Auswertung des `_Generic` Primärausdrucks `_Generic` .
2. `_Generic` Primärausdruck wird in Übersetzungsphase 7 ausgewertet. Daher wurden Phasen wie die Verkettung von Strings vor der Auswertung abgeschlossen.

Examples

Prüfen Sie, ob eine Variable einen bestimmten qualifizierten Typ hat

```
#include <stdio.h>

#define is_const_int(x) _Generic((&x),  \
        const int *: "a const int",     \
        int *:       "a non-const int", \
        default:     "of other type")

int main(void)
{
    const int i = 1;
    int j = 1;
    double k = 1.0;
    printf("i is %s\n", is_const_int(i));
    printf("j is %s\n", is_const_int(j));
    printf("k is %s\n", is_const_int(k));
}
```

Ausgabe:

```
i is a const int
j is a non-const int
k is of other type
```

Wenn jedoch das generische Typmakro folgendermaßen implementiert ist:

```
#define is_const_int(x) _Generic((x), \
        const int: "a const int",     \
        int:       "a non-const int", \
        default:   "of other type")
```

Die Ausgabe ist:

```
i is a non-const int
j is a non-const int
k is of other type
```

Dies liegt daran, dass alle `_Generic` für die Auswertung des steuernden Ausdrucks eines `_Generic` Primärausdrucks `_Generic` .

Typisches generisches Druckmakro

```
#include <stdio.h>

void print_int(int x) { printf("int: %d\n", x); }
void print_dbl(double x) { printf("double: %g\n", x); }
void print_default() { puts("unknown argument"); }

#define print(X) _Generic((X), \
        int: print_int, \
        double: print_dbl, \
        default: print_default)(X)

int main(void) {
    print(42);
    print(3.14);
    print("hello, world");
}
```

Ausgabe:

```
int: 42
double: 3.14
unknown argument
```

Wenn der Typ weder `int` noch `double` , wird eine Warnung generiert. Um die Warnung zu beseitigen, können Sie diesen Typ zum Makro `print(X)` hinzufügen.

Generische Auswahl basierend auf mehreren Argumenten

Wenn eine Auswahl mehrerer Argumente für einen generischen `_Generic` gewünscht wird und alle `_Generic` Typen arithmetische Typen sind, können verschachtelte `_Generic` Ausdrücke auf einfache Weise `_Generic` indem die Parameter im steuernden Ausdruck `_Generic` :

```
int max_int(int, int);
unsigned max_unsigned(unsigned, unsigned);
```

```
double max_double(double, double);

#define MAX(X, Y) _Generic((X)+(Y),                 \
                           int:      max_int,       \
                           unsigned: max_unsigned, \
                           default:  max_double)    \
                    ((X), (Y))
```

Hier wird der steuernde Ausdruck `(X)+(Y)` nur nach Typ geprüft und nicht ausgewertet. Die üblichen Konvertierungen für arithmetische Operanden werden durchgeführt, um den ausgewählten Typ zu bestimmen.

In einer komplexeren Situation kann eine Auswahl auf der Grundlage mehrerer Argumente für den Bediener getroffen werden, indem diese verschachtelt werden.

In diesem Beispiel werden vier extern implementierte Funktionen ausgewählt, die Kombinationen von zwei Argumenten int und / oder string annehmen und deren Summe zurückgeben.

```
int AddIntInt(int a, int b);
int AddIntStr(int a, const char* b);
int AddStrInt(const char*  a, int b );
int AddStrStr(const char*  a, const char*  b);

#define AddStr(y)                            \
   _Generic((y),         int: AddStrInt,     \
                       char*: AddStrStr,     \
                 const char*: AddStrStr )

#define AddInt(y)                            \
   _Generic((y),          int: AddIntInt,    \
                       char*: AddIntStr,     \
                 const char*: AddIntStr )

#define Add(x, y)                            \
   _Generic((x) ,        int: AddInt(y) ,    \
                      char*: AddStr(y) ,     \
                const char*: AddStr(y))      \
                        ((x), (y))

int main( void )
{
    int result = 0;
    result = Add( 100 , 999 );
    result = Add( 100 , "999" );
    result = Add( "100" , 999 );
    result = Add( "100" , "999" );

    const int a = -123;
    char b[] = "4321";
    result = Add( a , b );

    int c = 1;
    const char d[] = "0";
    result = Add( d , ++c );
}
```

Obwohl es so aussieht, als würde Argument `y` mehr als einmal ausgewertet, ist es nicht [1] . Beide

Argumente werden nur einmal am Ende des Makros Add: `( x , y )` ausgewertet, genau wie bei einem normalen Funktionsaufruf.

---

[1] (Zitiert aus: ISO: IEC 9899: 201X 6.5.1.1 Generische Auswahl 3)
Der steuernde Ausdruck einer generischen Auswahl wird nicht ausgewertet.

Generische Auswahl online lesen: https://riptutorial.com/de/c/topic/571/generische-auswahl

Kapitel 26: Gewerkschaften

Examples

Unterschied zwischen struct und union

Dies zeigt, dass Gewerkschaftsmitglieder Speicher gemeinsam nutzen und dass Strukturmitglieder Speicher nicht gemeinsam nutzen.

```
#include <stdio.h>
#include <string.h>

union My_Union
{
  int variable_1;
  int variable_2;
};

struct My_Struct
{
  int variable_1;
  int variable_2;
};

int main (void)
{
  union My_Union u;
  struct My_Struct s;
  u.variable_1 = 1;
  u.variable_2 = 2;
  s.variable_1 = 1;
  s.variable_2 = 2;
  printf ("u.variable_1: %i\n", u.variable_1);
  printf ("u.variable_2: %i\n", u.variable_2);
  printf ("s.variable_1: %i\n", s.variable_1);
  printf ("s.variable_2: %i\n", s.variable_2);
  printf ("sizeof (union My_Union): %i\n", sizeof (union My_Union));
  printf ("sizeof (struct My_Struct): %i\n", sizeof (struct My_Struct));
  return 0;
}
```

Verwendung von Vereinigungen, um Werte neu zu interpretieren

Einige C-Implementierungen erlauben es dem Code, in ein Mitglied eines Unionstyps zu schreiben und dann von einem anderen zu lesen, um eine Art Neuinterpretation der Besetzung durchzuführen (Parsen des neuen Typs als Bitdarstellung des alten).

Beachten Sie jedoch, dass dies vom aktuellen oder früheren C-Standard nicht zulässig ist und zu undefiniertem Verhalten führt. Dies ist jedoch eine sehr häufige Erweiterung, die von Compilern angeboten wird (überprüfen Sie daher die Compiler-Dokumente, wenn Sie dies vorhaben.) .

Ein reales Beispiel für diese Technik ist der "Fast Inverse Square Root" -Algorithmus, der sich auf

Implementierungsdetails von IEEE 754-Gleitkommazahlen stützt, um eine inverse Quadratwurzel schneller als mit Gleitkommaoperationen auszuführen. Dieser Algorithmus kann entweder durch Zeigergießen ausgeführt werden (was sehr gefährlich ist und gegen die strikte Aliasing-Regel verstößt) oder durch eine union (was immer noch undefiniertes Verhalten ist, aber in vielen Compilern funktioniert):

```
union floatToInt
{
    int32_t intMember;
    float floatMember; /* Float must be 32 bits IEEE 754 for this to work */
};

float inverseSquareRoot(float input)
{
    union floatToInt x;
    int32_t i;
    float f;
    x.floatMember = input;     /* Assign to the float member */
    i = x.intMember;           /* Read back from the integer member */
    i = 0x5f3759df - (i >> 1);
    x.intMember = i;           /* Assign to the integer member */
    f = x.floatMember;         /* Read back from the float member */
    f = f * (1.5f - input * 0.5f * f * f);
    return f * (1.5f - input * 0.5f * f * f);
}
```

Diese Technik wurde in der Vergangenheit aufgrund ihrer höheren Geschwindigkeit im Vergleich zu Fließkommaoperationen häufig in Computergrafiken und Spielen verwendet und stellt einen Kompromiss dar. Sie verliert an Genauigkeit und ist im Gegenzug für Geschwindigkeit nicht tragbar.

Einem Gewerkschaftsmitglied schreiben und von einem anderen lesen

Die Mitglieder einer Gewerkschaft teilen sich den gleichen Speicherplatz. Das bedeutet, dass das Schreiben an ein Mitglied die Daten aller anderen Mitglieder überschreibt und dass das Lesen eines Mitglieds die gleichen Daten ergibt wie das Lesen aller anderen Mitglieder. Da Gewerkschaftsmitglieder jedoch unterschiedliche Typen und Größen haben können, können die gelesenen Daten unterschiedlich interpretiert werden, siehe
http://www.riptutorial.com/c/example/9399/using-unions-to-reinterpret-values

Das folgende einfache Beispiel zeigt eine Vereinigung mit zwei Mitgliedern des gleichen Typs. Es zeigt, dass das Schreiben in Member `m_1` führt, dass der geschriebene Wert aus Member `m_2` gelesen wird, und das Schreiben in Member `m_2` führt, dass der geschriebene Wert aus Member `m_1` gelesen wird.

```
#include <stdio.h>

union my_union /* Define union */
{
    int m_1;
    int m_2;
};
```

```
int main (void)
{
    union my_union u;             /* Declare union */
    u.m_1 = 1;                    /* Write to m_1 */
    printf("u.m_2: %i\n", u.m_2); /* Read from m_2 */
    u.m_2 = 2;                    /* Write to m_2 */
    printf("u.m_1: %i\n", u.m_1); /* Read from m_1 */
    return 0;
}
```

Ergebnis

```
u.m_2: 1
u.m_1: 2
```

Gewerkschaften online lesen: https://riptutorial.com/de/c/topic/7645/gewerkschaften

Kapitel 27: Häufige Fehler

Einführung

In diesem Abschnitt werden einige der häufigsten Fehler beschrieben, die ein C-Programmierer kennen sollte und die er vermeiden sollte. Weitere Informationen zu unerwarteten Problemen und deren Ursachen finden Sie unter Undefiniertes Verhalten

Examples

Mischen von vorzeichenbehafteten und vorzeichenlosen Ganzzahlen in arithmetischen Operationen

Es ist normalerweise keine gute Idee, `signed` und `unsigned` Ganzzahlen in arithmetischen Operationen zu mischen. Was wird zum Beispiel in folgendem Beispiel ausgegeben?

```
#include <stdio.h>

int main(void)
{
    unsigned int a = 1000;
    signed int b = -1;

    if (a > b) puts("a is more than b");
    else puts("a is less or equal than b");

    return 0;
}
```

Da 1000 mehr als -1 ist, würde man erwarten, dass die Ausgabe `a is more than b` ist. Dies wird jedoch nicht der Fall sein.

Arithmetische Operationen zwischen verschiedenen Integraltypen werden innerhalb eines allgemeinen Typs ausgeführt, der durch die so genannten üblichen arithmetischen Konvertierungen definiert wird (siehe Sprachspezifikation, 6.3.1.8).

In diesem Fall ist der "common type" `unsigned int` , weil, wie in Üblichen arithmetischen Konvertierungen angegeben ,

> 714 Wenn der Operand mit einem vorzeichenlosen Integer-Typ einen Rang hat, der größer oder gleich dem Rang des Typs des anderen Operanden ist, wird der Operand mit dem vorzeichenbehafteten Integer-Typ in den Typ des Operanden mit dem vorzeichenlosen Integer-Typ umgewandelt.

Das bedeutet, dass `int` Operand `b` vor dem Vergleich in `unsigned int` konvertiert wird.

Wenn -1 in ein `unsigned int` konvertiert wird, ist das Ergebnis der maximal mögliche `unsigned int` Wert, der größer als 1000 ist, was bedeutet, dass `a > b` falsch ist.

Beim Vergleichen fälschlicherweise = anstelle von ==

Der Operator `=` wird für die Zuweisung verwendet.

Der Operator `==` wird zum Vergleich verwendet.

Man sollte vorsichtig sein, die beiden nicht zu mischen. Manchmal schreibt man irrtümlich

```
/* assign y to x */
if (x = y) {
     /* logic */
}
```

wenn was wirklich gesucht wurde, ist:

```
/* compare if x is equal to y */
if (x == y) {
    /* logic */
}
```

Erstere weist x einen Wert von y zu und prüft, ob dieser Wert nicht Null ist, anstatt einen Vergleich durchzuführen. Dies ist äquivalent zu:

```
if ((x = y) != 0) {
    /* logic */
}
```

Es gibt Zeiten, in denen das Testen des Ergebnisses einer Zuweisung beabsichtigt ist und häufig verwendet wird, weil dadurch vermieden werden muss, dass Code kopiert werden muss und das erste Mal speziell behandelt werden muss. Vergleichen Sie

```
while ((c = getopt_long(argc, argv, short_options, long_options, &option_index)) != -1) {
        switch (c) {
        ...
        }
}
```

gegen

```
c = getopt_long(argc, argv, short_options, long_options, &option_index);
while (c != -1) {
        switch (c) {
        ...
        }
        c = getopt_long(argc, argv, short_options, long_options, &option_index);
}
```

Moderne Compiler erkennen dieses Muster und warnen nicht, wenn sich die Zuweisung in Klammern befindet (siehe oben). Möglicherweise warnt sie jedoch vor anderen Verwendungen. Zum Beispiel:

```
if (x = y)          /* warning */

if ((x = y))        /* no warning */
if ((x = y) != 0)   /* no warning; explicit */
```

Einige Programmierer verwenden die Strategie, die Konstante links vom Operator zu platzieren (im Allgemeinen als Yoda-Bedingungen bezeichnet). Da Konstanten R-Werte sind, führt der Stil dieser Bedingung dazu, dass der Compiler einen Fehler ausgibt, wenn der falsche Operator verwendet wurde.

```
if (5 = y) /* Error */

if (5 == y) /* No error */
```

Dies verringert jedoch die Lesbarkeit des Codes erheblich und wird nicht als notwendig erachtet, wenn der Programmierer die guten C-Codierpraktiken befolgt und beim Vergleich zweier Variablen nicht hilfreich ist. Darüber hinaus geben viele moderne Compiler möglicherweise Warnungen aus, wenn Code mit Yoda-Bedingungen geschrieben wird.

Unvorsichtige Verwendung von Semikolons

Seien Sie vorsichtig mit Semikolons. Folgendes Beispiel

```
if (x > a);
   a = x;
```

bedeutet eigentlich:

```
if (x > a) {}
a = x;
```

Das heißt, `x` wird auf jeden Fall `a` zugewiesen, was möglicherweise nicht das war, was Sie ursprünglich wollten.

Das Fehlen eines Semikolons verursacht manchmal ein unbemerktes Problem:

```
if (i < 0)
    return
day = date[0];
hour = date[1];
minute = date[2];
```

Das Semikolon hinter return wird verfehlt, so dass day = date [0] zurückgegeben wird.

Eine Methode, um dieses und ähnliche Probleme zu vermeiden, besteht darin, bei mehrzeiligen Bedingungen und Schleifen immer Klammern zu verwenden. Zum Beispiel:

```
if (x > a) {
    a = x;
```

```
}
```

Vergessen, ein zusätzliches Byte für \ 0 zuzuweisen

`strlen` immer daran, 1 zu `strlen` zu addieren, wenn Sie eine Zeichenfolge in einen `malloc` ed Puffer `strlen` .

```
char *dest = malloc(strlen(src)); /* WRONG */
char *dest = malloc(strlen(src) + 1); /* RIGHT */

strcpy(dest, src);
```

Dies liegt daran, dass `strlen` das nachgestellte `\0` in der Länge nicht enthält. Wenn Sie den `WRONG` (wie oben gezeigt) `strcpy` , würde Ihr Programm beim Aufruf von `strcpy` undefiniertes Verhalten aufrufen.

Dies gilt auch für Situationen, in denen Sie eine Zeichenfolge mit bekannter maximaler Länge von `stdin` oder einer anderen Quelle lesen. Zum Beispiel

```
#define MAX_INPUT_LEN 42

char buffer[MAX_INPUT_LEN]; /* WRONG */
char buffer[MAX_INPUT_LEN + 1]; /* RIGHT */

scanf("%42s", buffer);  /* Ensure that the buffer is not overflowed */
```

Vergessen, Speicher freizugeben (Speicherverluste)

Eine bewährte Programmiermethode besteht darin, Speicher freizugeben, der direkt durch eigenen Code zugewiesen wurde, oder implizit durch Aufrufen einer internen oder externen Funktion, z. B. einer Bibliotheks-API wie `strdup()` . Wenn Sie den Speicher nicht freigeben, kann dies zu einem Speicherverlust führen, der sich in einem erheblichen Teil des vergeudeten Speichers ansammelt, der für Ihr Programm (oder das System) nicht verfügbar ist. Dies kann zu Abstürzen oder undefiniertem Verhalten führen. Probleme treten häufiger auf, wenn das Leck wiederholt in einer Schleife oder rekursiven Funktion auftritt. Das Risiko eines Programmfehlers steigt, je länger ein Programmleck läuft. Manchmal treten Probleme sofort auf; Zu anderen Zeiten werden Probleme für Stunden oder sogar Jahre des Dauerbetriebs nicht gesehen. Ausfälle der Speichermüdung können je nach den Umständen katastrophal sein.

Die folgende Endlosschleife ist ein Beispiel für ein Leck, das schließlich den verfügbaren Speicherverlust erschöpft, indem `getline()` , eine Funktion, die implizit neuen Speicher `getline()` , ohne diesen Speicher `getline()` .

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *line = NULL;
    size_t size = 0;
```

```
    /* The loop below leaks memory as fast as it can */

    for(;;) {
        getline(&line, &size, stdin); /* New memory implicitly allocated */

        /* <do whatever> */

        line = NULL;
    }

    return 0;
}
```

Im Gegensatz dazu wird im folgenden Code auch die Funktion `getline()` verwendet. Diesmal wird der zugewiesene Speicher jedoch ordnungsgemäß freigegeben, wodurch ein Leck vermieden wird.

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *line = NULL;
    size_t size = 0;

    for(;;) {
        if (getline(&line, &size, stdin) < 0) {
            free(line);
            line = NULL;

            /* Handle failure such as setting flag, breaking out of loop and/or exiting */
        }

        /* <do whatever> */

        free(line);
        line = NULL;

    }

    return 0;
}
```

Undichtes Gedächtnis hat nicht immer greifbare Konsequenzen und ist nicht notwendigerweise ein funktionelles Problem. Während "Best Practice" rigoros den Speicher an strategischen Punkten und Bedingungen freigibt, um den Speicherbedarf des Speichers zu verringern und das Risiko der Speicherplatzermüdung zu senken, kann es Ausnahmen geben. Wenn ein Programm beispielsweise in Dauer und Umfang begrenzt ist, kann das Risiko eines Fehlschlags der Zuordnung als zu gering angesehen werden, um sich darüber Sorgen zu machen. In diesem Fall kann die Umgehung einer expliziten Freigabe als akzeptabel betrachtet werden. Beispielsweise geben die meisten modernen Betriebssysteme automatisch den gesamten von einem Programm belegten Speicherplatz frei, wenn es beendet wird, sei es aufgrund eines Programmfehlers, eines Systemaufrufs `exit()` , der Prozessbeendigung oder des Endes von `main()` . Das explizite Freigeben von Speicher zum Zeitpunkt der bevorstehenden Programmbeendigung könnte

tatsächlich redundant sein oder einen Performance-Nachteil mit sich bringen.

Die Zuordnung kann fehlschlagen, wenn nicht genügend Speicher verfügbar ist. Die Behandlung von Fehlern sollte auf den entsprechenden Ebenen des Aufrufstapels berücksichtigt werden. `getline()` oben gezeigte `getline()` ist ein interessanter Anwendungsfall, da es sich um eine Bibliotheksfunktion handelt, die nicht nur dem Anrufer `getline()` , sondern auch aus verschiedenen Gründen fehlschlagen kann, die alle berücksichtigt werden müssen. Daher ist es bei der Verwendung einer C-API unbedingt erforderlich, die *Dokumentation (Manpage)* zu lesen und den Fehlerbedingungen und der Speicherauslastung besondere Aufmerksamkeit zu widmen und zu wissen, welche Softwareschicht die Last trägt, die zurückgegebener Speicher freizugeben.

Eine andere übliche Vorgehensweise bei der Speicherbehandlung besteht darin, die Speicherzeiger gleich nach dem Freigeben des von diesen Zeigern referenzierten Speichers auf NULL zu setzen, sodass diese Zeiger jederzeit auf Gültigkeit getestet werden können (z. B. auf NULL / Nicht-NULL geprüft), weil auf den freigegebenen Speicher zugegriffen wird kann zu schwerwiegenden Problemen führen, beispielsweise zum Abrufen von Mülldaten (Lesevorgang) oder Datenverfälschung (Schreibvorgang) und / oder einem Programmabsturz. In den meisten modernen Betriebssystemen ist das Freigeben von Speicherplatz 0 (`NULL`) ein NOP (z. B. harmlos), wie vom C-Standard gefordert. Wenn Sie also einen Zeiger auf NULL setzen, besteht kein Risiko, dass der Speicher doppelt freigegeben wird, wenn der Zeiger wird an `free()` . Denken Sie daran, dass das doppelte Freigeben des Speichers zu sehr zeitaufwändigen, verwirrenden und *schwer zu diagnostizierenden* Fehlern führen kann.

Zu viel kopieren

```
char buf[8]; /* tiny buffer, easy to overflow */

printf("What is your name?\n");
scanf("%s", buf); /* WRONG */
scanf("%7s", buf); /* RIGHT */
```

Wenn der Benutzer eine Zeichenfolge eingibt, die länger als 7 Zeichen ist (`buf` für den `buf`), wird der Speicher hinter dem `buf` überschrieben. Dies führt zu undefiniertem Verhalten. Schädliche Hacker nutzen dies häufig aus, um die Absenderadresse zu überschreiben und in die Adresse des bösartigen Codes des Hackers zu ändern.

Vergessen, den Rückgabewert von realloc in einen temporären Wert zu kopieren

Wenn `realloc` fehlschlägt, wird `NULL` . Wenn Sie den Wert des ursprünglichen Puffers dem Rückgabewert von `realloc` zuweisen und `NULL` , geht der ursprüngliche Puffer (der alte Zeiger) verloren und führt zu einem *Speicherverlust* . Die Lösung besteht darin, in einen temporären Zeiger zu kopieren, und wenn dieser temporär nicht NULL ist, **dann** in den realen Puffer kopieren.

```
char *buf, *tmp;

buf = malloc(...);
...
```

```
/* WRONG */
if ((buf = realloc(buf, 16)) == NULL)
    perror("realloc");

/* RIGHT */
if ((tmp = realloc(buf, 16)) != NULL)
    buf = tmp;
else
    perror("realloc");
```

Vergleich von Fließkommazahlen

Gleitkommatypen (`float` , `double` und `long double`) können einige Zahlen nicht genau darstellen, da sie eine endliche Genauigkeit haben und die Werte in einem binären Format darstellen. Genauso wie wir Dezimalstellen in der Basis 10 für Brüche wie 1/3 wiederholen, gibt es Brüche, die nicht auch binär endlich dargestellt werden können (wie 1/3, aber noch wichtiger 1/10). Vergleichen Sie Gleitkommawerte nicht direkt. Verwenden Sie stattdessen ein Delta.

```
#include <float.h> // for DBL_EPSILON and FLT_EPSILON
#include <math.h>  // for fabs()

int main(void)
{
    double a = 0.1; // imprecise: (binary) 0.000110...

    // may be false or true
    if (a + a + a + a + a + a + a + a + a + a == 1.0) {
        printf("10 * 0.1 is indeed 1.0. This is not guaranteed in the general case.\n");
    }

    // Using a small delta value.
    if (fabs(a + a + a + a + a + a + a + a + a + a - 1.0) < 0.000001) {
        // C99 5.2.4.2.2p8 guarantees at least 10 decimal digits
        // of precision for the double type.
        printf("10 * 0.1 is almost 1.0.\n");
    }

    return 0;
}
```

Ein anderes Beispiel:

```
gcc -O3   -g   -I./inc   -std=c11   -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-
prototypes  -Wold-style-definition       rd11.c -o rd11 -L./lib -lsoq
#include <stdio.h>
#include <math.h>

static inline double rel_diff(double a, double b)
{
    return fabs(a - b) / fmax(fabs(a), fabs(b));
}

int main(void)
{
    double d1 = 3.14159265358979;
```

```
    double d2 = 355.0 / 113.0;

    double epsilon = 1.0;
    for (int i = 0; i < 10; i++)
    {
        if (rel_diff(d1, d2) < epsilon)
            printf("%d:%.10f <=> %.10f within tolerance %.10f (rel diff %.4E)\n",
                   i, d1, d2, epsilon, rel_diff(d1, d2));
        else
            printf("%d:%.10f <=> %.10f out of tolerance %.10f (rel diff %.4E)\n",
                   i, d1, d2, epsilon, rel_diff(d1, d2));
        epsilon /= 10.0;
    }
    return 0;
}
```

Ausgabe:

```
0:3.1415926536 <=> 3.1415929204 within tolerance 1.0000000000 (rel diff 8.4914E-08)
1:3.1415926536 <=> 3.1415929204 within tolerance 0.1000000000 (rel diff 8.4914E-08)
2:3.1415926536 <=> 3.1415929204 within tolerance 0.0100000000 (rel diff 8.4914E-08)
3:3.1415926536 <=> 3.1415929204 within tolerance 0.0010000000 (rel diff 8.4914E-08)
4:3.1415926536 <=> 3.1415929204 within tolerance 0.0001000000 (rel diff 8.4914E-08)
5:3.1415926536 <=> 3.1415929204 within tolerance 0.0000100000 (rel diff 8.4914E-08)
6:3.1415926536 <=> 3.1415929204 within tolerance 0.0000010000 (rel diff 8.4914E-08)
7:3.1415926536 <=> 3.1415929204 within tolerance 0.0000001000 (rel diff 8.4914E-08)
8:3.1415926536 <=> 3.1415929204 out of tolerance 0.0000000100 (rel diff 8.4914E-08)
9:3.1415926536 <=> 3.1415929204 out of tolerance 0.0000000010 (rel diff 8.4914E-08)
```

Zusätzliche Skalierung in der Zeigerarithmetik

Bei der Zeigerarithmetik wird die Ganzzahl, die dem Zeiger hinzugefügt oder davon abgezogen werden soll, nicht als Änderung der *Adresse,* sondern als Anzahl der zu verschiebenden *Elemente* interpretiert.

```
#include <stdio.h>

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = ptr + sizeof(int) * 2; /* wrong */
    printf("%d %d\n", *ptr, *ptr2);
    return 0;
}
```

Dieser Code führt eine zusätzliche Skalierung bei der Berechnung des Zeigers durch, der `ptr2` zugewiesen `ptr2` . Wenn `sizeof(int)` 4 ist, was in modernen 32-Bit-Umgebungen typisch ist, steht der Ausdruck für "8 elements after `array[0]` " (außerhalb des Bereichs) und ruft ein *undefiniertes Verhalten auf* .

Damit `ptr2` auf 2 Elemente nach `array[0]` , sollten Sie einfach 2 hinzufügen.

```
#include <stdio.h>
```

```
int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = ptr + 2;
    printf("%d %d\n", *ptr, *ptr2); /* "1 3" will be printed */
    return 0;
}
```

Die explizite Zeigerarithmetik mit additiven Operatoren kann verwirrend sein, sodass die Verwendung von Array-Subskriptionen möglicherweise besser ist.

```
#include <stdio.h>

int main(void) {
    int array[] = {1, 2, 3, 4, 5};
    int *ptr = &array[0];
    int *ptr2 = &ptr[2];
    printf("%d %d\n", *ptr, *ptr2); /* "1 3" will be printed */
    return 0;
}
```

`E1[E2]` ist identisch mit `(*((E1)+(E2)))` (N1570 6.5.2.1, Absatz 2) und `&(E1[E2])` entspricht `((E1)+(E2))` (N1570 6.5.3.2, Fußnote 102).

Wenn die Zeigerarithmetik bevorzugt wird, kann das Umsetzen des Zeigers auf die Adresse eines anderen Datentyps eine Byte-Adressierung ermöglichen. Seien Sie jedoch vorsichtig: Endianness kann zu einem Problem werden, und das Umschalten auf andere Typen als "Zeiger auf Zeichen" führt zu strengen Aliasing-Problemen .

```
#include <stdio.h>

int main(void) {
    int array[3] = {1,2,3};  // 4 bytes * 3 allocated
    unsigned char *ptr = (unsigned char *) array;  // unsigned chars only take 1 byte
    /*
     * Now any pointer arithmetic on ptr will match
     * bytes in memory.  ptr can be treated like it
     * was declared as: unsigned char ptr[12];
     */

    return 0;
}
```

Makros sind einfache Zeichenfolgen

Makros sind einfache Zeichenfolgen. (Streng genommen arbeiten sie mit Vorverarbeitungsmarkern, nicht mit beliebigen Zeichenketten.)

```
#include <stdio.h>

#define SQUARE(x) x*x

int main(void) {
    printf("%d\n", SQUARE(1+2));
```

```
    return 0;
}
```

Sie können erwarten, dass dieser Code `9` (`3*3`) druckt, aber tatsächlich werden `5` gedruckt, da das Makro auf `1+2*1+2` .

Sie sollten die Argumente und den gesamten Makroausdruck in Klammern einschließen, um dieses Problem zu vermeiden.

```
#include <stdio.h>

#define SQUARE(x) ((x)*(x))

int main(void) {
    printf("%d\n", SQUARE(1+2));
    return 0;
}
```

Ein weiteres Problem besteht darin, dass die Argumente eines Makros nicht garantiert werden, einmalig ausgewertet zu werden. Sie werden möglicherweise überhaupt nicht oder mehrmals bewertet.

```
#include <stdio.h>

#define MIN(x, y) ((x) <= (y) ? (x) : (y))

int main(void) {
    int a = 0;
    printf("%d\n", MIN(a++, 10));
    printf("a = %d\n", a);
    return 0;
}
```

In diesem Code wird das Makro zu `((a++) <= (10) ? (a++) : (10))` . Da `a++` (`0`) kleiner als `10` , wird `a++` zweimal ausgewertet und der Wert von `a` wird davon abweichen, was von `MIN` .

Dies kann durch die Verwendung von Funktionen vermieden werden. Beachten Sie jedoch, dass die Typen durch die Funktionsdefinition festgelegt werden, während Makros bei Typen (zu) flexibel sein können.

```
#include <stdio.h>

int min(int x, int y) {
    return x <= y ? x : y;
}

int main(void) {
    int a = 0;
    printf("%d\n", min(a++, 10));
    printf("a = %d\n", a);
    return 0;
}
```

Das Problem der Doppelbewertung ist nun behoben, aber diese `min` Funktion kann beispielsweise

keine `double` verarbeiten, ohne sie zu kürzen.

Es gibt zwei Arten von Makroanweisungen:

```
#define OBJECT_LIKE_MACRO     followed by a "replacement list" of preprocessor tokens
#define FUNCTION_LIKE_MACRO(with, arguments) followed by a replacement list
```

Diese beiden Arten von Makros unterscheiden sich durch das Zeichen, das dem Bezeichner nach `#define` folgt: Wenn es sich um ein *lparen-Objekt handelt* , handelt es sich um ein funktionsähnliches Makro. Andernfalls handelt es sich um ein objektartiges Makro. Wenn die Absicht ist es, eine funktionsähnliche Makros zu schreiben, muss es kein Leerraum sein zwischen dem Ende des Namen des Makros und `(` . Überprüfen Sie dies für eine ausführliche Erklärung.

C99

In C99 oder später könnten Sie `static inline int min(int x, int y) { … }` .

C11

In C11 können Sie einen 'typgenerischen' Ausdruck für `min` .

```
#include <stdio.h>

#define min(x, y) _Generic((x), \
                        long double: min_ld, \
                        unsigned long long: min_ull, \
                        default: min_i \
                        )(x, y)

#define gen_min(suffix, type) \
    static inline type min_##suffix(type x, type y) { return (x < y) ? x : y; }

gen_min(ld, long double)
gen_min(ull, unsigned long long)
gen_min(i, int)

int main(void)
{
    unsigned long long ull1 = 50ULL;
    unsigned long long ull2 = 37ULL;
    printf("min(%llu, %llu) = %llu\n", ull1, ull2, min(ull1, ull2));
    long double ld1 = 3.141592653L;
    long double ld2 = 3.141592652L;
    printf("min(%.10Lf, %.10Lf) = %.10Lf\n", ld1, ld2, min(ld1, ld2));
    int i1 = 3141653;
    int i2 = 3141652;
    printf("min(%d, %d) = %d\n", i1, i2, min(i1, i2));
    return 0;
}
```

Der generische Ausdruck könnte um weitere Typen erweitert werden, z. B. `double` , `float` , `long long` , `unsigned long` , `long` , `unsigned` - und entsprechende Makroaufrufe in `gen_min` .

Undefinierte Referenzfehler beim Linken

Einer der häufigsten Fehler beim Kompilieren tritt während der Verbindungsphase auf. Der Fehler sieht ähnlich aus:

```
$ gcc undefined_reference.c
/tmp/ccoXhwF0.o: In function `main':
undefined_reference.c:(.text+0x15): undefined reference to `foo'
collect2: error: ld returned 1 exit status
$
```

Schauen wir uns den Code an, der diesen Fehler generiert hat:

```
int foo(void);

int main(int argc, char **argv)
{
    int foo_val;
    foo_val = foo();
    return foo_val;
}
```

Wir sehen hier eine *Deklaration* von foo (`int foo();`), aber keine *Definition* davon (tatsächliche Funktion). Also haben wir dem Compiler den Funktionsheader zur Verfügung gestellt, aber es wurde an keiner Stelle eine solche Funktion definiert, so dass die Kompilierungsphase durchläuft, der Linker jedoch mit einem `Undefined reference` wird.
Um diesen Fehler in unserem kleinen Programm zu beheben, müssen wir nur eine *Definition* für foo hinzufügen:

```
/* Declaration of foo */
int foo(void);

/* Definition of foo */
int foo(void)
{
    return 5;
}

int main(int argc, char **argv)
{
    int foo_val;
    foo_val = foo();
    return foo_val;
}
```

Nun wird dieser Code kompiliert. Eine alternative Situation tritt auf, wenn sich die Quelle für `foo()` in einer separaten Quelldatei `foo.c` (und es gibt einen Header `foo.h` , der `foo()` deklariert, der in `foo.c` und `undefined_reference.c`). Das `foo.c` besteht dann darin, sowohl die Objektdatei aus `foo.c` als auch `undefined_reference.c` zu verknüpfen oder beide Quelldateien zu kompilieren:

```
$ gcc -c undefined_reference.c
$ gcc -c foo.c
$ gcc -o working_program undefined_reference.o foo.o
$
```

Oder:

```
$ gcc -o working_program undefined_reference.c foo.c
$
```

In einem komplexeren Fall handelt es sich um Bibliotheken wie im Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char **argv)
{
    double first;
    double second;
    double power;

    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <denom> <nom>\n", argv[0]);
        return EXIT_FAILURE;
    }

    /* Translate user input to numbers, extra error checking
     * should be done here. */
    first = strtod(argv[1], NULL);
    second = strtod(argv[2], NULL);

    /* Use function pow() from libm - this will cause a linkage
     * error unless this code is compiled against libm! */
    power = pow(first, second);

    printf("%f to the power of %f = %f\n", first, second, power);

    return EXIT_SUCCESS;
}
```

Der Code ist syntaktisch korrekt. Deklaration für `pow()` existiert von `#include <math.h>` . Wir versuchen also zu kompilieren und zu verknüpfen, erhalten jedoch einen Fehler wie `#include <math.h>` :

```
$ gcc no_library_in_link.c -o no_library_in_link
/tmp/ccduQQqA.o: In function `main':
no_library_in_link.c:(.text+0x8b): undefined reference to `pow'
collect2: error: ld returned 1 exit status
$
```

Dies geschieht, weil die *Definition* für `pow()` während der Verbindungsphase nicht gefunden wurde. Um dies zu beheben, müssen wir angeben, dass wir eine Verknüpfung mit der Mathematikbibliothek namens `libm` indem Sie das Flag `-lm` . (Beachten Sie, dass es Plattformen wie macOS gibt, auf denen `-lm` nicht benötigt wird, aber wenn Sie die undefinierte Referenz erhalten, wird die Bibliothek benötigt.)

Deshalb führen wir die Kompilierungsphase erneut aus, wobei wir die Bibliothek (nach den Quell- oder Objektdateien) angeben:

```
$ gcc no_library_in_link.c -lm -o library_in_link_cmd
$ ./library_in_link_cmd 2 4
2.000000 to the power of 4.000000 = 16.000000
$
```

Und es funktioniert!

Missverständnis des Array-Zerfalls

Ein häufiges Problem in Code, der mehrdimensionale Arrays, Arrays von Zeigern usw. verwendet, ist die Tatsache, dass `Type**` und `Type[M][N]` grundsätzlich verschiedene Typen sind:

```
#include <stdio.h>

void print_strings(char **strings, size_t n)
{
    size_t i;
    for (i = 0; i < n; i++)
        puts(strings[i]);
}

int main(void)
{
    char s[4][20] = {"Example 1", "Example 2", "Example 3", "Example 4"};
    print_strings(s, 4);
    return 0;
}
```

Beispiel Compiler-Ausgabe:

```
file1.c: In function 'main':
file1.c:13:23: error: passing argument 1 of 'print_strings' from incompatible pointer type [-
Wincompatible-pointer-types]
         print_strings(strings, 4);
                       ^
file1.c:3:10: note: expected 'char **' but argument is of type 'char (*)[20]'
     void print_strings(char **strings, size_t n)
```

Der Fehler gibt an, dass das Array `s` in der `main` an die Funktion `print_strings` , die einen anderen `print_strings` erwartet als erwartet. Es enthält auch eine Notiz , die Art zum Ausdruck , die von erwartet wird `print_strings` und die Art, die er von geben wurde `main` .

Das Problem beruht auf einem sogenannten *Array-Zerfall* . Was passiert, wenn `s` mit seinem Typ `char[4][20]` (Array von 4 Arrays mit 20 Zeichen) an die Funktion übergeben wird, wenn sie zu einem Zeiger auf das erste Element wird, als hätten Sie `&s[0]` geschrieben der Typ `char (*)[20]` (Zeiger auf 1 Array mit 20 Zeichen). Dies tritt für jedes Array auf, einschließlich eines Arrays von Zeigern, eines Arrays von Arrays (3-D-Arrays) und eines Arrays von Zeigern auf ein Array. Die folgende Tabelle zeigt, was passiert, wenn ein Array zerfällt. Änderungen in der Typbeschreibung werden hervorgehoben, um zu zeigen, was passiert:

Vor dem Verfall		Nach dem Verfall	
char [20]	**Array von (20 Zeichen)**	char *	**Zeiger auf (1 Zeichen)**
char [4][20]	**Array von (4 Arrays** mit 20 Zeichen)	char (*)[20]	**Zeiger auf (1 Array** mit 20 Zeichen)
char *[4]	**Array von (4 Zeiger** auf 1 Zeichen)	char **	**Zeiger auf (1 Zeiger** auf 1 Zeichen)
char [3][4][20]	**Array von (3 Arrays** von 4 Arrays mit 20 Zeichen)	char (*)[4][20]	**Zeiger auf (1 Array** mit 4 Arrays mit 20 Zeichen)
char (*[4])[20]	**Array von (4 Zeiger** auf 1 Array von 20 Zeichen)	char (**)[20]	**Zeiger auf (1 Zeiger** auf 1 Array mit 20 Zeichen)

Wenn ein Array zu einem Zeiger zerfallen kann, kann gesagt werden, dass ein Zeiger als Array mit mindestens einem Element betrachtet werden kann. Eine Ausnahme bildet ein Nullzeiger, der auf nichts zeigt und somit kein Array ist.

Array-Zerfall tritt nur einmal auf. Wenn ein Array zu einem Zeiger zerfallen ist, ist es jetzt ein Zeiger, kein Array. Wenn Sie einen Zeiger auf ein Array haben, denken Sie daran, dass der Zeiger als Array mit mindestens einem Element betrachtet werden kann, sodass bereits ein Zerfall des Arrays aufgetreten ist.

Mit anderen Worten, ein Zeiger auf ein Array (`char (*)[20]`) wird niemals ein Zeiger auf einen Zeiger (`char **`). Um die `print_strings` Funktion zu `print_strings` , lassen Sie sie einfach den richtigen Typ erhalten:

```
void print_strings(char (*strings)[20], size_t n)
/* OR */
void print_strings(char strings[][20], size_t n)
```

Ein Problem tritt auf, wenn Sie möchten, dass die Funktion `print_strings` für ein beliebiges Array von Zeichen generisch ist: Was ist, wenn 30 Zeichen statt 20 Zeichen sind? Oder 50? Die Antwort besteht darin, vor dem Array-Parameter einen weiteren Parameter hinzuzufügen:

```
#include <stdio.h>

/*
 * Note the rearranged parameters and the change in the parameter name
 * from the previous definitions:
 *      n (number of strings)
 *   => scount (string count)
 *
 * Of course, you could also use one of the following highly recommended forms
 * for the `strings` parameter instead:
 *
 *    char strings[scount][ccount]
 *    char strings[][ccount]
 */
```

```
void print_strings(size_t scount, size_t ccount, char (*strings)[ccount])
{
    size_t i;
    for (i = 0; i < scount; i++)
        puts(strings[i]);
}

int main(void)
{
    char s[4][20] = {"Example 1", "Example 2", "Example 3", "Example 4"};
    print_strings(4, 20, s);
    return 0;
}
```

Das Kompilieren erzeugt keine Fehler und führt zu der erwarteten Ausgabe:

```
Example 1
Example 2
Example 3
Example 4
```

Übergeben nicht benachbarter Arrays an Funktionen, die "echte" mehrdimensionale Arrays erwarten

Wenn mehrdimensionalen Arrays mit Zuweisung `malloc` , `calloc` und `realloc` , ist ein gemeinsames Muster der inneren Arrays mit mehreren Anrufen zuzuweisen (auch wenn der Anruf nur einmal vorkommt, kann es in einer Schleife sein kann):

```
/* Could also be `int **` with malloc used to allocate outer array. */
int *array[4];
int i;

/* Allocate 4 arrays of 16 ints. */
for (i = 0; i < 4; i++)
    array[i] = malloc(16 * sizeof(*array[i]));
```

Die Bytedifferenz zwischen dem letzten Element eines der inneren Arrays und dem ersten Element des nächsten inneren Arrays darf nicht 0 sein, wie dies bei einem "echten" multidimensionalen Array der Fall wäre (z. B. `int array[4][16];`) :

```
/* 0x40003c, 0x402000 */
printf("%p, %p\n", (void *)(array[0] + 15), (void *)array[1]);
```

Berücksichtigt man die Größe von `int` , ergibt sich ein Unterschied von 8128 Bytes (8132-4), was 2032 `int` dimensionierten Array-Elementen entspricht, und das ist das Problem: Ein "reales" multidimensionales Array hat keine Lücken zwischen den Elementen.

Wenn Sie ein dynamisch zugewiesenes Array mit einer Funktion verwenden müssen, die ein "reales" mehrdimensionales Array erwartet, sollten Sie ein Objekt vom Typ `int *` zuordnen und zur Durchführung von Berechnungen Arithmetik verwenden:

```
void func(int M, int N, int *array);
```

```
...

/* Equivalent to declaring `int array[M][N] = {{0}};` and assigning to array4_16[i][j]. */
int *array;
int M = 4, N = 16;
array = calloc(M, N * sizeof(*array));
array[i * N + j] = 1;
func(M, N, array);
```

Wenn `N` ein Makro oder ein Integer-Literal ist und keine Variable, kann der Code einfach die natürlichere 2D-Array-Notation verwenden, nachdem ein Zeiger einem Array zugewiesen wurde:

```
void func(int M, int N, int *array);
#define N 16
void func_N(int M, int (*array)[N]);
...

int M = 4;
int (*array)[N];
array = calloc(M, sizeof(*array));
array[i][j] = 1;

/* Cast to `int *` works here because `array` is a single block of M*N ints with no gaps,
   just like `int array2[M * N];` and `int array3[M][N];` would be. */
func(M, N, (int *)array);
func_N(M, array);
```

C99

Wenn `N` kein Makro oder Integer-Literal ist, zeigt das `array` auf ein `array` mit variabler Länge (VLA). Dies kann immer noch mit `func` werden, indem in `int *` und eine neue Funktion `func_vla` würde `func_N` ersetzen:

```
void func(int M, int N, int *array);
void func_vla(int M, int N, int array[M][N]);
...

int M = 4, N = 16;
int (*array)[N];
array = calloc(M, sizeof(*array));
array[i][j] = 1;
func(M, N, (int *)array);
func_vla(M, N, array);
```

C11

Hinweis : VLAs sind ab C11 optional. Wenn Ihre Implementierung C11 unterstützt und das Makro `__STDC_NO_VLA__` auf 1 definiert, bleiben Sie bei den pre-C99-Methoden.

Verwenden Sie Zeichenkonstanten anstelle von String-Literalen und umgekehrt

In C sind Zeichenkonstanten und Zeichenkettenliterale verschieden.

Ein Zeichen, das von einfachen Anführungszeichen wie `'a'` ist `'a'` ist eine *Zeichenkonstante* . Eine Zeichenkonstante ist eine ganze Zahl, deren Wert der Zeichencode ist, der für das Zeichen steht. Die Interpretation von Zeichenkonstanten mit mehreren Zeichen wie `'abc'` ist von der Implementierung definiert.

Null oder mehr Zeichen, die von Anführungszeichen wie `"abc"` sind, sind ein *String-Literal* . Ein String-Literal ist ein nicht veränderbares Array, dessen Elemente vom Typ `char` . Die Zeichenfolge in doppelten Anführungszeichen sowie das abschließende Nullzeichen sind die Inhalte. `"abc"` hat also 4 Elemente (`{'a', 'b', 'c', '\0'}`).

In diesem Beispiel wird eine Zeichenkonstante verwendet, bei der ein Zeichenfolgenliteral verwendet werden soll. Diese Zeichenkonstante wird auf implementierungsdefinierte Weise in einen Zeiger konvertiert. Es besteht nur eine geringe Möglichkeit, dass der konvertierte Zeiger gültig ist. In diesem Beispiel wird daher *undefiniertes Verhalten* aufgerufen.

```
#include <stdio.h>

int main(void) {
    const char *hello = 'hello, world'; /* bad */
    puts(hello);
    return 0;
}
```

In diesem Beispiel wird ein String-Literal verwendet, bei dem eine Zeichenkonstante verwendet werden soll. Der aus dem String-Literal konvertierte Zeiger wird in einer implementierungsdefinierten Weise in eine Ganzzahl und in einer implementierungsdefinierten Art in `char` konvertiert. (Wie eine Ganzzahl in einen vorzeichenbehafteten Typ konvertiert wird, der den umzuwandelnden Wert nicht darstellen kann, ist implementierungsdefiniert, und ob `char` signiert ist, ist ebenfalls implementierungsdefiniert.)

```
#include <stdio.h>

int main(void) {
    char c = "a"; /* bad */
    printf("%c\n", c);
    return 0;
}
```

In fast allen Fällen wird der Compiler über diese Verwechslungen klagen. Ist dies nicht der Fall, müssen Sie weitere Warnmeldungen für den Compiler verwenden oder es wird empfohlen, einen besseren Compiler zu verwenden.

Rückgabewerte von Bibliotheksfunktionen ignorieren

Fast jede Funktion in der C-Standardbibliothek gibt bei Erfolg etwas und bei Fehlern etwas anderes zurück. `malloc` gibt beispielsweise bei Erfolg einen Zeiger auf den von der Funktion zugewiesenen Speicherblock zurück und, falls die Funktion den angeforderten Speicherblock nicht zuordnen konnte, einen Nullzeiger. Sie sollten daher immer den Rückgabewert überprüfen, um das Debuggen zu erleichtern.

Das ist schlecht:

```
char* x = malloc(100000000000UL * sizeof *x);
/* more code */
scanf("%s", x); /* This might invoke undefined behaviour and if lucky causes a segmentation
violation, unless your system has a lot of memory */
```

Das ist gut:

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char* x = malloc(100000000000UL * sizeof *x);
    if (x == NULL) {
        perror("malloc() failed");
        exit(EXIT_FAILURE);
    }

    if (scanf("%s", x) != 1) {
        fprintf(stderr, "could not read string\n");
        free(x);
        exit(EXIT_FAILURE);
    }

    /* Do stuff with x. */

    /* Clean up. */
    free(x);

    return EXIT_SUCCESS;
}
```

Auf diese Weise wissen Sie sofort die Fehlerursache, ansonsten könnten Sie stundenlang an einem völlig falschen Ort nach einem Fehler suchen.

Bei einem typischen Aufruf von scanf () wird kein Zeilenvorschubzeichen verwendet

Wenn dieses Programm

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char str[128], *lf;

    scanf("%d", &num);
    fgets(str, sizeof(str), stdin);

    if ((lf = strchr(str, '\n')) != NULL) *lf = '\0';
    printf("%d \"%s\"\n", num, str);
    return 0;
}
```

wird mit dieser Eingabe ausgeführt

```
42
life
```

Die Ausgabe wird `42 ""` anstelle der erwarteten `42 "life"` .

Dies liegt daran, dass nach dem Aufruf von `scanf()` kein Newline-Zeichen nach `42` und `fgets()` vor dem Lesen der `life` . Dann `fgets()` auf zu lesen, bevor es das `life` liest.

Um dieses Problem zu vermeiden, ist eine Methode, die nützlich ist, wenn die maximale Länge einer Zeile bekannt ist, beispielsweise bei der Lösung von Problemen im Online-Richtersystem, die `scanf()` Verwendung von `scanf()` und das Lesen aller Zeilen über `fgets()` . Sie können `sscanf()` , um die gelesenen Zeilen zu analysieren.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char line_buffer[128] = "", str[128], *lf;

    fgets(line_buffer, sizeof(line_buffer), stdin);
    sscanf(line_buffer, "%d", &num);
    fgets(str, sizeof(str), stdin);

    if ((lf = strchr(str, '\n')) != NULL) *lf = '\0';
    printf("%d \"%s\"\n", num, str);
    return 0;
}
```

Eine andere Möglichkeit besteht darin, zu lesen, bis Sie nach der Verwendung von `scanf()` und vor der Verwendung von `fgets()` ein Zeilenumbruchzeichen `fgets()` .

```
#include <stdio.h>
#include <string.h>

int main(void) {
    int num = 0;
    char str[128], *lf;
    int c;

    scanf("%d", &num);
    while ((c = getchar()) != '\n' && c != EOF);
    fgets(str, sizeof(str), stdin);

    if ((lf = strchr(str, '\n')) != NULL) *lf = '\0';
    printf("%d \"%s\"\n", num, str);
    return 0;
}
```

Hinzufügen eines Semikolons zu einer # Definition

Es ist leicht, sich im C-Präprozessor zu verwirren und als Teil von C selbst zu behandeln. Dies ist

jedoch ein Fehler, da der Präprozessor nur ein Textsubstitutionsmechanismus ist. Zum Beispiel, wenn Sie schreiben

```
/* WRONG */
#define MAX 100;
int arr[MAX];
```

der Code erweitert sich zu

```
int arr[100;];
```

Das ist ein Syntaxfehler. Das Hilfsmittel besteht darin, das Semikolon aus der Zeile `#define` zu entfernen. Es ist fast immer ein Fehler, ein `#define` mit einem Semikolon zu beenden.

Mehrzeilige Kommentare können nicht geschachtelt werden

In C schachteln mehrzeilige Kommentare / * und * / nicht.

Wenn Sie einen Codeblock oder eine Funktion mit diesem Kommentar kommentieren:

```
/*
 * max(): Finds the largest integer in an array and returns it.
 * If the array length is less than 1, the result is undefined.
 * arr: The array of integers to search.
 * num: The number of integers in arr.
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
```

Sie werden es nicht leicht kommentieren können:

```
//Trying to comment out the block...
/*

/*
 * max(): Finds the largest integer in an array and returns it.
 * If the array length is less than 1, the result is undefined.
 * arr: The array of integers to search.
 * num: The number of integers in arr.
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
```

```
//Causes an error on the line below...
*/
```

Eine Lösung ist die Verwendung von Kommentaren im C99-Stil:

```
// max(): Finds the largest integer in an array and returns it.
// If the array length is less than 1, the result is undefined.
// arr: The array of integers to search.
// num: The number of integers in arr.
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
```

Nun kann der gesamte Block einfach auskommentiert werden:

```
/*

// max(): Finds the largest integer in an array and returns it.
// If the array length is less than 1, the result is undefined.
// arr: The array of integers to search.
// num: The number of integers in arr.
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

*/
```

Eine andere Lösung besteht darin, das Deaktivieren von Code mithilfe der Kommentarsyntax zu vermeiden, stattdessen die Direktive `#ifdef` oder `#ifndef` . Diese Richtlinien *tun* Nest, so dass Sie frei Ihren Code im Stil Kommentar Sie bevorzugen.

```
#define DISABLE_MAX /* Remove or comment this line to enable max() code block */

#ifdef DISABLE_MAX
/*
 * max(): Finds the largest integer in an array and returns it.
 * If the array length is less than 1, the result is undefined.
 * arr: The array of integers to search.
 * num: The number of integers in arr.
 */
int max(int arr[], int num)
{
    int max = arr[0];
    for (int i = 0; i < num; i++)
        if (arr[i] > max)
            max = arr[i];
```

```
    return max;
}
#endif
```

Einige Handbücher gehen so weit, zu empfehlen, dass Codeabschnitte *niemals* kommentiert werden dürfen. Wenn Code vorübergehend deaktiviert werden soll, kann auf eine `#if 0` Direktive zurückgegriffen werden.

Siehe #if 0, um Codeabschnitte auszublenden .

Array-Grenzen überschreiten

Arrays sind nullbasiert, das heißt, der Index beginnt immer bei 0 und endet mit der Länge des Index-Arrays minus 1. Der folgende Code gibt also nicht das erste Element des Arrays aus und gibt für den endgültigen Wert, den er druckt, eine Mülltonne aus.

```
#include <stdio.h>

int main(void)
{
    int x = 0;
    int myArray[5] = {1, 2, 3, 4, 5}; //Declaring 5 elements

    for(x = 1; x <= 5; x++) //Looping from 1 till 5.
       printf("%d\t", myArray[x]);

    printf("\n");
    return 0;
}
```

Ausgabe: `2 3 4 5 GarbageValue`

Im Folgenden wird der richtige Weg beschrieben, um die gewünschte Leistung zu erzielen:

```
#include <stdio.h>

int main(void)
{
    int x = 0;
    int myArray[5] = {1, 2, 3, 4, 5}; //Declaring 5 elements

    for(x = 0; x < 5; x++) //Looping from 0 till 4.
       printf("%d\t", myArray[x]);

    printf("\n");
    return 0;
}
```

Ausgabe: `1 2 3 4 5`

Es ist wichtig, die Länge eines Arrays zu kennen, bevor Sie mit dem Array arbeiten. Andernfalls können Sie den Puffer beschädigen oder einen Segmentierungsfehler verursachen, indem Sie auf Speicherbereiche zugreifen, die außerhalb der Grenzen liegen.

Rekursive Funktion - Die Basisbedingung wird übersehen

Die Berechnung der Fakultät einer Zahl ist ein klassisches Beispiel für eine rekursive Funktion.

Fehlende Grundbedingung:

```
#include <stdio.h>

int factorial(int n)
{
       return n * factorial(n - 1);
}

int main()
{
    printf("Factorial %d = %d\n", 3, factorial(3));
    return 0;
}
```

Typischer Ausgang: `Segmentation fault: 11`

Das Problem bei dieser Funktion ist, dass sie endlos in einer Schleife abläuft und einen Segmentierungsfehler verursacht. Sie benötigt eine Basisbedingung, um die Rekursion zu stoppen.

Grundbedingung angegeben:

```
#include <stdio.h>

int factorial(int n)
{
    if (n == 1) // Base Condition, very crucial in designing the recursive functions.
    {
       return 1;
    }
    else
    {
       return n * factorial(n - 1);
    }
}

int main()
{
    printf("Factorial %d = %d\n", 3, factorial(3));
    return 0;
}
```

Beispielausgabe

```
Factorial 3 = 6
```

Diese Funktion endet, sobald die Bedingung `n` gleich 1 ist (vorausgesetzt, der Anfangswert von `n` ist klein genug - die obere Grenze ist `12` wenn `int` eine 32-Bit-Menge ist).

Regeln, die zu beachten sind:

1. Initialisieren Sie den Algorithmus. Rekursive Programme benötigen häufig einen Startwert. Dies wird entweder durch Verwendung eines an die Funktion übergebenen Parameters oder durch Bereitstellung einer Gateway-Funktion erreicht, die nicht rekursiv ist, aber die Ausgangswerte für die rekursive Berechnung einrichtet.
2. Prüfen Sie, ob die aktuell verarbeiteten Werte mit dem Basisfall übereinstimmen. Wenn ja, den Wert bearbeiten und zurückgeben.
3. Definieren Sie die Antwort in Bezug auf kleinere oder einfachere Unterprobleme oder Unterprobleme.
4. Führen Sie den Algorithmus für das Unterproblem aus.
5. Kombinieren Sie die Ergebnisse in der Formulierung der Antwort.
6. Gib die Ergebnisse zurück.

Quelle: Rekursive Funktion

Logischen Ausdruck gegen 'true' prüfen

Der ursprüngliche C-Standard hatte keinen intrinsischen booleschen Typ, so dass `bool` , `true` und `false` keine inhärente Bedeutung hatten und oft von Programmierern definiert wurden. Normalerweise würde " `true` als "1" definiert und " `false` als 0 definiert.

C99

C99 fügt den eingebauten Typ `_Bool` und den Header `<stdbool.h>` , die definiert , `bool` (Erweiterung auf `_Bool`), `false` und `true` . Sie können auch `bool` neu definieren, `true` und `false` , aber es wird darauf `bool` , dass dies eine veraltete Funktion ist.

Noch wichtiger ist, dass logische Ausdrücke alles, was zu Null ausgewertet wird, als falsch und jede Nicht-Null-Bewertung als wahr betrachtet. Zum Beispiel:

```
/* Return 'true' if the most significant bit is set */
bool isUpperBitSet(uint8_t bitField)
{
    if ((bitField & 0x80) == true)  /* Comparison only succeeds if true is 0x80 and bitField
has that bit set */
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

In dem obigen Beispiel versucht die Funktion zu prüfen, ob das obere Bit gesetzt ist, und gibt `true` wenn dies der `true` ist. Durch die explizite Prüfung auf " `true` wird die `if (bitfield & 0x80)` jedoch nur dann erfolgreich ausgeführt, wenn `(bitfield & 0x80)` als " `true` ausgewertet wird. `(bitfield & 0x80)` ist in der Regel `1` und sehr selten `0x80` . Überprüfen Sie entweder explizit den Fall, den Sie erwarten:

```
/* Return 'true' if the most significant bit is set */
bool isUpperBitSet(uint8_t bitField)
{
    if ((bitField & 0x80) == 0x80) /* Explicitly test for the case we expect */
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Oder bewerten Sie einen Wert ungleich Null als wahr.

```
/* Return 'true' if the most significant bit is set */
bool isUpperBitSet(uint8_t bitField)
{
    /* If upper bit is set, result is 0x80 which the if will evaluate as true */
    if (bitField & 0x80)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Fließkomma-Literale sind standardmäßig vom Typ double

Beim Initialisieren von Variablen des Typs `float` mit Literalwerten oder beim Vergleich mit Literalwerten ist Vorsicht geboten, da reguläre Fließkomma-Literale wie `0.1` vom Typ `double` . Dies kann zu Überraschungen führen:

```
#include <stdio.h>
int main() {
    float  n;
    n = 0.1;
    if (n > 0.1) printf("Wierd\n");
    return 0;
}
// Prints "Wierd" when n is float
```

Hier wird `n` initialisiert und auf einfache Genauigkeit gerundet, was den Wert 0,10000000149011612 ergibt. Dann wird `n` zurück in die doppelte Genauigkeit umgewandelt, um mit `0.1` Literal verglichen zu werden (was 0,10000000000000001 entspricht), was zu einer Nichtübereinstimmung führt.

Neben Rundungsfehlern führt das Mischen von `float` Variablen mit `double` schlechten Leistung auf Plattformen, die keine Hardware-Unterstützung für doppelte Genauigkeit bieten.

Häufige Fehler online lesen: https://riptutorial.com/de/c/topic/2006/haufige-fehler

Kapitel 28: Identifier-Bereich

Examples

Bereich blockieren

Ein Bezeichner hat einen Blockbereich, wenn seine entsprechende Deklaration innerhalb eines Blocks erscheint (Parameterdeklaration in der Funktionsdefinition gilt). Der Gültigkeitsbereich endet am Ende des entsprechenden Blocks.

Keine unterschiedlichen Entitäten mit derselben Kennung dürfen denselben Bereich haben, aber Bereiche können sich überlappen. Bei überlappenden Bereichen ist nur derjenige sichtbar, der im innersten Bereich angegeben ist.

```
#include <stdio.h>

void test(int bar)                   // bar has scope test function block
{
    int foo = 5;                     // foo has scope test function block
    {
        int bar = 10;                // bar has scope inner block, this overlaps with previous
test:bar declaration, and it hides test:bar
        printf("%d %d\n", foo, bar); // 5 10
    }                                // end of scope for inner bar
    printf("%d %d\n", foo, bar);     // 5 5, here bar is test:bar
}                                    // end of scope for test:foo and test:bar

int main(void)
{
    int foo = 3;         // foo has scope main function block

    printf("%d\n", foo); // 3
    test(5);
    printf("%d\n", foo); // 3
    return 0;
}                        // end of scope for main:foo
```

Funktionsprototyp-Umfang

```
#include <stdio.h>

/* The parameter name, apple, has function prototype scope.  These names
   are not significant outside the prototype itself.  This is demonstrated
   below. */

int test_function(int apple);

int main(void)
{
    int orange = 5;

    orange = test_function(orange);
    printf("%d\r\n", orange); //orange = 6
```

```
    return 0;
}

int test_function(int fruit)
{
    fruit += 1;
    return fruit;
}
```

Beachten Sie, dass Sie verwirrende Fehlermeldungen erhalten, wenn Sie einen Typnamen in einen Prototyp einfügen:

```
int function(struct whatever *arg);

struct whatever
{
    int a;
    // ...
};

int function(struct whatever *arg)
{
    return arg->a;
}
```

Mit GCC 6.3.0 erzeugt dieser Code (Quelldatei `dc11.c`):

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -c dc11.c
dc11.c:1:25: error: ‘struct whatever’ declared inside parameter list will not be visible
outside of this definition or declaration [-Werror]
     int function(struct whatever *arg);
                         ^~~~~~~~
dc11.c:9:9: error: conflicting types for ‘function’
     int function(struct whatever *arg)
         ^~~~~~~~
dc11.c:1:9: note: previous declaration of ‘function’ was here
     int function(struct whatever *arg);
         ^~~~~~~~
cc1: all warnings being treated as errors
$
```

Platzieren Sie die Strukturdefinition vor der Funktionsdeklaration oder fügen Sie eine `struct whatever;` als Zeile vor der Funktionsdeklaration, und es gibt kein Problem. Sie sollten keine neuen Typnamen in einen Funktionsprototyp einfügen, da es keine Möglichkeit gibt, diesen Typ zu verwenden, und daher keine Möglichkeit, diese Funktion zu definieren oder zu verwenden.

Dateibereich

```
#include <stdio.h>

/* The identifier, foo, is declared outside all blocks.
   It can be used anywhere after the declaration until the end of
   the translation unit. */
static int foo;
```

```
void test_function(void)
{
    foo += 2;
}

int main(void)
{
    foo = 1;

    test_function();
    printf("%d\r\n", foo); //foo = 3;

    return 0;
}
```

Funktionsumfang

Funktionsumfang ist der spezielle Umfang für **Etiketten** . Dies liegt an ihrer ungewöhnlichen Eigenschaft. Ein **Label** ist durch die gesamte Funktion sichtbar, in der es definiert ist, und man kann (mit der Anweisung `goto label`) von jedem Punkt derselben Funktion zu ihm springen. Obwohl es nicht nützlich ist, veranschaulicht das folgende Beispiel den Punkt:

```
#include <stdio.h>

int main(int argc,char *argv[]) {
    int a = 0;
    goto INSIDE;
  OUTSIDE:
    if (a!=0) {
        int i=0;
      INSIDE:
        printf("a=%d\n",a);
        goto OUTSIDE;
    }
}
```

`INSIDE` mag *innerhalb* des `if` Blocks definiert zu sein, wie es für `i` der Gültigkeitsbereich der Block ist, aber nicht. Es ist in der gesamten Funktion sichtbar, wenn die Anweisung `goto INSIDE;` veranschaulicht. Es können also nicht zwei Labels mit derselben Kennung in einer einzelnen Funktion enthalten sein.

Eine mögliche Verwendung ist das folgende Muster, um korrekte komplexe Bereinigungen zugewiesener Ressourcen zu realisieren:

```
#include <stdlib.h>
#include <stdio.h>

void a_function(void) {
   double* a = malloc(sizeof(double[34]));
   if (!a) {
      fprintf(stderr,"can't allocate\n");
      return;                 /* No point in freeing a if it is null */
   }
   FILE* b = fopen("some_file","r");
```

```
    if (!b) {
        fprintf(stderr,"can't open\n");
        goto CLEANUP1;          /* Free a; no point in closing b */
    }
    /* do something reasonable */
    if (error) {
        fprintf(stderr,"something's wrong\n");
        goto CLEANUP2;       /* Free a and close b to prevent leaks */
    }
    /* do yet something else */
CLEANUP2:
    close(b);
CLEANUP1:
    free(a);
}
```

`CLEANUP1` wie `CLEANUP1` und `CLEANUP2` sind spezielle Bezeichner, die sich von allen anderen Bezeichnern unterscheiden. Sie sind von überall innerhalb der Funktion sichtbar, auch an Stellen, die vor der gekennzeichneten Anweisung ausgeführt werden, oder sogar an Stellen, die niemals erreichbar wären, wenn kein `goto` ausgeführt wird. Beschriftungen werden häufig in Kleinbuchstaben und nicht in Großbuchstaben geschrieben.

Identifier-Bereich online lesen: https://riptutorial.com/de/c/topic/1804/identifier-bereich

Kapitel 29: Implementierungsdefiniertes Verhalten

Bemerkungen

Überblick

Der C-Standard beschreibt die Sprachsyntax, die von der Standardbibliothek bereitgestellten Funktionen und das Verhalten von konformen C-Prozessoren (grob gesagt Compiler) und konformen C-Programmen. In Bezug auf das Verhalten spezifiziert der Standard größtenteils bestimmte Verhaltensweisen für Programme und Prozessoren. Andererseits haben einige Operationen ein explizites oder implizites *undefiniertes Verhalten* - solche Operationen sind immer zu vermeiden, da Sie sich auf nichts verlassen können. Dazwischen gibt es verschiedene *implementierungsdefinierte* Verhaltensweisen. Diese Verhaltensweisen können zwischen C-Prozessoren, Laufzeiten und Standardbibliotheken (gemeinsam *Implementierungen*) variieren, sie sind jedoch für jede gegebene Implementierung konsistent und zuverlässig, und konforme Implementierungen dokumentieren ihr Verhalten in jedem dieser Bereiche.

Es ist manchmal vernünftig, dass ein Programm auf implementierungsdefiniertes Verhalten angewiesen ist. Wenn das Programm zum Beispiel ohnehin spezifisch für eine bestimmte Betriebsumgebung ist, ist es unwahrscheinlich, dass es von Problemen der Implementierung abhängig ist, die allgemein auf die allgemeinen Prozessoren für diese Umgebung angewendet werden. Alternativ können Sie Direktive zur bedingten Kompilierung verwenden, um durch die Implementierung definierte Verhaltensweisen auszuwählen, die für die verwendete Implementierung geeignet sind. In jedem Fall ist es wichtig zu wissen, für welche Operationen das Verhalten der Implementierung definiert ist, um sie entweder zu vermeiden oder eine fundierte Entscheidung darüber zu treffen, ob und wie sie verwendet werden sollen.

Der Saldo dieser Anmerkungen bildet eine Liste aller durch die Implementierung definierten Verhaltensweisen und Merkmale, die im C2011-Standard festgelegt sind, mit Verweisen auf den Standard. Viele von ihnen verwenden die Terminologie des Standards . Einige andere sind allgemeiner auf den Kontext des Standards angewiesen, z. B. auf die acht Stufen der Übersetzung von Quellcode in ein Programm oder den Unterschied zwischen gehosteten und freistehenden Implementierungen. Einige, die besonders überraschend oder bemerkenswert sind, werden in fetter Schrift dargestellt. Nicht alle beschriebenen Verhaltensweisen werden von früheren C-Standards unterstützt, im Allgemeinen weisen sie jedoch in allen Versionen des Standards, die sie unterstützen, ein durch die Implementierung definiertes Verhalten auf.

Programme und Prozessoren

Allgemeines

- **Die Anzahl der Bits in einem Byte** (3,6 / 3). Mindestens `8` kann der Istwert mit dem Makro

`CHAR_BIT` abgefragt werden.

- Welche Ausgabenachrichten gelten als "Diagnosemeldungen" (3.10 / 1)

Quellübersetzung

- Die Art und Weise, in der physische Quelldatei-Multibyte-Zeichen dem Quellzeichensatz (5.1.1.2/1) zugeordnet werden.
- Ob nicht leere Sequenzen von nicht Newline-Leerzeichen in der Übersetzungsphase 3 durch einzelne Leerzeichen ersetzt werden (5.1.1.2/1)
- Das oder die Ausführungssatzzeichen, in die Zeichenliterale und Zeichen in Zeichenfolgekonstanten umgewandelt werden (während der Übersetzungsphase 5), wenn ansonsten kein entsprechendes Zeichen vorhanden ist (5.1.1.2/1).

Betriebsumgebung

- Die Art und Weise, in der die zu sendenden Diagnosemeldungen identifiziert werden (5.1.1.3/1).
- Name und Typ der aufgerufenen Funktion in einer freistehenden Implementierung (5.1.2.1/1).
- Welche Bibliotheksfunktionen stehen in einer freistehenden Implementierung über einen festgelegten Mindestsatz hinaus zur Verfügung (5.1.2.1/1).
- Die Auswirkung der Programmbeendigung in einer freistehenden Umgebung (5.1.2.1/2).
- In einer gehosteten Umgebung alle zulässigen Signaturen für die `main()` Funktion außer `int main(int argc, char *arg[])` und `int main(void)` (5.1.2.2.1 / 1).
- Die Art und Weise, in der eine gehostete Implementierung die Zeichenfolgen definiert, auf die das zweite Argument von `main()` (5.1.2.2.1 / 2) verweist .
- Was ist ein "interaktives Gerät" im Sinne der Abschnitte 5.1.2.3 (Programmausführung) und 7.21.3 (Dateien) (5.1.2.3/7).
- Einschränkungen für Objekte, auf die von Interrupt-Handler-Routinen in einer Optimierungsimplementierung verwiesen wird (5.1.2.3/10).
- In einer freistehenden Implementierung, ob mehrere Ausführungsthreads unterstützt werden (5.1.2.4/1).
- Die Werte der Mitglieder des Ausführungszeichensatzes (5.2.1 / 1).
- Die `char` entsprechen den definierten alphabetischen Escape-Sequenzen (5.2.2 / 3).
- **Die numerischen Ganzzahl- und Fließkommazahlen und** -merkmale (5.2.4.2/1).

- Die Genauigkeit der Fließkomma-Arithmetikoperationen und der Konvertierungen der Standardbibliothek von internen Fließkommadaten in Stringdarstellungen (5.2.4.2.2 / 6).
- Der Wert des Makros `FLT_ROUNDS` , das den Standardrundungsmodus mit Gleitkommazahlen codiert (5.2.4.2.2 / 8).
- Rundungsverhalten, das durch unterstützte Werte von `FLT_ROUNDS` mehr als 3 oder weniger als -1 (5.2.4.2.2 / 8) gekennzeichnet ist.
- Der Wert des Makros `FLT_EVAL_METHOD` , das das Verhalten der Gleitkomma-Auswertung kennzeichnet (5.2.4.2.2 / 9).
- Das Verhalten wird durch unterstützte Werte von `FLT_EVAL_METHOD` weniger als -1 (5.2.4.2.2 / 9) charakterisiert .
- Die Werte der Makros `FLT_HAS_SUBNORM` , `DBL_HAS_SUBNORM` und `LDBL_HAS_SUBNORM` , ob die Standard-Gleitkommaformate Subnormalzahlen unterstützen (5.2.4.2.2 / 10).

Typen

- Das Ergebnis des Versuchs, (indirekt) auf ein Objekt mit Threadspeicherdauer von einem anderen Thread als demjenigen zuzugreifen, dem das Objekt zugeordnet ist (6.2.4 / 4).
- Der Wert eines `char` ein Zeichen außerhalb des Grundausführungssatzes hat , zu denen (zugeordnet 6.2.5 / 3).
- Die unterstützten erweiterten Integer-Typen mit Vorzeichen (falls vorhanden) (6.2.5 / 4) und alle Erweiterungsschlüsselwörter, mit denen sie identifiziert werden.
- Gibt an, **ob `char` dieselbe Repräsentation und dasselbe Verhalten wie `signed char` oder als `unsigned char`** (6.2.5 / 15). Kann mit `CHAR_MIN` abgefragt werden. `CHAR_MIN` ist entweder `0` oder `SCHAR_MIN` wenn `char` unsigniert bzw. signiert ist.
- **Anzahl, Reihenfolge und Kodierung von Bytes in den Darstellungen von Objekten** , sofern nicht ausdrücklich im Standard (6.2.6.1/2) angegeben.
- **Welche der drei erkannten Formen der Ganzzahldarstellung gilt in einer bestimmten Situation und ob bestimmte Bitmuster von Ganzzahlobjekten** Trapdarstellungen **sind** (6.2.6.2/2).
- Die Ausrichtungsanforderung jedes Typs (6.2.8 / 1).
- Ob und in welchen Kontexten erweiterte Ausrichtungen unterstützt werden (6.2.8 / 3).
- Der Satz unterstützter erweiterter Ausrichtungen (6.2.8 / 4).
- Die Ganzzahlkonvertierungsranking aller erweiterten Ganzzahlentypen mit Vorzeichen relativ zueinander (6.3.1.1/1).
- **Die Auswirkung der Zuweisung eines Bereichs außerhalb des Bereichs zu einer**

vorzeichenbehafteten Ganzzahl (6.3.1.3/3).

- Wenn einem Gleitkommaobjekt ein nicht darstellbarer Wert innerhalb des Bereichs zugewiesen wird, wird die Art des im Objekt gespeicherten darstellbaren Werts aus den beiden nächstgelegenen darstellbaren Werten (6.3.1.4/2 ; 6.3.1.5/1 ; 6.4.4.2) ausgewählt / 3).
- **Das Ergebnis der Konvertierung einer Ganzzahl in einen** Zeigertyp mit Ausnahme von ganzzahligen konstanten Ausdrücken mit dem Wert `0` (6.3.2.3/5).

Quellformular

- Die Speicherorte in `#pragma` Direktiven, an denen `#pragma` erkannt werden (6.4 / 4).
- Die Zeichen, einschließlich Multibyte-Zeichen, mit Ausnahme von Unterstrich, lateinischen Buchstaben ohne Akzent , universelle Zeichennamen und Dezimalstellen, die in Bezeichnern (6.4.2.1/1) vorkommen können.
- **Die Anzahl der signifikanten Zeichen in einem Bezeichner** (6.4.2.1/5).
- Mit einigen Ausnahmen wird die Art und Weise, in der die Quellzeichen in einer Ganzzahl-Zeichenkonstante auf Ausführungssatzzeichen (6.4.4.4/2 ; 6.4.4.4/10) abgebildet werden .
- Das aktuelle Gebietsschema, das zur Berechnung des Werts einer Wide-Zeichenkonstante verwendet wird, und die meisten anderen Aspekte der Konvertierung für viele dieser Konstanten (6.4.4.4/11).
- Gibt an, ob Wide-String-Literal-Token mit unterschiedlichen Präfixen verkettet werden können, und falls ja, die Behandlung der resultierenden Multibyte-Zeichenfolge (6.4.5 / 5)
- Das Gebietsschema, das während der Übersetzungsphase 7 zum Konvertieren von Wide-String-Literalen in Mehrbyte-Zeichenfolgen verwendet wird, und deren Wert, wenn das Ergebnis nicht im Ausführungszeichensatz darstellbar ist (6.4.5 / 6).
- Die Art und Weise, in der Header-Namen Dateinamen zugeordnet werden (6.4.7 / 2).

Auswertung

- Ob und wie Fließkommaausdrücke kontrahiert werden, wenn `FP_CONTRACT` nicht verwendet wird (6.5 / 8).
- **Die Werte der Ergebnisse der `sizeof` und `_Alignof` Operatoren** (6.5.3.4/5).
- Die Größe des Ergebnistyps der Zeigersubtraktion (6.5.6 / 9).
- **Das Ergebnis der Verschiebung einer vorzeichenbehafteten Ganzzahl mit negativem Wert** (6,5,7 / 5) nach rechts .

Laufzeitverhalten

- Inwieweit das `register` gültig ist (6.7.1 / 6)
- Ob der Typ eines bitfield als deklariert `int` ist der gleiche Typ wie `unsigned int` oder als `signed int` (6.7.2 / 5).
- Welche Arten von Bitfeldern können anders als optional qualifiziertes `_Bool` , `signed int` und `unsigned int` ; ob Bitfelder atomare Typen haben können (6.7.2.1/5).
- Aspekte, wie Implementierungen den Speicher für Bitfelder auslegen (6.7.2.1/11).
- Die Ausrichtung von Nicht-Bitfeld-Mitgliedern von Strukturen und Vereinigungen (6.7.2.1/14).
- Der zugrunde liegende Typ für jeden aufgelisteten Typ (6.7.2.2/4).
- Was ist ein "Zugriff" auf ein Objekt vom Typ " `volatile` " (6.7.3 / 7).
- Die Wirksamkeit von `inline` Funktionsdeklarationen (6.7.4 / 6).

Präprozessor

- Ob Zeichenkonstanten in Präprozessor-Bedingungen in ganzzahlige Werte konvertiert werden wie in gewöhnlichen Ausdrücken und ob eine Einzelzeichenkonstante einen negativen Wert (6.10.1 / 4) haben kann.
- Die Speicherorte suchten nach Dateien, die in einer `#include` Direktive (6.10.2 / 2-3) angegeben wurden.
- Die Art und Weise, in der ein Headername aus den Token einer Multi- Token- `#include` Direktive (6.10.2 / 4) gebildet wird.
- Die Grenze für die `#include` Verschachtelung von `#include` (6.10.2 / 6).
- Gibt an, ob ein `\` Zeichen vor dem `\` eingefügt wird, um einen universellen Zeichennamen in das Ergebnis des Operators `#` des Präprozessors (6.10.3.2/2) einzufügen .
- Das Verhalten der `#pragma` Vorverarbeitungsrichtlinie für andere Pragmas als `STDC` (6.10.6 / 1).
- Der Wert der Makros `__DATE__` und `__TIME__` wenn kein Übersetzungsdatum bzw. -zeit verfügbar ist (6.10.8.1/1).
- Die interne Zeichencodierung für `wchar_t` wenn das Makro `__STDC_ISO_10646__` nicht definiert ist (6.10.8.2/1).
- Die interne Zeichenkodierung für `char32_t` wenn das Makro `__STDC_UTF_32__` nicht definiert ist (6.10.8.2/1).

Standardbibliothek

Allgemeines

- Das Format der Nachrichten, die gesendet werden, wenn Assertions fehlschlagen (7.2.1.1/2).

Gleitkomma-Umgebungsfunktionen

- Alle zusätzlichen Fließkomma-Ausnahmen, die über die im Standard definierten (7.6 / 6) hinausgehen.
- Alle zusätzlichen Gleitkomma-Rundungsmodi, die über die vom Standard definierten Werte (7.6 / 8) hinausgehen.
- Alle zusätzlichen Gleitkomma-Umgebungen, die über die vom Standard definierten (7.6 / 10) hinausgehen.
- Der Standardwert des Zugriffsschalters für die Gleitkommaumgebung (7.6.1 / 2).
- Die mit `fegetexceptflag()` (7.6.2.2/1) aufgezeichnete Darstellung der Gleitkommazustandsflags.
- `feraiseexcept()` ob die Funktion `feraiseexcept()` zusätzlich die Gleitkommaausnahme "ungenau" `feraiseexcept()` wenn sie die Gleitkommaausnahme "Überlauf" oder "Unterlauf" (7.6.2.3/2) auslöst .

Gebietsschema-bezogene Funktionen

- Die anderen `setlocale()` als `"C"` von `setlocale()` (7.11.1.1/3) unterstützt.

Mathematische Funktionen

- Die durch `float_t` und `double_t` `float_t` `double_t` wenn das Makro `FLT_EVAL_METHOD` einen anderen Wert als `0` , `1` und `2` (7.12 / 2) hat.
- Alle unterstützten Fließkomma-Klassifizierungen, die über die vom Standard definierten Werte (7.12 / 6) hinausgehen.
- Der von `math.h` zurückgegebene Wert im Falle eines Domänenfehlers (7.12.1 / 2).
- Der von `math.h` zurückgegebene Wert funktioniert bei einem Polfehler (7.12.1 / 3).
- Der von `math.h` zurückgegebene Wert funktioniert, wenn das Ergebnis `math.h` , und Aspekte, ob `errno` auf `ERANGE` und ob unter diesen Umständen eine Gleitkommaausnahme `errno` wird (7.12.1 / 6).
- Der Standardwert des FP-Kontraktionsschalters (7.12.2 / 2).
- `fmod()` ob die `fmod()` Funktionen 0 zurückgeben oder einen Domänenfehler auslösen, wenn das zweite Argument 0 (7.12.10.1/3) ist.

- Gibt an, ob die `remainder()` Funktionen 0 zurückgeben oder einen Domänenfehler auslösen, wenn das zweite Argument 0 (7.12.10.2/3) ist.
- Die Anzahl der signifikanten Bits in den Quotientenmodulen, die von den Funktionen `remquo()` (7.12.10.3/2) berechnet werden.
- `remquo()` ob die `remquo()` Funktionen 0 zurückgeben oder einen Domänenfehler auslösen, wenn das zweite Argument 0 (7.12.10.3/3) ist.

Signale

- Der vollständige Satz unterstützter Signale, ihre Semantik und ihre Standardbehandlung (7.14 / 4).
- Wenn ein Signal ausgelöst wird und ein benutzerdefinierter Handler mit diesem Signal verbunden ist, werden die Signale, sofern vorhanden, für die Dauer der Ausführung des Handlers gesperrt (7.14.1.1/3).
- Welche anderen Signale als `SIGFPE` , `SIGILL` und `SIGSEGV` bewirken, dass das Verhalten bei der Rückkehr von einem benutzerdefinierten Signalhandler undefiniert ist (7.14.1.1/3).
- Welche Signale sind anfangs so konfiguriert, dass sie ignoriert werden (unabhängig von ihrer Standardbehandlung; 7.14.1.1/6).

Verschiedenes

- Die spezifische Nullzeigerkonstante, auf die sich das Makro `NULL` erweitert (7.19 / 3).

Dateibehandlungsfunktionen

- Gibt an, ob die letzte Zeile eines Textstroms eine abschließende neue Zeile erfordert (7.21.2 / 2).
- Die Anzahl der Nullzeichen, die automatisch an einen Binärstrom angehängt werden (7.21.2 / 3).
- Die Anfangsposition einer Datei, die im Anfügemodus (7.21.3 / 1) geöffnet wurde.
- Gibt an, ob beim Schreiben in einen Textstream der Stream abgeschnitten wird (7.21.3 / 2).
- Unterstützung für Stream-Pufferung (7.21.3 / 3).
- Gibt an, ob Dateien mit der Länge Null vorhanden sind (7.21.3 / 4).
- Die Regeln für das Erstellen gültiger Dateinamen (7.21.3 / 8).
- Ob dieselbe Datei gleichzeitig mehrmals geöffnet werden kann (7.21.3 / 8).
- Art und Wahl der Kodierung für Multibyte-Zeichen (7.21.3 / 10).

- Das Verhalten der Funktion `remove()` , wenn die Zieldatei geöffnet ist (7.21.4.1/2).
- Das Verhalten der Funktion rename `rename()` , wenn die Zieldatei bereits vorhanden ist (7.21.4.2/2).
- `tmpfile()` ob Dateien, die mit der Funktion `tmpfile()` , entfernt werden, falls das Programm abnormal beendet wird (7.21.4.3/2).
- Welcher Modus unter welchen Umständen wechselt, ist über `freopen()` (7.21.5.4/3) zulässig.

E / A-Funktionen

- Welche der zulässigen Darstellungen von unendlichen und nicht-nummerierten FP-Werten von den Funktionen der printf () - Familie erzeugt werden (7.21.6.1/8).
- Die Art und Weise, in der Zeiger von den Funktionen der `printf()` -Familie (7.21.6.1/8) formatiert werden.
- Das Verhalten der `scanf()` -Familie funktioniert, wenn das Zeichen `-` an einer internen Position der Scanliste eines `[` -Felds (7.21.6.2/12) erscheint.
- Die meisten Aspekte der Übergabe von `p` Feldern in den `scanf()` -Familienfunktionen (7.21.6.2/12).
- Der von `fgetpos()` bei einem Fehler festgelegte `errno` Wert (7.21.9.1/2).
- Der von `fsetpos()` bei einem Fehler `errno` Wert (7.21.9.3/2).
- Der von `ftell()` bei einem Fehler `errno` Wert (7.21.9.4/3).
- Die Bedeutung der `strtod()` -Familienfunktionen einiger unterstützter Aspekte einer NaN-Formatierung (7.22.1.3p4).
- `strtod()` ob die `strtod()` -Familie-Funktionen `errno` auf `ERANGE` wenn das Ergebnis unterläuft (7.22.1.3/10).

Speicherbelegungsfunktionen

- Das Verhalten der Speicherzuordnung funktioniert, wenn die Anzahl der angeforderten Bytes 0 ist (7.22.3 / 1).

Systemumgebungsfunktionen

- Welche Bereinigungen werden durchgeführt und welcher Status wird an das Hostbetriebssystem zurückgegeben, wenn die Funktion `abort()` aufgerufen wird (7.22.4.1/2).
- Welcher Status wird an die Hostumgebung zurückgegeben, wenn `exit()` aufgerufen wird (

7.22.4.4/5).

- Die Behandlung offener Streams und welcher Status wird an die `_Exit()` zurückgegeben, wenn `_Exit()` aufgerufen wird (7.22.4.5/2).
- Die über `getenv()` zugänglichen Umgebungsnamen und die Methode zum Ändern der Umgebung (7.22.4.6/2).
- Der Rückgabewert der Funktion `system()` (7.22.4.8/3).

Datums- und Zeitfunktionen

- Die lokale Zeitzone und die Sommerzeit (7.27.1 / 1).
- Der Bereich und die Genauigkeit der Zeiten können mit den Typen `clock_t` und `time_t` (7.27.1 / 4) dargestellt werden.
- Der Beginn der Ära, der als Referenz für die von der Funktion `clock()` Zeiten (7.27.2.1/3) dient.
- Der Beginn der Epoche, die als Referenz für die von der Funktion `timespec_get()` Zeiten dient (wenn die Zeitbasis `TIME_UTC` ; 7.27.2.5/3).
- Die `strftime()` für den `%Z` Konvertierungsspezifizierer in der Ländereinstellung "C" (7.27.3.5/7).

Wide-Character-E / A-Funktionen

- Welche der zulässigen Darstellungen von unendlichen und nicht-nummerierten FP-Werten von den Funktionen der `wprintf()` erzeugt werden (7.29.2.1/8).
- Die Art und Weise, in der Zeiger von den Funktionen der `wprintf()` -Familie (7.29.2.1/8) formatiert werden.
- Das Verhalten der `wscanf()` -Familie funktioniert, wenn das Zeichen `-` an einer internen Position der Scanliste eines `[` -Feldes (7.29.2.2/12) erscheint.
- Die meisten Aspekte der `wscanf()` -Familie behandeln `p` Felder (7.29.2.2/12).
- Die Bedeutung der `wstrtod()` -Familie einiger unterstützter Aspekte der NaN-Formatierung (7.29.4.1.1 / 4).
- `wstrtod()` ob die `wstrtod()` -Familienfunktionen `errno` auf `ERANGE` wenn das Ergebnis untergeht (7.29.4.1.1 / 10)

Examples

Rechtsverschiebung einer negativen Ganzzahl

```
int signed_integer = -1;

// The right shift operation exhibits implementation-defined behavior:
int result = signed_integer >> 1;
```

Einer Ganzzahl einen Wert außerhalb des Bereichs zuweisen

```
// Supposing SCHAR_MAX, the maximum value that can be represented by a signed char, is
// 127, the behavior of this assignment is implementation-defined:
signed char integer;
integer = 128;
```

Zuweisung von null Bytes

```
// The allocation functions have implementation-defined behavior when the requested size
// of the allocation is zero.
void *p = malloc(0);
```

Darstellung von signierten ganzen Zahlen

Jeder vorzeichenbehaftete ganzzahlige Typ kann in einem von drei Formaten dargestellt werden; Es ist implementierungsdefiniert, welche verwendet wird. Die für jeden gegebenen Integer-Typ mit Vorzeichen verwendete Implementierung, die mindestens so breit wie `int` ist, kann zur Laufzeit aus den zwei niedrigstwertigen Bits der Repräsentation von Wert `-1` in diesem Typ bestimmt werden:

```
enum { sign_magnitude = 1, ones_compl = 2, twos_compl = 3, };
#define SIGN_REP(T) ((T)-1 & (T)3)

switch (SIGN_REP(long)) {
   case sign_magnitude: { /* do something */ break; }
   case ones_compl:     { /* do otherwise */ break; }
   case twos_compl:     { /* do yet else  */ break; }
   case 0:  { _Static_assert(SIGN_REP(long), "bogus sign representation"); }
}
```

Dasselbe Muster gilt für die Darstellung engerer Typen, sie können jedoch mit dieser Technik nicht getestet werden, da die Operanden von `&` vor den Berechnungen "den üblichen arithmetischen Konvertierungen" unterliegen.

Implementierungsdefiniertes Verhalten online lesen: https://riptutorial.com/de/c/topic/4832/implementierungsdefiniertes-verhalten

Kapitel 30: Implizite und explizite Konvertierungen

Syntax

- Explizite Konvertierung (auch bekannt als "Casting"): (Typ) Ausdruck

Bemerkungen

" *Explizite Konvertierung* " wird im Allgemeinen auch als "Casting" bezeichnet.

Examples

Ganzzahlkonvertierungen in Funktionsaufrufen

Da die Funktion über einen geeigneten Prototyp verfügt, werden Ganzzahlen für Aufrufe von Funktionen nach den Regeln der Ganzzahlkonvertierung erweitert. C11 6.3.1.3.

> **6.3.1.3 Vorzeichenbehaftete und vorzeichenlose Ganzzahlen**
> Wenn ein Wert mit dem ganzzahligen Typ in einen anderen ganzzahligen Typ als _Bool konvertiert wird, bleibt der Wert unverändert, wenn der Wert durch den neuen Typ dargestellt werden kann.
>
> Wenn der neue Typ nicht vorzeichenbehaftet ist, wird der Wert ansonsten konvertiert, indem wiederholt ein Wert mehr als der Maximalwert hinzugefügt oder subtrahiert wird, der im neuen Typ dargestellt werden kann, bis der Wert im Bereich des neuen Typs liegt.
>
> Andernfalls ist der neue Typ vorzeichenbehaftet und der Wert kann nicht darin dargestellt werden. Entweder ist das Ergebnis implementierungsdefiniert oder ein implementierungsdefiniertes Signal wird ausgelöst.

Normalerweise sollten Sie einen Wide-Signed-Typ nicht auf einen engeren Signaltyp abschneiden, da die Werte offensichtlich nicht passen und es keine eindeutige Bedeutung für diesen Wert gibt. Der oben zitierte C-Standard definiert diese Fälle als "implementierungsdefiniert", das heißt, sie sind nicht portierbar.

Im folgenden Beispiel wird angenommen, dass `int` 32 Bit breit ist.

```
#include <stdio.h>
#include <stdint.h>

void param_u8(uint8_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}
```

```
void param_u16(uint16_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}

void param_u32(uint32_t val) {
    printf("%s val is %u\n", __func__, val);  /* here val fits into unsigned */
}

void param_u64(uint64_t val) {
    printf("%s val is " PRI64u "\n", __func__, val); /* Fixed with format string */
}

void param_s8(int8_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}

void param_s16(int16_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}

void param_s32(int32_t val) {
    printf("%s val is %d\n", __func__, val); /* val has same width as int */
}

void param_s64(int64_t val) {
    printf("%s val is " PRI64d "\n", __func__, val); /* Fixed with format string */
}

int main(void) {

    /* Declare integers of various widths */
    uint8_t  u8  = 127;
    uint8_t  s64  = INT64_MAX;

    /* Integer argument is widened when function parameter is wider */
    param_u8(u8);   /* param_u8 val is 127 */
    param_u16(u8);  /* param_u16 val is 127 */
    param_u32(u8);  /* param_u32 val is 127 */
    param_u64(u8);  /* param_u64 val is 127 */
    param_s8(u8);   /* param_s8 val is 127 */
    param_s16(u8);  /* param_s16 val is 127 */
    param_s32(u8);  /* param_s32 val is 127 */
    param_s64(u8);  /* param_s64 val is 127 */

    /* Integer argument is truncated when function parameter is narrower */
    param_u8(s64);  /* param_u8 val is 255 */
    param_u16(s64); /* param_u16 val is 65535 */
    param_u32(s64); /* param_u32 val is 4294967295 */
    param_u64(s64); /* param_u64 val is 9223372036854775807 */
    param_s8(s64);  /* param_s8 val is implementation defined */
    param_s16(s64); /* param_s16 val is implementation defined */
    param_s32(s64); /* param_s32 val is implementation defined */
    param_s64(s64); /* param_s64 val is 9223372036854775807 */

    return 0;
}
```

Zeigerkonvertierungen in Funktionsaufrufen

Zeigerkonvertierungen in `void*` sind implizit, aber jede andere Zeigerkonvertierung muss explizit sein. Während der Compiler eine explizite Konvertierung von einem beliebigen Zeiger-zu-Datentyp in einen anderen Zeiger-zu-Datentyp erlaubt, ist der Zugriff auf ein Objekt über einen falsch typisierten Zeiger fehlerhaft und führt zu undefiniertem Verhalten. Diese sind nur zulässig, wenn die Typen kompatibel sind oder wenn der Zeiger, mit dem Sie das Objekt betrachten, ein Zeichentyp ist.

```
#include <stdio.h>

void func_voidp(void* voidp) {
    printf("%s Address of ptr is %p\n", __func__, voidp);
}

/* Structures have same shape, but not same type */
struct struct_a {
    int a;
    int b;
} data_a;

struct struct_b {
    int a;
    int b;
} data_b;

void func_struct_b(struct struct_b* bp) {
    printf("%s Address of ptr is %p\n", __func__, (void*) bp);
}

int main(void) {


    /* Implicit ptr conversion allowed for void* */
    func_voidp(&data_a);

    /*
     * Explicit ptr conversion for other types
     *
     * Note that here although the have identical definitions,
     * the types are not compatible, and that the this call is
     * erroneous and leads to undefined behavior on execution.
     */
    func_struct_b((struct struct_b*)&data_a);

    /* My output shows: */
    /* func_charp Address of ptr is 0x601030 */
    /* func_voidp Address of ptr is 0x601030 */
    /* func_struct_b Address of ptr is 0x601030 */

    return 0;
}
```

Implizite und explizite Konvertierungen online lesen: https://riptutorial.com/de/c/topic/2529/implizite-und-explizite-konvertierungen

Kapitel 31: Initialisierung

Examples

Initialisierung von Variablen in C

Ohne explizite Initialisierung werden externe und `static` Variablen garantiert auf Null initialisiert. automatische Variablen (einschließlich der `register` haben *unbestimmte* [1] (dh garbage) Anfangswerte.

Skalare Variablen können initialisiert werden, wenn sie definiert werden, indem dem Namen ein Gleichheitszeichen und ein Ausdruck folgen:

```
int x = 1;
char squota = '\'';
long day = 1000L * 60L * 60L * 24L; /* milliseconds/day */
```

Bei externen und `static` Variablen muss der Initialisierer ein *konstanter Ausdruck* [2 sein] . Die Initialisierung erfolgt einmalig, konzeptionell, bevor das Programm ausgeführt wird.

Bei automatischen und `register` ist der Initialisierer nicht darauf beschränkt, eine Konstante zu sein: Es kann sich um einen beliebigen Ausdruck handeln, der zuvor definierte Werte enthält, sogar Funktionsaufrufe.

Sehen Sie sich zum Beispiel den Code-Ausschnitt unten an

```
int binsearch(int x, int v[], int n)
{
   int low = 0;
   int high = n - 1;
   int mid;
   ...
}
```

anstatt

```
   int low, high, mid;

   low = 0;
   high = n - 1;
```

Die Initialisierung automatischer Variablen ist eigentlich nur eine Abkürzung für Zuweisungsanweisungen. Welche Form zu bevorzugen, ist weitgehend Geschmacksache. Im Allgemeinen verwenden wir explizite Zuweisungen, da Initialisierer in Deklarationen schwieriger zu erkennen sind und weiter von der Verwendungsstelle entfernt sind. Auf der anderen Seite sollten Variablen nur dann deklariert werden, wenn sie verwendet werden, wann immer dies möglich ist.

Array initialisieren:

Ein Array kann initialisiert werden, indem es seiner Deklaration mit einer Liste von Initialisierern folgt, die in geschweifte Klammern eingeschlossen und durch Kommas getrennt sind.

Um beispielsweise ein Array-Tag mit der Anzahl der Tage in jedem Monat zu initialisieren:

```
int days_of_month[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
```

(Beachten Sie, dass der Januar in dieser Struktur als Monat Null codiert ist.)

Wenn die Größe des Arrays weggelassen wird, berechnet der Compiler die Länge, indem er die Initialisierer zählt, von denen in diesem Fall 12 vorhanden sind.

Wenn für ein Array weniger Initialisierer als die angegebene Größe vorhanden sind, sind die anderen für alle Variablentypen gleich Null.

Es ist ein Fehler, zu viele Initialisierer zu haben. Es gibt keine Standardmethode, um die Wiederholung eines Initialisierers anzugeben. GCC hat jedoch eine Erweiterung, um dies zu tun.

C99

In C89 / C90 oder früheren Versionen von C gab es keine Möglichkeit, ein Element in der Mitte eines Arrays zu initialisieren, ohne alle vorherigen Werte anzugeben.

C99

Mit C99 und darüber, bezeichnet initializers können Sie beliebige Elemente eines Arrays initialisieren, alle nicht initialisierte Werte als Nullen zu verlassen.

Character Arrays initialisieren:

Character Arrays sind ein Spezialfall der Initialisierung. Anstelle der geschriebenen Klammern und Kommas kann eine Zeichenfolge verwendet werden:

```
char chr_array[] = "hello";
```

ist eine Abkürzung für die längere aber gleichwertige:

```
char chr_array[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

In diesem Fall beträgt die Array-Größe sechs (fünf Zeichen plus das abschließende `'\0'`).

[1] Was passiert mit einer deklarierten, nicht initialisierten Variablen in C? Hat es einen Wert?

[2] Beachten Sie, dass ein *konstanter Ausdruck* als etwas definiert wird, das zur Kompilierzeit ausgewertet werden kann. `int global_var = f();` ist ungültig. Ein anderes weit verbreitetes Missverständnis ist, eine `const` qualifizierte Variable als *konstanten Ausdruck zu betrachten* . In C bedeutet `const` "Nur-Lesen", nicht "Kompilierzeitkonstante". Also globale Definitionen wie `const int SIZE = 10; int global_arr[SIZE];` und `const int SIZE = 10; int global_var = SIZE;` sind nicht legal in C.

Initialisieren von Strukturen und Arrays von Strukturen

Strukturen und Arrays von Strukturen können durch eine Reihe von Werten, die in geschweifte Klammern eingeschlossen sind, initialisiert werden, ein Wert pro Element der Struktur.

```
struct Date
{
    int year;
    int month;
    int day;
};

struct Date us_independence_day = { 1776, 7, 4 };

struct Date uk_battles[] =
{
    { 1066, 10, 14 },    // Battle of Hastings
    { 1815,  6, 18 },    // Battle of Waterloo
    { 1805, 10, 21 },    // Battle of Trafalgar
};
```

Beachten Sie, dass die Array-Initialisierung ohne die inneren Klammern geschrieben werden könnte und in früheren Zeiten (vor 1990) häufig ohne sie geschrieben worden wäre:

```
struct Date uk_battles[] =
{
    1066, 10, 14,    // Battle of Hastings
    1815,  6, 18,    // Battle of Waterloo
    1805, 10, 21,    // Battle of Trafalgar
};
```

Obwohl dies funktioniert, ist es nicht gut im modernen Stil - Sie sollten nicht versuchen, diese Notation in neuem Code zu verwenden, und sollte die Compiler-Warnungen korrigieren, die normalerweise ausgegeben werden.

Siehe auch benannte Initialisierer .

Verwenden von festgelegten Initialisierern

C99

Mit C99 wurde das Konzept der *vorgesehenen Initialisierer eingeführt.* Damit können Sie festlegen, welche Elemente eines Arrays, einer Struktur oder einer Vereinigung durch die folgenden Werte initialisiert werden sollen.

Designierte Initialisierer für Array-Elemente

Für einen einfachen Typ wie plain `int` :

```
int array[] = { [4] = 29, [5] = 31, [17] = 101, [18] = 103, [19] = 107, [20] = 109 };
```

Der Begriff in eckigen Klammern, bei dem es sich um einen konstanten ganzzahligen Ausdruck handeln kann, gibt an, welches Element des Arrays durch den Wert des Begriffs nach dem `=` - Zeichen initialisiert werden soll. Nicht spezifizierte Elemente werden standardmäßig initialisiert, dh es werden Nullen definiert. Das Beispiel zeigt die angegebenen Initialisierer in Reihenfolge. Sie müssen nicht in Ordnung sein. Das Beispiel zeigt Lücken; das ist legitim. Das Beispiel zeigt nicht zwei verschiedene Initialisierungen für dasselbe Element. das ist auch erlaubt (ISO / IEC 9899: 2011, §6.7.9 Initialisierung, ¶19 *Die Initialisierung erfolgt in der Reihenfolge der Initialisiererliste, wobei jeder Initialisierer, der für ein bestimmtes Unterobjekt vorgesehen ist, alle zuvor aufgelisteten Initialisierer für dasselbe Unterobjekt überschreibt*).

In diesem Beispiel ist die Größe des Arrays nicht explizit definiert. Daher gibt der in den angegebenen Initialisierern angegebene maximale Index die Größe des Arrays an - im Beispiel also 21 Elemente. Wenn die Größe definiert wurde, wäre das Initialisieren eines Eintrags über das Arrayende hinaus wie üblich ein Fehler.

Designierte Initialisierer für Strukturen

Mit dem können Sie angeben, welche Elemente einer Struktur initialisiert werden sollen `. element` :

```
struct Date
{
    int year;
    int month;
    int day;
};

struct Date us_independence_day = { .day = 4, .month = 7, .year = 1776 };
```

Wenn Elemente nicht aufgelistet sind, werden sie standardmäßig initialisiert (auf Null gesetzt).

Designierter Initialisierer für Gewerkschaften

Sie können angeben, welches Element einer Union mit einem bestimmten Initialisierer initialisiert werden soll.

C89

Vor dem C-Standard gab es keine Möglichkeit, eine `union` zu initialisieren. Mit dem Standard C89 / C90 können Sie das erste Mitglied einer `union` initialisieren. Die Auswahl des ersten Mitglieds ist daher wichtig.

```
struct discriminated_union
{
    enum { DU_INT, DU_DOUBLE } discriminant;
    union
    {
        int     du_int;
        double  du_double;
    } du;
};
```

```
struct discriminated_union du1 = { .discriminant = DU_INT, .du = { .du_int = 1 } };
struct discriminated_union du2 = { .discriminant = DU_DOUBLE, .du = { .du_double = 3.14159 }
};
```

C11

Beachten Sie, dass Sie mit C11 anonyme Gewerkschaftsmitglieder innerhalb einer Struktur verwenden können, sodass Sie im vorherigen Beispiel keinen `du` Namen benötigen:

```
struct discriminated_union
{
    enum { DU_INT, DU_DOUBLE } discriminant;
    union
    {
        int     du_int;
        double  du_double;
    };
};

struct discriminated_union du1 = { .discriminant = DU_INT, .du_int = 1 };
struct discriminated_union du2 = { .discriminant = DU_DOUBLE, .du_double = 3.14159 };
```

Designierte Initialisierer für Arrays von Strukturen usw

Diese Konstrukte können für Arrays von Strukturen kombiniert werden, die Elemente enthalten, die Arrays usw. sind. Durch Verwendung vollständiger Sätze von geschweiften Klammern wird sichergestellt, dass die Notation eindeutig ist.

```
typedef struct Date Date;  // See earlier in this example

struct date_range
{
    Date    dr_from;
    Date    dr_to;
    char    dr_what[80];
};

struct date_range ranges[] =
{
    [3] = { .dr_from = { .year = 1066, .month = 10, .day = 14 },
            .dr_to   = { .year = 1066, .month = 12, .day = 25 },
            .dr_what = "Battle of Hastings to Coronation of William the Conqueror",
          },
    [2] = { .dr_from = { .month = 7, .day =  4, .year = 1776 },
            .dr_to   = { .month = 5, .day = 14, .year = 1787 },
            .dr_what = "US Declaration of Independence to Constitutional Convention",
          }
 };
```

Angeben von Bereichen in Array-Initialisierern

GCC bietet eine Erweiterung , mit der Sie einen Bereich von Elementen in einem Array angeben können, für den derselbe Initialisierer angegeben werden sollte:

```
int array[] = { [3 ... 7] = 29, 19 = 107 };
```

Die dreifachen Punkte müssen von den Zahlen getrennt sein, damit nicht einer der Punkte als Teil einer Gleitkommazahl interpretiert wird (*Maximale Munch-* Regel).

Initialisierung online lesen: https://riptutorial.com/de/c/topic/4547/initialisierung

Kapitel 32: Inline-Montage

Bemerkungen

Bei der Inline-Assembly werden in der Mitte des C-Quellcodes Assembly-Anweisungen hinzugefügt. Kein ISO-C-Standard erfordert die Unterstützung der Inline-Montage. Da dies nicht erforderlich ist, variiert die Syntax für die Inline-Assembly von Compiler zu Compiler. Obwohl es normalerweise unterstützt wird, gibt es sehr wenige Gründe für die Inline-Montage und viele Gründe, warum dies nicht der Fall ist.

Pros

1. **Leistung** Durch das Schreiben der spezifischen Assembly-Anweisungen für eine Operation können Sie eine bessere Leistung als der vom Compiler generierte Assembly-Code erzielen. Beachten Sie, dass diese Leistungssteigerungen selten sind. In den meisten Fällen können Sie bessere Leistungssteigerungen erzielen, indem Sie den C-Code so umstellen, dass der Optimierer seine Arbeit erledigen kann.
2. **Hardwareschnittstelle** Gerätetreiber oder Prozessor-Startcode benötigen möglicherweise einen Assembler-Code, um auf die richtigen Register zuzugreifen und um sicherzustellen, dass bestimmte Operationen in einer bestimmten Reihenfolge mit einer bestimmten Verzögerung zwischen den Operationen ausgeführt werden.

Cons

1. **Die Compiler-Portabilitätssyntax** für die Inline-Assembly ist von einem Compiler zu einem anderen nicht garantiert. Wenn Sie Code mit Inline-Assembly schreiben, der von verschiedenen Compilern unterstützt werden soll, verwenden Sie Präprozessor-Makros (`#ifdef`), um zu prüfen, welcher Compiler verwendet wird. Schreiben Sie dann für jeden unterstützten Compiler einen separaten Inline-Assembly-Abschnitt.
2. **Prozessorportabilität** Sie können keine Inline-Assembly für einen x86-Prozessor schreiben und davon ausgehen, dass sie mit einem ARM-Prozessor funktioniert. Inline-Assembly soll für einen bestimmten Prozessor oder eine bestimmte Prozessorfamilie geschrieben werden. Wenn Sie eine Inline-Assembly haben, die auf verschiedenen Prozessoren unterstützt werden soll, überprüfen Sie mithilfe von Präprozessor-Makros, für welchen Prozessor der Code kompiliert wird, und wählen Sie den entsprechenden Assemblycode-Abschnitt aus.
3. **Zukünftige Leistungsänderungen** Inline-Assembly kann mit Verzögerungen basierend auf einer bestimmten Prozessortaktrate geschrieben werden. Wenn das Programm für einen Prozessor mit einer schnelleren Uhr kompiliert wird, funktioniert der Assemblercode möglicherweise nicht wie erwartet.

Examples

gcc Basic ASM-Unterstützung

Grundlegende Assemblierungsunterstützung mit gcc hat die folgende Syntax:

```
asm [ volatile ] ( AssemblerInstructions )
```

Dabei ist `AssemblerInstructions` der direkte Assemblycode für den angegebenen Prozessor. Das volatile-Schlüsselwort ist optional und hat keine Auswirkung, da gcc den Code in einer einfachen asm-Anweisung nicht optimiert. `AssemblerInstructions` können mehrere Montageanweisungen enthalten. Eine grundlegende asm-Anweisung wird verwendet, wenn Sie eine asm-Routine haben, die außerhalb einer C-Funktion vorhanden sein muss. Das folgende Beispiel stammt aus dem GCC-Handbuch:

```
 /* Note that this code will not compile with -masm=intel */
 #define DebugBreak() asm("int $3")
```

In diesem Beispiel könnten Sie dann `DebugBreak()` an anderen Stellen in Ihrem Code verwenden und es wird die Assemblyanweisung `int $3` . Obwohl gcc keinen Code in einer einfachen asm-Anweisung ändert, kann das Optimierungsprogramm dennoch aufeinanderfolgende asm-Anweisungen verschieben. Wenn Sie mehrere Assembly-Anweisungen haben, die in einer bestimmten Reihenfolge vorkommen müssen, fügen Sie sie in einer asm-Anweisung ein.

gcc Extended ASM-Unterstützung

Erweiterte ASM-Unterstützung in gcc hat die folgende Syntax:

```
asm [volatile] ( AssemblerTemplate
                  : OutputOperands
                  [ : InputOperands
                  [ : Clobbers ] ])

 asm [volatile] goto ( AssemblerTemplate
                       :
                       : InputOperands
                       : Clobbers
                       : GotoLabels)
```

wo `AssemblerTemplate` die Vorlage für die Assembler - Befehl ist, `OutputOperands` irgendwelche C - Variablen , die durch die Assemblercode modifiziert werden kann, `InputOperands` irgendwelche C Variablen als Eingangsparameter verwendet, `Clobbers` sind eine Liste oder Register , die durch den Assembler - Code modifiziert sind, und `GotoLabels` sind alle goto-Anweisungsetiketten, die im Assemblycode verwendet werden können.

Das erweiterte Format wird innerhalb von C-Funktionen verwendet und ist die typischere Verwendung von Inline-Baugruppen. Im Folgenden finden Sie ein Beispiel aus dem Linux-Kernel für den Byte-Austausch von 16-Bit- und 32-Bit-Nummern für einen ARM-Prozessor:

```
/* From arch/arm/include/asm/swab.h in Linux kernel version 4.6.4 */
#if __LINUX_ARM_ARCH__ >= 6

static inline __attribute_const__ __u32 __arch_swahb32(__u32 x)
{
```

```
    __asm__ ("rev16 %0, %1" : "=r" (x) : "r" (x));
    return x;
}
#define __arch_swahb32 __arch_swahb32
#define __arch_swab16(x) ((__u16)__arch_swahb32(x))

static inline __attribute_const__ __u32 __arch_swab32(__u32 x)
{
    __asm__ ("rev %0, %1" : "=r" (x) : "r" (x));
    return x;
}
#define __arch_swab32 __arch_swab32

#endif
```

Jeder asm-Abschnitt verwendet die Variable `x` als Eingabe- und Ausgabeparameter. Die C-Funktion gibt dann das manipulierte Ergebnis zurück.

Mit dem erweiterten asm-Format kann gcc die Assembly-Anweisungen in einem asm-Block nach denselben Regeln optimieren, die es für die Optimierung von C-Code verwendet. Wenn Sie möchten, dass Ihr ASM-Abschnitt unberührt bleibt, verwenden Sie das `volatile` Schlüsselwort für den ASM-Abschnitt.

gcc Inline-Montage in Makros

Wir können Montageanweisungen in ein Makro einfügen und das Makro so verwenden, als würden Sie eine Funktion aufrufen.

```
#define mov(x,y) \
{ \
    __asm__ ("l.cmov %0,%1,%2" : "=r" (x) : "r" (y), "r" (0x0000000F)); \
}

/// some definition and assignment
unsigned char sbox[size][size];
unsigned char sbox[size][size];

///Using
mov(state[0][1], sbox[si][sj]);
```

Die Verwendung von Inline-Assembly-Anweisungen, die in C-Code eingebettet sind, kann die Laufzeit eines Programms verbessern. Dies ist in zeitkritischen Situationen wie kryptografischen Algorithmen wie AES sehr hilfreich. Für einen einfachen Verschiebevorgang, der im AES-Algorithmus erforderlich ist, können Sie beispielsweise eine direkte Anweisung für die `Rotate Right` durch den C-Verschiebungsoperator `>>` ersetzen.

In einer Implementierung von 'AES256' haben wir in der Funktion 'AddRoundKey ()' einige Anweisungen wie diese:

```
unsigned int w;          // 32-bit
unsigned char subkey[4]; // 8-bit, 4*8 = 32

subkey[0] = w >> 24;     // hold 8 bit, MSB, leftmost group of 8-bits
```

```
subkey[1] = w >> 16;      // hold 8 bit, second group of 8-bit from left
subkey[2] = w >> 8;       // hold 8 bit, second group of 8-bit from right
subkey[3] = w;            // hold 8 bit, LSB, rightmost group of 8-bits

/// subkey <- w
```

Sie weisen einfach den Bitwert von `w` dem `subkey` .

Wir können drei Verschiebungs-, Zuweisungs- und einen Zuweisungs-C-Ausdruck mit nur einer Baugruppe `Rotate Right` Operation ändern.

```
__asm__ ("l.ror  %0,%1,%2" : "=r" (* (unsigned int *) subkey)  : "r" (w), "r" (0x10));
```

Das Endergebnis ist genau das gleiche.

Inline-Montage online lesen: https://riptutorial.com/de/c/topic/4263/inline-montage

Kapitel 33: Inlining

Examples

Inlining-Funktionen, die in mehreren Quelldateien verwendet werden

Bei kleinen Funktionen, die häufig aufgerufen werden, kann der mit dem Funktionsaufruf verbundene Aufwand einen erheblichen Bruchteil der gesamten Ausführungszeit dieser Funktion ausmachen. Eine Möglichkeit, die Leistung zu verbessern, besteht darin, den Overhead zu beseitigen.

In diesem Beispiel verwenden wir vier Funktionen (plus `main()`) in drei Quelldateien. Zwei davon (`plusfive()` und `timestwo()`) werden jeweils von den anderen beiden in "source1.c" und "source2.c" aufgerufen. Das `main()` ist enthalten, so dass wir ein Arbeitsbeispiel haben.

Haupt c:

```
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int main(void) {
    int start = 3;
    int intermediate = complicated1(start);
    printf("First result is %d\n", intermediate);
    intermediate = complicated2(start);
    printf("Second result is %d\n", intermediate);
    return 0;
}
```

source1.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int complicated1(int input) {
    int tmp = timestwo(input);
    tmp = plusfive(tmp);
    return tmp;
}
```

source2.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int complicated2(int input) {
    int tmp = plusfive(input);
```

```
    tmp = timestwo(tmp);
    return tmp;
}
```

Headerdatei.h:

```
#ifndef HEADERFILE_H
#define HEADERFILE_H

int complicated1(int input);
int complicated2(int input);

inline int timestwo(int input) {
  return input * 2;
}
inline int plusfive(int input) {
  return input + 5;
}

#endif
```

Die Funktionen `timestwo` und `plusfive` werden von sowohl `complicated1` als auch `complicated2` aufgerufen, die sich in unterschiedlichen "Übersetzungseinheiten" oder Quelldateien befinden. Um sie auf diese Weise verwenden zu können, müssen wir sie im Header definieren.

Kompilieren Sie wie folgt, vorausgesetzt, gcc:

```
cc -O2 -std=c99 -c -o main.o main.c
cc -O2 -std=c99 -c -o source1.o source1.c
cc -O2 -std=c99 -c -o source2.o source2.c
cc main.o source1.o source2.o -o main
```

Wir verwenden die -O2-Optimierungsoption, da einige Compiler nicht ohne eingeschaltete Optimierung inline sind.

Das `inline` Schlüsselwort bewirkt, dass das betreffende Funktionssymbol nicht in die Objektdatei ausgegeben wird. Andernfalls würde in der letzten Zeile ein Fehler auftreten, in dem die Objektdateien zur endgültigen ausführbaren Datei verknüpft werden. Wenn wir keine `inline` , würde dasselbe Symbol in beiden `.o` Dateien definiert, und es würde ein Fehler "mehrfach definiertes Symbol" auftreten.

In Situationen, in denen das Symbol tatsächlich benötigt wird, hat dies den Nachteil, dass das Symbol überhaupt nicht erzeugt wird. Hierfür gibt es zwei Möglichkeiten. Die erste ist das Hinzufügen einer zusätzlichen `extern` Deklaration der eingebetteten Funktionen in genau einer der `.c` Dateien. `source1.c` zu `source1.c` folgendes `source1.c` :

```
extern int timestwo(int input);
extern int plusfive(int input);
```

Die andere Möglichkeit besteht darin, die Funktion mit `static inline` statt mit `inline` zu definieren. Diese Methode hat den Nachteil, dass in **jeder** Objektdatei, die mit diesem Header erstellt wird,

möglicherweise eine Kopie der fraglichen Funktion erstellt wird.

Inlining online lesen: https://riptutorial.com/de/c/topic/7427/inlining

Kapitel 34: Iterationsanweisungen / -schleifen: für, während, währenddessen

Syntax

- /* alle Versionen */
- für ([Ausdruck]; [Ausdruck]; [Ausdruck]) eine_Anweisung
- for ([Ausdruck]; [Ausdruck]; [Ausdruck]) {null oder mehrere Anweisungen}
- while (Ausdruck) one_statement
- while (Ausdruck) {null oder mehrere Anweisungen}
- one_statement while (Ausdruck);
- do {eine oder mehrere Anweisungen} while (Ausdruck);
- // seit C99 zusätzlich zum obigen Formular
- for (Deklaration; [Ausdruck]; [Ausdruck]) one_statement;
- for (Deklaration; [Ausdruck]; [Ausdruck]) {null oder mehrere Anweisungen}

Bemerkungen

Iteration Statement / Loops lassen sich in zwei Kategorien einteilen:

- kopfgesteuerte Iterationsanweisung / -schleifen
- fußgesteuerte Iterationsanweisung / -schleifen

Kopfgesteuerte Iterationsanweisung / -schleifen

```
for ([<expression>]; [<expression>]; [<expression>]) <statement>
while (<expression>) <statement>
```

C99

```
for ([declaration expression]; [expression] [; [expression]]) statement
```

Fußgesteuerte Iterationsanweisung / -schleifen

```
do <statement> while (<expression>);
```

Examples

Für Schleife

Um einen Codeblock immer wieder auszuführen, kommen Schleifen ins Bild. Die `for` Schleife ist zu verwenden, wenn ein Codeblock eine festgelegte Anzahl von Malen ausgeführt werden soll.

Um zum Beispiel ein Array der Größe `n` mit den Benutzereingaben zu füllen, müssen wir `scanf()` `n` mal ausführen.

C99

```
#include <stddef.h>          // for size_t

int array[10];               // array of 10 int

for (size_t i = 0; i < 10; i++)    // i starts at 0 and finishes with 9
{
   scanf("%d", &array[i]);
}
```

Auf diese Weise wird der `scanf()` Funktion `scanf()` `n` mal (in unserem Beispiel 10-mal) ausgeführt, jedoch nur einmal geschrieben.

Hier ist die Variable `i` der Schleifenindex und wird am besten als dargestellt deklariert. Der Typ `size_t` (*size type*) sollte für alles verwendet werden, was Datenobjekte zählt oder durchläuft.

Diese Möglichkeit, Variablen innerhalb von `for` deklarieren, ist nur für Compiler verfügbar, die auf den C99-Standard aktualisiert wurden. Wenn Sie aus irgendeinem Grund immer noch bei einem älteren Compiler stecken, können Sie den Schleifenindex vor der `for` Schleife angeben:

C99

```
#include <stddef.h>        /* for size_t */
size_t i;
int array[10];             /* array of 10 int */

for (i = 0; i < 10; i++)       /* i starts at 0 and finishes at 9 */
{
   scanf("%d", &array[i]);
}
```

While-Schleife

Eine `while` Schleife wird verwendet, um einen Code auszuführen, während eine Bedingung erfüllt ist. Die `while` Schleife ist zu verwenden, wenn ein Codeblock mehrmals variabel ausgeführt werden soll. Der angezeigte Code erhält beispielsweise die Benutzereingabe, solange der Benutzer Zahlen einfügt, die nicht `0` . Wenn der Benutzer `0` einfügt, ist die while-Bedingung nicht mehr wahr, sodass die Ausführung die Schleife verlässt und mit dem nachfolgenden Code fortgesetzt wird:

```
int num = 1;

while (num != 0)
{
    scanf("%d", &num);
}
```

Do-While-Schleife

Im Gegensatz zu `for` und `while` `do-while` Schleifen prüfen `do-while` Schleifen die Wahrheit der Bedingung am Ende der Schleife. Dies bedeutet, dass der `do` Block einmal ausgeführt wird, und dann den Zustand des `while` am unteren Rand des Blocks. Das bedeutet, dass eine `do-while` Schleife *immer* mindestens einmal ausgeführt wird.

Diese `do-while` Schleife erhält beispielsweise Zahlen vom Benutzer, bis die Summe dieser Werte größer oder gleich `50` :

```
int num, sum;
num = sum = 0;

do
{
  scanf("%d", &num);
  sum += num;

} while (sum < 50);
```

`do-while` Loops sind bei den meisten Programmierstilen relativ selten.

Struktur und Ablauf der Kontrolle in einer for-Schleife

```
for ([declaration-or-expression]; [expression2]; [expression3])
{
    /* body of the loop */
}
```

In einer `for` Schleife hat die Schleifenbedingung drei optionale Ausdrücke.

- Der erste Ausdruck, `declaration-or-expression` , *initialisiert* die Schleife. Es wird genau einmal am Anfang der Schleife ausgeführt.

C99

Dies kann entweder eine Deklaration und Initialisierung einer Schleifenvariablen oder ein allgemeiner Ausdruck sein. Wenn es sich um eine Deklaration handelt, wird der Gültigkeitsbereich der deklarierten Variablen durch die `for` Anweisung eingeschränkt.

C99

Historische Versionen von C erlaubten hier nur einen Ausdruck, und die Deklaration einer Schleifenvariablen musste vor dem `for` .

- Der zweite Ausdruck, `expression2` , ist die *Testbedingung* . Sie wird erst nach der Initialisierung ausgeführt. Wenn die Bedingung `true` , dann tritt die Steuerung in den Körper der Schleife. Wenn nicht, verschiebt es sich am Ende der Schleife nach außerhalb des Schleifenrumpfes. Anschließend wird diese Bedingung nach jeder Ausführung des Hauptteils sowie der Aktualisierungsanweisung überprüft. Wenn dies `true` , `true` das Steuerelement zurück zum Anfang des Loop-Körpers. Die Bedingung soll in der Regel eine Überprüfung der Anzahl der Male, die der Body der Schleife ausführt, sein. Dies ist die primäre Möglichkeit, eine Schleife zu verlassen, die andere Möglichkeit besteht darin,

Sprunganweisungen zu verwenden .
- Der dritte Ausdruck, `expression3` , ist die *Aktualisierungsanweisung* . Sie wird nach jeder Ausführung des Hauptteils der Schleife ausgeführt. Sie wird häufig verwendet, um eine Variable zu erhöhen, die die Anzahl der Ausführungen des Schleifenkörpers zählt, und diese Variable wird *Iterator genannt* .

Jede Ausführung des Schleifenkörpers wird als *Iteration bezeichnet* .

Beispiel:

C99

```
for(int i = 0; i < 10 ; i++)
{
    printf("%d", i);
}
```

Die Ausgabe ist:

```
0123456789
```

In dem obigen Beispiel wird zuerst `i = 0` ausgeführt, wobei `i` initialisiert wird. Dann wird die Bedingung `i < 10` geprüft, die als `true` bewertet wird. Das Steuerelement geht in den Körper der Schleife und der Wert von `i` wird gedruckt. Dann wechselt die Steuerung zu `i++` , wobei der Wert von `i` von 0 auf 1 aktualisiert wird. Dann wird die Bedingung erneut geprüft und der Prozess wird fortgesetzt. Dies setzt sich fort, bis der Wert von `i` 10 wird. Dann wird die Bedingung `i < 10 false` ausgewertet, wonach die Steuerung die Schleife verlässt.

Unendliche Schleifen

Eine Schleife wird als eine *unendliche Schleife bezeichnet,* wenn die Steuerung in den Kreis der Schleife eintritt, diese jedoch nie verlässt. Dies geschieht, wenn die Testbedingung der Schleife niemals als `false` bewertet wird.

Beispiel:

C99

```
for (int i = 0; i >= 0; )
{
    /* body of the loop where i is not changed*/
}
```

Im obigen Beispiel wird die Variable `i` , der Iterator, auf 0 initialisiert. Die Testbedingung ist anfangs `true` . `i` wird jedoch an keiner Stelle im Hauptteil geändert und der Aktualisierungsausdruck ist leer. Daher `i` wird 0 bleiben, und die Testbedingung wird nie bewerten `false` , zu einer Endlosschleife führt.

Angenommen, es gibt keine Sprunganweisungen, kann die Endlosschleife auch dadurch gebildet werden, dass die Bedingung explizit wahr bleibt:

```
while (true)
{
    /* body of the loop */
}
```

In einer `for` Schleife ist die Bedingungsanweisung optional. In diesem Fall ist die Bedingung immer `true` vacuously, zu einer Endlosschleife führt.

```
for (;;)
{
    /* body of the loop */
}
```

Doch in bestimmten Fällen kann der Zustand gehalten wird `true` absichtlich, mit der Absicht, die Schleife verlass eine mit Sprunganweisung wie `break` .

```
while (true)
{
    /* statements */
    if (condition)
    {
         /* more statements */
         break;
    }
}
```

Endlosschleifen und Duff's Device

Manchmal kann die direkte Schleife nicht vollständig im Schleifenkörper enthalten sein. Dies liegt daran, dass die Schleife durch einige Anweisungen ***B*** vorbereitet werden muss. Dann beginnt die Iteration mit einigen Anweisungen ***A*** , denen vor der Schleife erneut ***B*** folgt.

```
do_B();
while (condition) {
    do_A();
    do_B();
}
```

Um potenzielle Probleme beim Ausschneiden / Einfügen bei der zweimaligen Wiederholung von ***B*** im Code zu vermeiden, kann Duff's Device angewendet werden, um die Schleife von der Mitte des `while` Körpers aus mit einer switch-Anweisung zu starten und das Verhalten zu beeinflussen.

```
switch (true) while (condition) {
case false: do_A(); /* FALL THROUGH */
default:    do_B(); /* FALL THROUGH */
}
```

Duff's Device wurde eigentlich erfunden, um die Schleifenabwicklung zu implementieren. Stellen Sie sich vor, Sie wenden eine Maske auf einen Speicherblock an, wobei `n` ein vorzeichenbehafteter Integraltyp mit einem positiven Wert ist.

```
do {
    *ptr++ ^= mask;
} while (--n > 0);
```

Wenn `n` immer durch 4 teilbar wäre, könnten Sie dies leicht abwickeln als:

```
do {
    *ptr++ ^= mask;
    *ptr++ ^= mask;
    *ptr++ ^= mask;
    *ptr++ ^= mask;
} while ((n -= 4) > 0);
```

Mit Duff's Device kann der Code jedoch dieser Abwicklungsform folgen, die in der Mitte der Schleife an die richtige Stelle springt, wenn `n` nicht durch 4 teilbar ist.

```
switch (n % 4) do {
case 0: *ptr++ ^= mask; /* FALL THROUGH */
case 3: *ptr++ ^= mask; /* FALL THROUGH */
case 2: *ptr++ ^= mask; /* FALL THROUGH */
case 1: *ptr++ ^= mask; /* FALL THROUGH */
} while ((n -= 4) > 0);
```

Diese Art des manuellen Abrollens ist bei modernen Compilern selten erforderlich, da die Optimierungs-Engine des Compilers Schleifen im Namen des Programmierers abwickeln kann.

Iterationsanweisungen / -schleifen: für, während, währenddessen online lesen: https://riptutorial.com/de/c/topic/5151/iterationsanweisungen----schleifen--fur--wahrend--wahrenddessen

Kapitel 35: Kommandozeilenargumente

Syntax

- int main (int argc, char * argv [])

Parameter

Parameter	Einzelheiten
argc	argument count - wird mit der Anzahl der durch Leerzeichen getrennten Argumente initialisiert, die das Programm über die Befehlszeile erhält, sowie den Programmnamen.
argv	Argument vector - initialisiert auf ein Array von `char` -pointers (strings) enthält , die Argumente (und den Programmnamen) , die auf der Befehlszeile angegeben wurden.

Bemerkungen

AC - Programm in einer ‚gehosteten Umgebung' (die normale Art - im Gegensatz zu einer ‚freistehenden Umwelt') ausgeführt wird, muss eine hat `main` Es wird traditionell definiert als:

```
int main(int argc, char *argv[])
```

Beachten Sie, dass `argv` auch als `char **argv` definiert werden kann und sehr oft ist. Das Verhalten ist das gleiche. Die Parameternamen können auch geändert werden, da sie nur lokale Variablen in der Funktion sind. `argc` und `argv` sind jedoch üblich und Sie sollten diese Namen verwenden.

Verwenden Sie für `main` , bei denen der Code keine Argumente verwendet, `int main(void)` .

Beide Parameter werden beim Programmstart initialisiert:

- `argc` wird auf die Anzahl der durch Leerzeichen getrennten Argumente, die dem Programm von der Befehlszeile aus gegeben werden, sowie den Programmnamen selbst initialisiert.
- `argv` ist ein Array von `char` -pointers (strings), die die Argumente (und den Programmnamen) enthalten, die in der Befehlszeile angegeben wurden.
- Einige Systeme erweitern Befehlszeilenargumente "in der Shell", andere nicht. Wenn der Benutzer unter Unix `myprogram *.txt` Programm `myprogram *.txt` , erhält das Programm eine Liste von Textdateien. Unter Windows erhält es die Zeichenfolge " `*.txt` ".

Hinweis: Vor der Verwendung von `argv` müssen Sie möglicherweise den Wert von `argc` überprüfen. In der Theorie könnte `argc 0` , und wenn `argc` Null ist, gibt es keine Argumente, und `argv[0]` (entspricht `argv[argc]`) ist ein Nullzeiger. Es wäre ein ungewöhnliches System mit einer

gehosteten Umgebung, wenn dieses Problem auftritt. Ebenso ist es möglich, wenn auch sehr ungewöhnlich, keine Informationen über den Programmnamen zu erhalten. In diesem Fall ist `argv[0][0] == '\0'` - der Programmname ist möglicherweise leer.

Angenommen, wir starten das Programm folgendermaßen:

```
./some_program abba banana mamajam
```

Dann ist `argc` gleich `4` und die Befehlszeilenargumente:

- `argv[0]` zeigt auf `"./some_program"` (den Programmnamen), wenn der Programmname in der `"./some_program"` verfügbar ist. Ansonsten eine leere Zeichenfolge `""` .
- `argv[1]` verweist auf `"abba"` ,
- `argv[2]` verweist auf `"banana"` ,
- `argv[3]` verweist auf `"mamajam"` ,
- `argv[4]` enthält den Wert `NULL` .

Siehe auch Was sollte `main()` in C und C ++ für vollständige Zitate aus dem Standard zurückgeben.

Examples

Befehlszeilenargumente drucken

Nachdem Sie die Argumente erhalten haben, können Sie sie wie folgt drucken:

```
int main(int argc, char **argv)
{
    for (int i = 1; i < argc; i++)
    {
        printf("Argument %d: [%s]\n", i, argv[i]);
    }
}
```

Anmerkungen

1. Der Parameter `argv` kann auch als `char *argv[]` .
2. `argv[0]` *kann* den Programmnamen selbst enthalten (abhängig davon, wie das Programm ausgeführt wurde). Das erste "echte" Befehlszeilenargument lautet `argv[1]` . Dies ist der Grund, warum die Schleifenvariable `i` auf 1 initialisiert wird.
3. In der print-Anweisung können Sie `*(argv + i)` anstelle von `argv[i]` - es wird dasselbe ausgewertet, ist jedoch ausführlicher.
4. Die eckigen Klammern um den Argumentwert helfen, den Anfang und das Ende zu identifizieren. Dies kann von unschätzbarem Wert sein, wenn das Argument nachstehende Leerzeichen, Zeilenumbrüche, Zeilenumbrüche oder andere Sonderzeichen enthält. Einige Varianten dieses Programms sind ein nützliches Werkzeug zum Debuggen von Shell-Skripts, bei denen Sie wissen müssen, was die Argumentliste tatsächlich enthält (obwohl es einfache Shell-Alternativen gibt, die fast gleichwertig sind).

Drucken Sie die Argumente an ein Programm und konvertieren Sie sie in Ganzzahlwerte

Der folgende Code gibt die Argumente an das Programm aus und der Code versucht, jedes Argument in eine Zahl (in eine `long` Zahl) umzuwandeln:

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <limits.h>

int main(int argc, char* argv[]) {

    for (int i = 1; i < argc; i++) {
        printf("Argument %d is: %s\n", i, argv[i]);

        errno = 0;
        char *p;
        long argument_numValue = strtol(argv[i], &p, 10);

        if (p == argv[i]) {
            fprintf(stderr, "Argument %d is not a number.\n", i);
        }
        else if ((argument_numValue == LONG_MIN || argument_numValue == LONG_MAX) && errno ==
ERANGE) {
            fprintf(stderr, "Argument %d is out of range.\n", i);
        }
        else {
            printf("Argument %d is a number, and the value is: %ld\n",
                   i, argument_numValue);
        }
    }
    return 0;
}
```

VERWEISE:

- strtol () gibt einen falschen Wert zurück
- Richtige Verwendung von strtol

Verwenden von GNU-getopt-Werkzeugen

Befehlszeilenoptionen für Anwendungen werden von der C-Sprache nicht anders als Befehlszeilenargumente behandelt. Sie sind nur Argumente, die in einer Linux- oder Unix-Umgebung traditionell mit einem Bindestrich (`-`) beginnen.

Mit glibc in einer Linux- oder Unix-Umgebung können Sie die getopt-Tools verwenden, um auf einfache Weise Befehlszeilenoptionen für die restlichen Argumente zu definieren, zu überprüfen und zu analysieren.

Diese Tools erwarten, dass Ihre Optionen gemäß den GNU-Codierungsstandards formatiert sind. Dies ist eine Erweiterung der von POSIX für das Format der Befehlszeilenoptionen festgelegten Optionen.

Das folgende Beispiel veranschaulicht die Handhabung von Befehlszeilenoptionen mit den GNU-getopt-Werkzeugen.

```
#include <stdio.h>
#include <getopt.h>
#include <string.h>

/* print a description of all supported options */
void usage (FILE *fp, const char *path)
{
    /* take only the last portion of the path */
    const char *basename = strrchr(path, '/');
    basename = basename ? basename + 1 : path;

    fprintf (fp, "usage: %s [OPTION]\n", basename);
    fprintf (fp, "  -h, --help\t\t"
                 "Print this help and exit.\n");
    fprintf (fp, "  -f, --file[=FILENAME]\t"
                 "Write all output to a file (defaults to out.txt).\n");
    fprintf (fp, "  -m, --msg=STRING\t"
                 "Output a particular message rather than 'Hello world'.\n");
}

/* parse command-line options and print message */
int main(int argc, char *argv[])
{
    /* for code brevity this example just uses fixed buffer sizes for strings */
    char filename[256] = { 0 };
    char message[256] = "Hello world";
    FILE *fp;
    int help_flag = 0;
    int opt;

    /* table of all supported options in their long form.
     * fields: name, has_arg, flag, val
     * `has_arg` specifies whether the associated long-form option can (or, in
     * some cases, must) have an argument. the valid values for `has_arg` are
     * `no_argument`, `optional_argument`, and `required_argument`.
     * if `flag` points to a variable, then the variable will be given a value
     * of `val` when the associated long-form option is present at the command
     * line.
     * if `flag` is NULL, then `val` is returned by `getopt_long` (see below)
     * when the associated long-form option is found amongst the command-line
     * arguments.
     */
    struct option longopts[] = {
        { "help", no_argument, &help_flag, 1 },
        { "file", optional_argument, NULL, 'f' },
        { "msg", required_argument, NULL, 'm' },
        { 0 }
    };

    /* infinite loop, to be broken when we are done parsing options */
    while (1) {
        /* getopt_long supports GNU-style full-word "long" options in addition
         * to the single-character "short" options which are supported by
         * getopt.
         * the third argument is a collection of supported short-form options.
         * these do not necessarily have to correlate to the long-form options.
         * one colon after an option indicates that it has an argument, two
```

```
         * indicates that the argument is optional. order is unimportant.
         */
        opt = getopt_long (argc, argv, "hf::m:", longopts, 0);

        if (opt == -1) {
            /* a return value of -1 indicates that there are no more options */
            break;
        }

        switch (opt) {
        case 'h':
            /* the help_flag and value are specified in the longopts table,
             * which means that when the --help option is specified (in its long
             * form), the help_flag variable will be automatically set.
             * however, the parser for short-form options does not support the
             * automatic setting of flags, so we still need this code to set the
             * help_flag manually when the -h option is specified.
             */
            help_flag = 1;
            break;
        case 'f':
            /* optarg is a global variable in getopt.h. it contains the argument
             * for this option. it is null if there was no argument.
             */
            printf ("outarg: '%s'\n", optarg);
            strncpy (filename, optarg ? optarg : "out.txt", sizeof (filename));
            /* strncpy does not fully guarantee null-termination */
            filename[sizeof (filename) - 1] = '\0';
            break;
        case 'm':
            /* since the argument for this option is required, getopt guarantees
             * that aptarg is non-null.
             */
            strncpy (message, optarg, sizeof (message));
            message[sizeof (message) - 1] = '\0';
            break;
        case '?':
            /* a return value of '?' indicates that an option was malformed.
             * this could mean that an unrecognized option was given, or that an
             * option which requires an argument did not include an argument.
             */
            usage (stderr, argv[0]);
            return 1;
        default:
            break;
        }
    }

    if (help_flag) {
        usage (stdout, argv[0]);
        return 0;
    }

    if (filename[0]) {
        fp = fopen (filename, "w");
    } else {
        fp = stdout;
    }

    if (!fp) {
        fprintf(stderr, "Failed to open file.\n");
```

```
        return 1;
    }

    fprintf (fp, "%s\n", message);

    fclose (fp);
    return 0;
}
```

Es kann mit `gcc` kompiliert werden:

```
gcc example.c -o example
```

Es unterstützt drei Befehlszeilenoptionen (`--help` , `--file` und `--msg`). Alle haben auch eine "Kurzform" (`-h` , `-f` und `-m`). Die Optionen "Datei" und "msg" akzeptieren beide Argumente. Wenn Sie die Option "msg" angeben, ist dessen Argument erforderlich.

Argumente für Optionen werden wie folgt formatiert:

- `--option=value` (für Langformoptionen)
- `-ovalue` oder `-o"value"` (für Kurzformoptionen)

Kommandozeilenargumente online lesen:
https://riptutorial.com/de/c/topic/1285/kommandozeilenargumente

Kapitel 36: Literale für Zahlen, Zeichen und Zeichenfolgen

Bemerkungen

Der Begriff **Literal** wird allgemein verwendet, um eine Folge von Zeichen in einem C-Code zu beschreiben, die einen konstanten Wert wie eine Zahl (z. B. `0`) oder eine Zeichenfolge (z. B. `"C"`) bezeichnet. Genau genommen verwendet der Standard die **Termkonstante** für Ganzzahlkonstanten, Fließkonstanten, Aufzählungskonstanten und Zeichenkonstanten, wobei der Begriff 'Literal' für String-Literale reserviert wird.

Literale können **Präfixe** oder **Suffixe** (aber nicht beide) enthalten. Dies sind zusätzliche Zeichen, die ein Literal starten oder beenden können, um seinen Standardtyp oder seine Darstellung zu ändern.

Examples

Ganzzahlige Literale

Ganzzahlliterale werden verwendet, um ganzzahlige Werte bereitzustellen. Es werden drei numerische Grundlagen unterstützt, die durch Präfixe gekennzeichnet sind:

Base	Präfix	Beispiel
Dezimal	Keiner	`5`
Oktal	`0`	`0345`
Hexadezimal	`0x` oder `0X`	`0x12AB` , `0X12AB` , `0x12ab` , `0x12Ab`

Beachten Sie, dass diese Schreibweise kein Vorzeichen enthält. Ganzzahlige Literale sind daher immer positiv. So etwas wie `-1` wird als Ausdruck behandelt, der ein ganzzahliges Literal (`1`) hat, das mit einem `-` negiert wird.

Der Typ eines Dezimal-Integer-Literal ist der erste Datentyp, der den Wert von `int` und `long` passen kann. `long long` wird seit C99 auch für sehr große Literale unterstützt.

Der Typ eines oktalen oder hexadezimalen Integer-Literal ist der erste Datentyp, der den Wert von `int` , `unsigned` , `long` und `unsigned long` . `long long` und `unsigned long long` werden seit C99 auch für sehr große Literale unterstützt.

Mit verschiedenen Suffixen kann der Standardtyp eines Literal geändert werden.

Suffix	Erläuterung
`L` , `l`	`long int`
`LL` , `ll` (seit C99)	`long long int`
`U` , `u`	`unsigned`

Die Suffixe U und L / LL können in beliebiger Reihenfolge und Fall kombiniert werden. Es ist ein Fehler, Suffixe zu duplizieren (z. B. zwei `U` Suffixe anzugeben), auch wenn diese unterschiedliche Fälle haben.

String-Literale

String-Literale werden verwendet, um Zeichenarrays anzugeben. Sie sind Zeichenfolgen, die in doppelte Anführungszeichen eingeschlossen sind (z. B. `"abcd"` und haben den Typ `char*`).

Das Präfix `L` macht das Literal zu einem breiten Zeichenfeld vom Typ `wchar_t*` . Zum Beispiel `L"abcd"` .

Seit C11 gibt es andere Codierungspräfixe, ähnlich wie `L` :

Präfix	Basistyp	Codierung
keiner	`char`	Plattformabhängig
`L`	`wchar_t`	Plattformabhängig
`u8`	`char`	UTF-8
`u`	`char16_t`	normalerweise UTF-16
`U`	`char32_t`	normalerweise UTF-32

Für die letzten beiden kann mit Feature-Test-Makros abgefragt werden, ob die Kodierung tatsächlich die entsprechende UTF-Kodierung ist.

Fließkomma-Literale

Fließkomma-Literale werden verwendet, um vorzeichenbehaftete reelle Zahlen darzustellen. Die folgenden Suffixe können verwendet werden, um den Typ eines Literal anzugeben:

Suffix	Art	Beispiele
keiner	`double`	`3.1415926` `-3E6`
`f` , `F`	`float`	`3.1415926f` `2.1E-6F`
`l` , `L`	`long double`	`3.1415926L` `1E126L`

Um diese Suffixe verwenden zu können, *muss* das Literal ein Fließkomma-Literal sein. Beispielsweise ist `3f` ein Fehler, da `3` ein ganzzahliges Literal ist, während `3.f` oder `3.0f` korrekt sind. Bei einem `long double` wird empfohlen, zur besseren Lesbarkeit immer das Kapital `L` zu verwenden.

Zeichenliterale

Zeichenliterale sind ein spezieller Typ von Ganzzahl-Literalen, die zur Darstellung eines Zeichens verwendet werden. Sie sind in einfache Anführungszeichen eingeschlossen, z. B. `'a'` und haben den Typ ' `int` . Der Wert des Literal ist ein ganzzahliger Wert, der dem Zeichensatz der Maschine entspricht. Sie erlauben keine Suffixe.

Das Präfix `L` vor einem Zeichenliteral macht es zu einem breiten Zeichen vom Typ `wchar_t` . Ebenso machen die Präfixe `u` und `U` seit C11 breite Zeichen vom Typ `char16_t` bzw. `char32_t` .

Bei der Darstellung bestimmter Sonderzeichen, beispielsweise eines nicht druckbaren Zeichens, werden Escape-Sequenzen verwendet. Escape-Sequenzen verwenden eine Folge von Zeichen, die in ein anderes Zeichen übersetzt werden. Alle Escape - Sequenzen bestehen aus zwei oder mehreren Zeichen, von denen die erste ist ein Backslash `\` . Die Zeichen unmittelbar nach dem Backslash legen fest, unter welchem Zeichenliteral die Sequenz interpretiert wird.

Fluchtabfolge	Dargestellter Charakter
`\b`	Rücktaste
`\f`	Formularvorschub
`\n`	Zeilenvorschub (neue Zeile)
`\r`	Wagenrücklauf
`\t`	Horizontale Registerkarte
`\v`	Vertikale Registerkarte
`\\`	Backslash
`\'`	Einfaches Anführungszeichen
`\"`	Doppelte Anführungszeichen
`\?`	Fragezeichen
`\nnn`	Oktalwert
`\xnn` ...	Hexadezimalwert

C89

Fluchtabfolge	Dargestellter Charakter
`\a`	Alarm (Piepton, Glocke)

C99

Fluchtabfolge	Dargestellter Charakter
`\unnnn`	Universeller Zeichenname
`\Unnnnnnnn`	Universeller Zeichenname

Ein universeller Zeichenname ist ein Unicode-Codepunkt. Ein universeller Zeichenname kann mehreren Zeichen zugeordnet werden. Die Ziffern `n` werden als Hexadezimalziffern interpretiert. Je nach UTF - Codierung verwendet wird , in einem Codepunkt eine universelle Zeichennamensfolge führen kann, die aus mehreren Zeichen bestehen, anstelle einem einzigen normalen `char` Charakter.

Bei Verwendung der Zeilenvorschub-Escape-Sequenz in der Textmodus-E / A wird sie in die betriebssystemspezifische Zeilen- oder Bytefolge von Zeilenumbrüchen konvertiert.

Die Fragezeichen-Escape-Sequenz wird verwendet, um Trigraphen zu vermeiden. Zum Beispiel wird `??/` als Trigraph kompiliert, der ein Backslash-Zeichen `'\'` , aber bei Verwendung von `?\?/` Würde die *Zeichenfolge* `"??/"` .

Es können eine, zwei oder drei Oktalziffern `n` in der Escape-Folge des Oktalwerts stehen.

Literale für Zahlen, Zeichen und Zeichenfolgen online lesen: https://riptutorial.com/de/c/topic/3455/literale-fur-zahlen--zeichen-und-zeichenfolgen

Kapitel 37: Multithreading

Einführung

In C11 gibt es eine Standard-Thread-Bibliothek `<threads.h>` , jedoch keinen bekannten Compiler, der sie noch implementiert. `pthread.h` Multithreading in C zu verwenden, müssen Sie daher plattformspezifische Implementierungen verwenden, z. B. die POSIX-Threads-Bibliothek (häufig als pthreads bezeichnet), die `pthread.h` Header `pthread.h` .

Syntax

- thrd_t // Implementierungsdefinierter vollständiger Objekttyp, der einen Thread identifiziert
- int thrd_create (thrd_t * thr, thrd_start_t func, void * arg); // Erstellt einen Thread
- int thrd_equal (thrd_t thr0, thrd_t thr1); // Prüfen Sie, ob sich Argumente auf denselben Thread beziehen
- thr_t thrd_current (void); // Gibt den Bezeichner des Threads zurück, der ihn aufruft
- int thrd_sleep (const struct Zeitangabe * Dauer, Struktur Zeitangabe * verbleibend); // Die Ausführung des Aufruf-Threads mindestens für eine bestimmte Zeit aussetzen
- void thrd_yield (void); // Erlaube die Ausführung anderer Threads anstelle des Threads, der ihn aufruft
- _Noreturn void thrd_exit (int res); // beendet den Thread den Thread, der ihn aufruft
- int thrd_detatch (thrd_t thr; // Entfernt einen bestimmten Thread aus der aktuellen Umgebung
- int thrd_join (thrd_t thr, int * res); // Blockiert den aktuellen Thread, bis der angegebene Thread beendet ist

Bemerkungen

Durch die Verwendung von Threads kann ein zusätzliches undefiniertes Verhalten eingeführt werden, z. B. eine http://www.riptutorial.com/c/example/2622/data-race . Für den Race-freien Zugriff auf Variablen, die von verschiedenen Threads gemeinsam genutzt werden, bietet C11 die Mutex-Funktion `mtx_lock()` oder die (optional) http://www.riptutorial.com/c/topic/4924/atomics-Datentypen und zugehörigen Funktionen in `stdatomic.h` .

Examples

C11 Threads einfaches Beispiel

```
#include <threads.h>
#include <stdio.h>

int run(void *arg)
{
    printf("Hello world of C11 threads.");

    return 0;
```

```
}

int main(int argc, const char *argv[])
{
    thrd_t thread;
    int result;

    thrd_create(&thread, run, NULL);

    thrd_join(&thread, &result);

    printf("Thread return %d at the end\n", result);
}
```

Multithreading online lesen: https://riptutorial.com/de/c/topic/10489/multithreading

Kapitel 38: Nebenwirkungen

Examples

Operatoren vor / nach Inkrementieren / Verringern

In C gibt es zwei unäre Operatoren - '++' und '-', die sehr häufig zu Verwirrung führen. Der Operator `++` wird als *Inkrementoperator* und der Operator `--` als *Dekrementoperator bezeichnet* . Beide können entweder als *Präfix* oder als *Postfix verwendet* werden. Die Syntax für das Präfix-Formular für den Operator `++` ist `++operand` und die Syntax für das Postfix-Formular ist `operand++` . Bei Verwendung in der Präfixform wird der Operand zuerst um `1` inkrementiert und der resultierende Wert des Operanden wird bei der Auswertung des Ausdrucks verwendet. Betrachten Sie das folgende Beispiel:

```
int n, x = 5;
n = ++x; /* x is incremented  by 1(x=6), and result is assigned to n(6) */
         /* this is a short form for two statements: */
         /* x = x + 1; */
         /* n = x ; */
```

Bei Verwendung im Postfix-Formular wird der aktuelle Wert des Operanden im Ausdruck verwendet, und der Wert des Operanden wird um `1` erhöht. Betrachten Sie das folgende Beispiel:

```
int n, x = 5;
n = x++; /* value of x(5) is assigned first to n(5), and then x is incremented by 1; x(6) */
         /* this is a short form for two statements: */
         /* n = x; */
         /* x = x + 1; */
```

Die Funktionsweise des Dekrementoperator `--` kann in ähnlicher Weise verstanden werden.

Der folgende Code veranschaulicht, was jeder tut

```
    int main()
    {
        int a, b, x = 42;
        a = ++x; /* a and x are 43 */
        b = x++; /* b is 43, x is 44 */
        a = x--; /* a is is 44, x is 43 */
        b = --x; /* b and x are 42 */

        return 0;
    }
```

Aus dem oben Gesagten ist klar , dass Postbetreiber den aktuellen Wert einer Variablen zurückgeben und es *dann* ändern, aber vor Operatoren die Variable ändern und *dann* den geänderten Wert zurück.

In allen Versionen von C ist die Reihenfolge der Auswertung von Vor- und Nachoperatoren nicht

definiert. Daher kann der folgende Code unerwartete Ausgaben zurückgeben:

```
int main()
{
    int a, x = 42;
    a = x++ + x; /* wrong */
    a = x + x; /* right */
    ++x;

    int ar[10];
    x = 0;
    ar[x] = x++; /* wrong */
    ar[x++] = x; /* wrong */
    ar[x] = x; /* right */
    ++x;
    return 0;
}
```

Beachten Sie, dass es auch empfehlenswert ist, Pre-Post-Operatoren zu verwenden, wenn sie alleine in einer Anweisung verwendet werden. Sehen Sie sich dazu den obigen Code an.

Beachten Sie außerdem, dass beim Aufruf einer Funktion alle Nebeneffekte auf Argumente auftreten müssen, bevor die Funktion ausgeführt wird.

```
int foo(int x)
{
    return x;
}

int main()
{
    int a = 42;
    int b = foo(a++);  /* This returns 43, even if it seems like it should return 42 */
    return 0;
}
```

Nebenwirkungen online lesen: https://riptutorial.com/de/c/topic/7094/nebenwirkungen

Kapitel 39: Operatoren

Einführung

Ein Operator in einer Programmiersprache ist ein Symbol, das den Compiler oder Interpreter anweist, eine bestimmte mathematische, relationale oder logische Operation auszuführen und ein Endergebnis zu erzeugen.

C hat viele mächtige Operatoren. Viele C-Operatoren sind binäre Operatoren, dh sie haben zwei Operanden. In `a / b` ist `/` beispielsweise ein binärer Operator, der zwei Operanden (`a` , `b`) akzeptiert. Es gibt einige unäre Operatoren, die einen Operanden verwenden (zum Beispiel: `~` , `++`) und nur einen ternären Operator `? :` .

Syntax

- expr1 Operator
- Operator Ausdruck2
- Ausdruck1 Operator Ausdruck2
- Ausdruck1? Ausdruck2: Ausdruck3

Bemerkungen

Die Betreiber haben ein *arity,* einen *Vorrang* und eine *Assoziativität.*

- *Arity* gibt die Anzahl der Operanden an. In C gibt es drei verschiedene Operatoren:
    - Unary (1 Operand)
    - Binär (2 Operanden)
    - Ternär (3 Operanden)
- *Vorrang* gibt an, welche Operatoren sich zuerst an ihre Operanden "binden". Das heißt, welcher Operator hat Priorität, um seine Operanden zu bearbeiten. Beispielsweise folgt die C-Sprache der Konvention, dass Multiplikation und Division Vorrang vor Addition und Subtraktion haben:

  ```
  a * b + c
  ```

  Gibt das gleiche Ergebnis wie

  ```
  (a * b) + c
  ```

  Wenn dies nicht erwünscht ist, kann der Vorrang mithilfe von Klammern erzwungen werden, da sie die *höchste* Priorität aller Operatoren haben.

```
a * (b + c)
```

Dieser neue Ausdruck führt zu einem Ergebnis, das sich von den vorherigen beiden Ausdrücken unterscheidet.

Die C-Sprache hat viele Prioritätsstufen. Nachfolgend wird eine Tabelle aller Operatoren in absteigender Rangfolge aufgeführt.

Vorrang-Tabelle

<table>
<tr><th>Operatoren</th><th>Assoziativität</th></tr>
<tr><td><code>() [] -> .</code></td><td>links nach rechts</td></tr>
<tr><td><code>! ~ ++ -- + - *</code> (dereference) <code>(type) sizeof</code></td><td>rechts nach links</td></tr>
<tr><td><code>*</code> (Multiplikation) <code>/ %</code></td><td>links nach rechts</td></tr>
<tr><td><code>+ -</code></td><td>links nach rechts</td></tr>
<tr><td><code><< >></code></td><td>links nach rechts</td></tr>
<tr><td><code>< <= > >=</code></td><td>links nach rechts</td></tr>
<tr><td><code>== !=</code></td><td>links nach rechts</td></tr>
<tr><td><code>&</code></td><td>links nach rechts</td></tr>
<tr><td><code>^</code></td><td>links nach rechts</td></tr>
<tr><td><code>|</code></td><td>links nach rechts</td></tr>
<tr><td><code>&&</code></td><td>links nach rechts</td></tr>
<tr><td><code>||</code></td><td>links nach rechts</td></tr>
<tr><td><code>?:</code></td><td>rechts nach links</td></tr>
<tr><td><code>= += -= *= /= %= &= ^= |= <<= >>=</code></td><td>rechts nach links</td></tr>
<tr><td><code>,</code></td><td>links nach rechts</td></tr>
</table>

- *Assoziativität* gibt an, wie Gleichheitsoperatoren standardmäßig gebunden werden, und es gibt zwei Arten: *Links nach rechts* und *Rechts nach links* . Ein Beispiel für die *Links-Rechts-*Bindung ist der Subtraktionsoperator (`-`). Der Ausdruck

```
a - b - c - d
```

hat drei Subtraktionen mit identischer Priorität, ergibt jedoch das gleiche Ergebnis wie

```
((a - b) - c) - d
```

weil der am weitesten links `-` zuerst an seine zwei Operanden bindet.

Ein Beispiel für die Assoziativität von *rechts nach links* sind die Dereferenzierungsoperatoren `*` und Post-Increment `++` . Beide haben gleiche Priorität, wenn sie also in einem Ausdruck wie verwendet werden

```
* ptr ++
```

ist das äquivalent zu

```
* (ptr ++)
```

weil der rechte, unäre Operator (`++`) zuerst an seinen einzelnen Operanden bindet.

Examples

Beziehungsoperatoren

Beziehungsoperatoren prüfen, ob eine bestimmte Beziehung zwischen zwei Operanden wahr ist. Das Ergebnis wird mit 1 (was *wahr ist*) oder 0 (was *falsch* bedeutet) ausgewertet. Dieses Ergebnis wird häufig verwendet, um den Steuerungsfluss zu beeinflussen (über `if` , `while` , `for`), kann aber auch in Variablen gespeichert werden.

Gleich "=="

Prüft, ob die angegebenen Operanden gleich sind.

```
1 == 0;         /* evaluates to 0. */
1 == 1;         /* evaluates to 1. */

int x = 5;
int y = 5;
int *xptr = &x, *yptr = &y;
xptr == yptr;   /* evaluates to 0, the operands hold different location addresses. */
*xptr == *yptr; /* evaluates to 1, the operands point at locations that hold the same value.
*/
```

Achtung: Dieser Operator sollte nicht mit dem Zuweisungsoperator (`=`) verwechselt werden!

Nicht gleich "! ="

Prüft, ob die angegebenen Operanden nicht gleich sind.

```
1 != 0;         /* evaluates to 1. */
1 != 1;         /* evaluates to 0. */
```

```
int x = 5;
int y = 5;
int *xptr = &x, *yptr = &y;
xptr != yptr;   /* evaluates to 1, the operands hold different location addresses. */
*xptr != *yptr; /* evaluates to 0, the operands point at locations that hold the same value.
*/
```

Dieser Operator gibt effektiv das Gegenteil von dem Gleichheitsoperator (`==`) zurück.

Nicht "!"

Prüfen Sie, ob ein Objekt gleich `0` .

Die `!` kann auch direkt mit einer Variablen verwendet werden:

```
!someVal
```

Das hat den gleichen Effekt wie:

```
someVal == 0
```

Größer als ">"

Überprüft, ob der linke Operand einen größeren Wert als der rechte Operand hat

```
5 > 4      /* evaluates to 1. */
4 > 5      /* evaluates to 0. */
4 > 4      /* evaluates to 0. */
```

Weniger als "<"

Überprüft, ob der linke Operand einen kleineren Wert als der rechte Operand hat

```
5 < 4      /* evaluates to 0. */
4 < 5      /* evaluates to 1. */
4 < 4      /* evaluates to 0. */
```

Größer oder gleich "> ="

Überprüft, ob der linke Operand dem rechten Operanden einen größeren oder gleichen Wert hat.

```
5 >= 4      /* evaluates to 1. */
4 >= 5      /* evaluates to 0. */
4 >= 4      /* evaluates to 1. */
```

Weniger oder gleich "<="

Überprüft, ob der linke Operand einen kleineren oder gleichen Wert für den rechten Operanden hat.

```
5 <= 4      /* evaluates to 0. */
4 <= 5      /* evaluates to 1. */
4 <= 4      /* evaluates to 1. */
```

Zuweisungsoperatoren

Weist den Wert des rechten Operanden dem vom linken Operanden genannten Speicherort zu und gibt den Wert zurück.

```
int x = 5;      /* Variable x holds the value 5. Returns 5. */
char y = 'c';   /* Variable y holds the value 99. Returns 99
                 * (as the character 'c' is represented in the ASCII table with 99).
                 */
float z = 1.5;  /* variable z holds the value 1.5. Returns 1.5. */
char const* s = "foo"; /* Variable s holds the address of the first character of the string
'foo'. */
```

Mehrere arithmetische Operationen haben einen *zusammengesetzten Zuweisungsoperator* .

```
a += b  /* equal to: a = a + b */
a -= b  /* equal to: a = a - b */
a *= b  /* equal to: a = a * b */
a /= b  /* equal to: a = a / b */
a %= b  /* equal to: a = a % b */
a &= b  /* equal to: a = a & b */
a |= b  /* equal to: a = a | b */
a ^= b  /* equal to: a = a ^ b */
a <<= b /* equal to: a = a << b */
a >>= b /* equal to: a = a >> b */
```

Ein wichtiges Merkmal dieser zusammengesetzten Zuordnungen ist, dass der Ausdruck auf der linken Seite (`a`) nur einmal ausgewertet wird. ZB wenn `p` ein Zeiger ist

```
*p += 27;
```

Dereferences `p` nur einmal, während das Folgende zweimal tut.

```
*p = *p + 27;
```

Es sei auch darauf hingewiesen, dass das Ergebnis einer Zuweisung wie `a = b` sogenannte *r-Wert ist* . Die Zuordnung hat also tatsächlich einen Wert, der dann einer anderen Variablen zugewiesen werden kann. Dies ermöglicht die Verkettung von Zuweisungen, um mehrere Variablen in einer einzigen Anweisung festzulegen.

Dieser *Wert* kann in den steuernden Ausdrücken von `if` Anweisungen (oder Schleifen- oder `switch` Anweisungen) verwendet werden, die Code für das Ergebnis eines anderen Ausdrucks oder Funktionsaufrufs schützen. Zum Beispiel:

```
char *buffer;
if ((buffer = malloc(1024)) != NULL)
{
    /* do something with buffer */
    free(buffer);
}
else
{
    /* report allocation failure */
}
```

Aus diesem Grund muss ein häufiger Tippfehler vermieden werden, der zu mysteriösen Fehlern führen kann.

```
int a = 2;
/* ... */
if (a = 1)
    /* Delete all files on my hard drive */
```

Das wird katastrophale Folgen haben, da `a = 1` wird immer bewerten `1` und damit der kontrollierende Ausdruck der , `if` Aussage immer wahr sein wird (mehr über diese gemeinsame pitfall lesen hier). Der Autor wollte den Gleichheitsoperator (`==`) wie folgt verwenden:

```
int a = 2;
/* ... */
if (a == 1)
    /* Delete all files on my hard drive */
```

Operator Assoziativität

```
int a, b = 1, c = 2;
a = b = c;
```

Dies ordnet `c` zu `b` , der kehrt `b` , die als zugewiesen ist `a` . Dies geschieht, weil alle Zuweisungsoperatoren rechte Assoziativität besitzen. Das bedeutet, dass die Operation ganz rechts im Ausdruck zuerst ausgewertet wird und von rechts nach links verläuft.

Rechenzeichen

Grundrechenarten

Gibt einen Wert zurück, der das Ergebnis der Anwendung des linken Operanden auf den rechten Operanden ist, und zwar mit der zugehörigen mathematischen Operation. Es gelten die üblichen mathematischen Regeln der Kommutierung (dh Addition und Multiplikation sind kommutativ, Subtraktion, Division und Modulus nicht).

Zusatzoperator

Mit dem Additionsoperator (`+`) werden zwei Operanden zusammengefügt. Beispiel:

```
#include <stdio.h>

int main(void)
{
    int a = 5;
    int b = 7;

    int c = a + b; /* c now holds the value 12 */

    printf("%d + %d = %d",a,b,c); /* will output "5 + 7 = 12" */

    return 0;
}
```

Subtraktionsoperator

Der Subtraktionsoperator (`-`) dient zum Subtrahieren des zweiten Operanden vom ersten. Beispiel:

```
#include <stdio.h>

int main(void)
{
    int a = 10;
    int b = 7;

    int c = a - b; /* c now holds the value 3 */

    printf("%d - %d = %d",a,b,c); /* will output "10 - 7 = 3" */

    return 0;
}
```

Multiplikationsoperator

Mit dem Multiplikationsoperator (`*`) werden beide Operanden multipliziert. Beispiel:

```
#include <stdio.h>

int main(void)
{
    int a = 5;
    int b = 7;

    int c = a * b; /* c now holds the value 35 */

    printf("%d * %d = %d",a,b,c); /* will output "5 * 7 = 35" */

    return 0;
}
```

Nicht mit dem Dereferenzierungsoperator `*` verwechseln.

Bereichsoperator

Der Divisionsoperator (`/`) teilt den ersten Operanden durch den zweiten. Wenn beide Operanden der Division Ganzzahlen sind, wird ein ganzzahliger Wert zurückgegeben und der Rest verworfen (verwenden Sie den Modulo-Operator `%` zum Berechnen und Abrufen des Restes).

Wenn einer der Operanden ein Gleitkommawert ist, ist das Ergebnis eine Annäherung an den Bruch.

Beispiel:

```
#include <stdio.h>

int main (void)
{
    int a = 19 / 2 ; /* a holds value 9   */
    int b = 18 / 2 ; /* b holds value 9   */
    int c = 255 / 2; /* c holds value 127 */
    int d = 44 / 4 ; /* d holds value 11  */
    double e = 19 / 2.0 ; /* e holds value 9.5   */
    double f = 18.0 / 2 ; /* f holds value 9.0   */
    double g = 255 / 2.0; /* g holds value 127.5 */
    double h = 45.0 / 4 ; /* h holds value 11.25 */

    printf("19 / 2 = %d\n", a);    /* Will output "19 / 2 = 9"    */
    printf("18 / 2 = %d\n", b);    /* Will output "18 / 2 = 9"    */
    printf("255 / 2 = %d\n", c);   /* Will output "255 / 2 = 127" */
    printf("44 / 4 = %d\n", d);    /* Will output "44 / 4 = 11"   */
    printf("19 / 2.0 = %g\n", e);  /* Will output "19 / 2.0 = 9.5"    */
    printf("18.0 / 2 = %g\n", f);  /* Will output "18.0 / 2 = 9"      */
    printf("255 / 2.0 = %g\n", g); /* Will output "255 / 2.0 = 127.5" */
    printf("45.0 / 4 = %g\n", h);  /* Will output "45.0 / 4 = 11.25"  */

    return 0;
}
```

Modulo Operator

Der Modulo-Operator (`%`) empfängt nur ganzzahlige Operanden und wird verwendet, um den Rest zu berechnen, nachdem der erste Operand durch den zweiten geteilt wurde. Beispiel:

```
#include <stdio.h>

int main (void) {
    int a = 25 % 2;    /* a holds value 1  */
    int b = 24 % 2;    /* b holds value 0  */
    int c = 155 % 5;   /* c holds value 0  */
    int d = 49 % 25;   /* d holds value 24 */

    printf("25 % 2 = %d\n", a);     /* Will output "25 % 2 = 1"    */
    printf("24 % 2 = %d\n", b);     /* Will output "24 % 2 = 0"    */
    printf("155 % 5 = %d\n", c);    /* Will output "155 % 5 = 0"   */
    printf("49 % 25 = %d\n", d);    /* Will output "49 % 25 = 24"  */
```

```
    return 0;
}
```

Inkrement- / Dekrementoperatoren

Die Operatoren inkrementieren (`a++`) und dekrementieren (`a--`) unterscheiden sich dadurch, dass sie den Wert der Variablen ändern, auf die Sie sie anwenden, ohne einen Zuweisungsoperator. Sie können Inkrementierungs- und Dekrementierungsoperatoren entweder vor oder nach der Variablen verwenden. Durch die Platzierung des Operators wird der Zeitpunkt der Inkrementierung / Dekrementierung des Werts vor oder nach der Zuweisung der Variablen geändert. Beispiel:

```
#include <stdio.h>

int main(void)
{
    int a = 1;
    int b = 4;
    int c = 1;
    int d = 4;

    a++;
    printf("a = %d\n",a);    /* Will output "a = 2" */
    b--;
    printf("b = %d\n",b);    /* Will output "b = 3" */

    if (++c > 1) { /* c is incremented by 1 before being compared in the condition */
        printf("This will print\n");    /* This is printed */
    } else {
        printf("This will never print\n");    /* This is not printed */
    }

    if (d-- < 4) {  /* d is decremented after being compared */
        printf("This will never print\n");    /* This is not printed */
    } else {
        printf("This will print\n");    /* This is printed */
    }
}
```

Wie das Beispiel für `c` und `d` zeigt, haben beide Operatoren zwei Formen: Präfixnotation und Postfixnotation. Beide haben den gleichen Effekt beim Inkrementieren (`++`) oder Dekrementieren (`--`) der Variablen, unterscheiden sich jedoch durch den von ihnen zurückgegebenen Wert: Präfixoperationen führen die Operation zuerst aus und geben dann den Wert zurück, während Postfixoperationen zuerst den Wert bestimmen, der zu verwenden ist zurückgegeben werden und dann die Operation ausführen.

Aufgrund dieses möglicherweise kontraintuitiven Verhaltens ist die Verwendung von Inkrementierungs- / Dekrementierungsoperatoren in Ausdrücken umstritten.

Logische Operatoren

Logisches UND

Führt eine logische boolesche UND-Verknüpfung der beiden Operanden durch, wobei 1 zurückgegeben wird, wenn beide Operanden nicht Null sind. Der logische AND-Operator ist vom Typ `int` .

```
0 && 0  /* Returns 0. */
0 && 1  /* Returns 0. */
2 && 0  /* Returns 0. */
2 && 3  /* Returns 1. */
```

Logisches ODER

Führt eine logische boolesche ODER-Verknüpfung der beiden Operanden durch, wobei 1 zurückgegeben wird, wenn einer der Operanden nicht Null ist. Der logische Operator OR ist vom Typ `int` .

```
0 || 0  /* Returns 0. */
0 || 1  /* Returns 1.  */
2 || 0  /* Returns 1.  */
2 || 3  /* Returns 1.  */
```

Logisch NICHT

Führt eine logische Negation durch. Der logische NOT-Operator ist vom Typ `int` . Der Operator NOT prüft, ob mindestens ein Bit gleich 1 ist. Wenn dies der Fall ist, wird 0 zurückgegeben. Andernfalls wird 1 zurückgegeben.

```
!1 /* Returns 0. */
!5 /* Returns 0. */
!0 /* Returns 1. */
```

Kurzschlussbewertung

Sowohl `&&` als auch `||` haben einige wichtige Eigenschaften gemeinsam :

- der linke Operand (LHS) wird vollständig ausgewertet, bevor der rechte Operand (RHS) überhaupt ausgewertet wird,
- es gibt einen Sequenzpunkt zwischen der Auswertung des linken Operanden und des rechten Operanden,
- und am wichtigsten ist, dass der rechte Operand überhaupt nicht ausgewertet wird, wenn das Ergebnis des linken Operanden das Gesamtergebnis bestimmt.

Das bedeutet, dass:

- Wenn die LHS als "wahr" (nicht Null) bewertet wird, ist die RHS von `||` wird nicht ausgewertet (weil das Ergebnis von "wahr ODER alles" ist "wahr"),
- Wenn der LHS-Wert als "falsch" (Null) ausgewertet wird, wird der RHS-Wert von `&&` nicht ausgewertet (da das Ergebnis von "falsch AND irgendetwas" ist "falsch").

Dies ist wichtig, da Sie damit Code schreiben können:

```
const char *name_for_value(int value)
{
    static const char *names[] = { "zero", "one", "two", "three", };
    enum { NUM_NAMES = sizeof(names) / sizeof(names[0]) };
    return (value >= 0 && value < NUM_NAMES) ? names[value] : "infinity";
}
```

Wenn ein negativer Wert an die Funktion übergeben wird, wird der `value >= 0` als falsch ausgewertet, und der `value < NUM_NAMES` wird nicht ausgewertet.

Inkrement / Dekrement

Die Inkrementierungs- und Dekrementierungsoperatoren sind in *Präfix-* und *Postfixform vorhanden* .

```
int a = 1;
int b = 1;
int tmp = 0;

tmp = ++a;        /* increments a by one, and returns new value; a == 2, tmp == 2  */
tmp = a++;        /* increments a by one, but returns old value; a == 3, tmp == 2 */
tmp = --b;        /* decrements b by one, and returns new value; b == 0, tmp == 0 */
tmp = b--;        /* decrements b by one, but returns old value; b == -1, tmp == 0 */
```

Beachten Sie, dass arithmetische Operationen keine Sequenzpunkte einführen, sodass bestimmte Ausdrücke mit `++` oder `--` Operatoren undefiniertes Verhalten verursachen können .

Bedingter Operator / Ternärer Operator

Wertet den ersten Operanden aus und wertet den zweiten Operanden aus, wenn der resultierende Wert ungleich Null ist. Andernfalls wertet es den dritten Operanden aus, wie im folgenden Beispiel gezeigt:

```
a = b ? c : d;
```

ist äquivalent zu:

```
if (b)
    a = c;
else
    a = d;
```

Dieser Pseudo-Code repräsentiert es: `condition ? value_if_true : value_if_false` . Jeder Wert kann das Ergebnis eines bewerteten Ausdrucks sein.

```
int x = 5;
int y = 42;
printf("%i, %i\n", 1 ? x : y, 0 ? x : y); /* Outputs "5, 42" */
```

Der bedingte Operator kann geschachtelt werden. Der folgende Code bestimmt beispielsweise die größere von drei Zahlen:

```
big= a > b ? (a > c ? a : c)
           : (b > c ? b : c);
```

Im folgenden Beispiel werden sogar Ganzzahlen in eine Datei und ungerade Ganzzahlen in eine andere Datei geschrieben:

```
#include<stdio.h>

int main()
{
    FILE *even, *odds;
    int n = 10;
    size_t k = 0;

    even = fopen("even.txt", "w");
    odds = fopen("odds.txt", "w");

    for(k = 1; k < n + 1; k++)
    {
        k%2==0 ? fprintf(even, "\t%5d\n", k)
               : fprintf(odds, "\t%5d\n", k);
    }
    fclose(even);
    fclose(odds);

    return 0;
}
```

Der bedingte Operator verbindet sich von rechts nach links. Folgendes berücksichtigen:

```
exp1 ? exp2 : exp3 ? exp4 : exp5
```

Da die Zuordnung von rechts nach links erfolgt, wird der obige Ausdruck als ausgewertet

```
exp1 ? exp2 : ( exp3 ? exp4 : exp5 )
```

Komma-Operator

Wertet seinen linken Operanden aus, verwirft den resultierenden Wert und wertet dann seinen Rechte-Operanden aus, und result ergibt den Wert seines ganz rechten Operanden.

```
int x = 42, y = 42;
printf("%i\n", (x *= 2, y)); /* Outputs "42". */
```

Der Kommaoperator führt einen Sequenzpunkt zwischen seinen Operanden ein.

Beachten Sie, dass das *Komma,* das in Funktionsaufrufen verwendet wird, die separaten Argumente aufruft, NICHT der *Kommaoperator ist* , sondern als *Trennzeichen bezeichnet wird,* das sich vom *Kommaoperator unterscheidet* . Daher verfügt es nicht über die Eigenschaften des

Kommaoperators .

Der obige Aufruf von `printf()` enthält sowohl den *Kommaoperator* als auch das *Trennzeichen* .

```
printf("%i\n", (x *= 2, y)); /* Outputs "42". */
/*            ^        ^ this is a comma operator */
/*            this is a separator */
```

Der Kommaoperator wird häufig sowohl im Initialisierungsabschnitt als auch im Aktualisierungsabschnitt einer `for` Schleife verwendet. Zum Beispiel:

```
for(k = 1; k < 10; printf("\%d\\n", k), k += 2);   /*outputs the odd numbers below 9/*

/* outputs sum to first 9 natural numbers */
for(sumk = 1, k = 1; k < 10; k++, sumk += k)
    printf("\%5d\%5d\\n", k, sumk);
```

Cast Operator

Führt eine *explizite* Konvertierung in den angegebenen Typ aus dem Wert durch, der sich aus der Auswertung des angegebenen Ausdrucks ergibt.

```
int x = 3;
int y = 4;
printf("%f\n", (double)x / y); /* Outputs "0.750000". */
```

Hier ist der Wert von `x` in einem umgewandelt wird `double` fördert die Aufteilung um den Wert von `y` zu `double` , auch, und das Ergebnis der Division, ein `double` geleitet wird , um `printf` zum Drucken.

sizeof Operator

Mit einem Typ als Operand

Wertet die Größe der Objekte des angegebenen Typs in Bytes des Typs `size_t` aus. Benötigt Klammern um den Typ.

```
printf("%zu\n", sizeof(int)); /* Valid, outputs the size of an int object, which is platform-
dependent. */
printf("%zu\n", sizeof int); /* Invalid, types as arguments need to be surrounded by
parentheses! */
```

Mit einem Ausdruck als Operand

Wertet die Größe von Objekten des Typs des angegebenen Ausdrucks in Bytes des Typs `size_t` aus. Der Ausdruck selbst wird nicht ausgewertet. Klammern sind nicht erforderlich. Da der angegebene Ausdruck jedoch unärlich sein muss, sollten Sie ihn immer verwenden.

```
char ch = 'a';
```

```
printf("%zu\n", sizeof(ch)); /* Valid, will output the size of a char object, which is always
1 for all platforms. */
printf("%zu\n", sizeof ch);  /* Valid, will output the size of a char object, which is always
1 for all platforms. */
```

Zeigerarithmetik

Zeigerzusatz

Wenn ein Zeiger und ein skalarer Typ `N` , wird ein Zeiger auf das `N` te Element des Zeigetyps ausgewertet, der direkt auf das Zeigefachobjekt im Speicher folgt.

```
int arr[] = {1, 2, 3, 4, 5};
printf("*(arr + 3) = %i\n", *(arr + 3)); /* Outputs "4", arr's fourth element. */
```

Es spielt keine Rolle, ob der Zeiger als Operandenwert oder Skalarwert verwendet wird. Dies bedeutet, dass Dinge wie `3 + arr` gültig sind. Wenn `arr[k]` das `k+1` Glied eines Arrays ist, dann ist `arr+k` ein Zeiger auf `arr[k]` . Mit anderen Worten, `arr` oder `arr+0` ist ein Zeiger auf `arr[0]` , `arr+1` ist ein Zeiger auf `arr[2]` und so weiter. Im Allgemeinen ist `*(arr+k)` dasselbe wie `arr[k]` .

Im Gegensatz zur üblichen Arithmetik werden durch Hinzufügen von `1` zu einem Zeiger auf ein `int` `4` Werte zum aktuellen Adresswert hinzugefügt. Da Arraynamen konstante Zeiger sind, ist `+` der einzige Operator, mit dem auf die Mitglieder eines Arrays über die Zeigernotation über den Arraynamen zugegriffen werden kann. Wenn Sie jedoch einen Zeiger auf ein Array definieren, können Sie die Daten in einem Array flexibler verarbeiten. Beispielsweise können wir die Mitglieder eines Arrays wie folgt drucken:

```
#include<stdio.h>
static const size_t N = 5

int main()
{
    size_t k = 0;
    int arr[] = {1, 2, 3, 4, 5};
    for(k = 0; k < N; k++)
    {
        printf("\n\t%d", *(arr + k));
    }
    return 0;
}
```

Durch die Definition eines Zeigers auf das Array entspricht das obige Programm dem folgenden:

```
#include<stdio.h>
static const size_t N = 5

int main()
{
    size_t k = 0;
    int arr[] = {1, 2, 3, 4, 5};
    int *ptr = arr; /* or int *ptr = &arr[0]; */
    for(k = 0; k < N; k++)
```

```
    {
        printf("\n\t%d", ptr[k]);
        /* or   printf("\n\t%d", *(ptr + k)); */
        /* or   printf("\n\t%d", *ptr++); */
    }
    return 0;
}
```

Stellen Sie `arr` dass auf die Mitglieder des Arrays `arr` mit den Operatoren `+` und `++` zugegriffen wird. Die anderen Operatoren, die mit dem Zeiger `ptr` werden können, sind `-` und `--` .

Zeigerabzug

Wenn zwei Zeiger auf denselben Typ gegeben werden, wird ein Objekt vom Typ `ptrdiff_t` , das den Skalarwert enthält, der zum zweiten Zeiger hinzugefügt werden muss, um den Wert des ersten Zeigers zu erhalten.

```
int arr[] = {1, 2, 3, 4, 5};
int *p = &arr[2];
int *q = &arr[3];
ptrdiff_t diff = q - p;

printf("q - p = %ti\n", diff); /* Outputs "1". */
printf("*(p + (q - p)) = %d\n", *(p + diff)); /* Outputs "4". */
```

Zugriffsoperatoren

Die Mitglied Betreiber der Zugangskontrolle (dot `.` Pfeil und `->`) verwendet , um ein Mitglied eines für den Zugriff auf `struct` .

Mitglied des Objekts

Wertet den Wert lvalue aus, der das Objekt angibt, das Mitglied des aufgerufenen Objekts ist.

```
struct MyStruct
{
    int x;
    int y;
};

struct MyStruct myObject;
myObject.x = 42;
myObject.y = 123;

printf(".x = %i, .y = %i\n", myObject.x, myObject.y); /* Outputs ".x = 42, .y = 123". */
```

Mitglied des Objektes, auf das verwiesen wird

Syntaktischer Zucker zur Dereferenzierung, gefolgt vom Zugang der Mitglieder. Tatsächlich ist ein Ausdruck der Form `x->y` eine Abkürzung für `(*x).y` - der Pfeiloperator ist jedoch viel klarer,

insbesondere wenn die Strukturzeiger verschachtelt sind.

```
struct MyStruct
{
    int x;
    int y;
};

struct MyStruct myObject;
struct MyStruct *p = &myObject;

p->x = 42;
p->y = 123;

printf(".x = %i, .y = %i\n", p->x, p->y); /* Outputs ".x = 42, .y = 123". */
printf(".x = %i, .y = %i\n", myObject.x, myObject.y); /* Also outputs ".x = 42, .y = 123". */
```

Adresse von

Die unären `&` Operator ist die Adresse des Betreibers. Es wertet den angegebenen Ausdruck aus, wobei das resultierende Objekt ein Wert sein muss. Dann wird es zu einem Objekt ausgewertet, dessen Typ ein Zeiger auf den Typ des resultierenden Objekts ist, und enthält die Adresse des resultierenden Objekts.

```
int x = 3;
int *p = &x;
printf("%p = %p\n", (void *)&x, (void *)p); /* Outputs "A = A", for some implementation-
defined A. */
```

Dereferenz

Der unäre `*` Operator dereferenziert einen Zeiger. Sie wertet den Wert lvalue aus, der sich aus der Dereferenzierung des Zeigers ergibt, der sich aus der Auswertung des angegebenen Ausdrucks ergibt.

```
int x = 42;
int *p = &x;
printf("x = %d, *p = %d\n", x, *p); /* Outputs "x = 42, *p = 42". */

*p = 123;
printf("x = %d, *p = %d\n", x, *p); /* Outputs "x = 123, *p = 123". */
```

Indizierung

Indizierung ist syntaktischer Zucker für die Zeigeraddition gefolgt von Dereferenzierung. Ein Ausdruck der Form `a[i]` ist effektiv äquivalent zu `*(a + i)` - aber die explizite tiefgestellte Schreibweise wird bevorzugt.

```
int arr[] = { 1, 2, 3, 4, 5 };
printf("arr[2] = %i\n", arr[2]); /* Outputs "arr[2] = 3". */
```

Austauschbarkeit der Indexierung

Das Hinzufügen eines Zeigers zu einer Ganzzahl ist eine kommutative Operation (dh, die Reihenfolge der Operanden ändert das Ergebnis nicht), also `pointer + integer == integer + pointer` .

Eine Folge davon ist, dass `arr[3]` und `3[arr]` gleichwertig sind.

```
printf("3[arr] = %i\n", 3[arr]); /* Outputs "3[arr] = 4". */
```

Die Verwendung eines Ausdrucks `3[arr]` anstelle von `arr[3]` wird im Allgemeinen nicht empfohlen, da dies die Lesbarkeit des Codes beeinflusst. Es ist in populären Programmwettbewerben beliebt.

Operator für Funktionsaufruf

Der erste Operand muss ein Funktionszeiger sein (ein Funktionsbezeichner ist auch akzeptabel, weil er in einen Zeiger auf die Funktion umgewandelt wird), der die aufzurufende Funktion identifiziert, und alle anderen Operanden, sofern vorhanden, werden als Argumente des Funktionsaufrufs bezeichnet . Wertet den Rückgabewert aus, der sich aus dem Aufruf der entsprechenden Funktion mit den jeweiligen Argumenten ergibt.

```
int myFunction(int x, int y)
{
    return x * 2 + y;
}

int (*fn)(int, int) = &myFunction;
int x = 42;
int y = 123;

printf("(*fn)(%i, %i) = %i\n", x, y, (*fn)(x, y)); /* Outputs "fn(42, 123) = 207". */
printf("fn(%i, %i) = %i\n", x, y, fn(x, y)); /* Another form: you don't need to dereference
explicitly */
```

Bitweise Operatoren

Bitweise Operatoren können verwendet werden, um Variablen auf Bitebene zu bearbeiten. Nachfolgend finden Sie eine Liste aller sechs in C unterstützten bitweisen Operatoren:

Symbol	Operator
&	bitweise UND
|	bitweises ODER
^	bitweises exklusives ODER (XOR)
~	bitweise nicht (eine Ergänzung)
<<	logische Linksverschiebung

Symbol	Operator
>>	logische Rechtsverschiebung

Das folgende Programm veranschaulicht die Verwendung aller bitweisen Operatoren:

```
#include <stdio.h>

int main(void)
{
   unsigned int a = 29;    /* 29 = 0001 1101 */
   unsigned int b = 48;    /* 48 = 0011 0000 */
   int c = 0;

   c = a & b;              /* 32 = 0001 0000 */
   printf("%d & %d = %d\n", a, b, c );

   c = a | b;              /* 61 = 0011 1101 */
   printf("%d | %d = %d\n", a, b, c );

   c = a ^ b;              /* 45 = 0010 1101 */
   printf("%d ^ %d = %d\n", a, b, c );

   c = ~a;                 /* -30 = 1110 0010 */
   printf("~%d = %d\n", a, c );

   c = a << 2;             /* 116 = 0111 0100 */
   printf("%d << 2 = %d\n", a, c );

   c = a >> 2;             /* 7 = 0000 0111 */
   printf("%d >> 2 = %d\n", a, c );

   return 0;
}
```

Bitweise Operationen mit vorzeichenbehafteten Typen sollten vermieden werden, da das Vorzeichenbit einer solchen Bitdarstellung eine besondere Bedeutung hat. Für die Schichtbetreiber gelten besondere Einschränkungen:

- Das Verschieben von 1 Bit in das vorzeichenbehaftete Bit nach links ist fehlerhaft und führt zu undefiniertem Verhalten.
- Wenn Sie einen negativen Wert (mit Vorzeichenbit 1) nach rechts verschieben, ist die Implementierung definiert und daher nicht portierbar.
- Wenn der Wert des rechten Operanden eines Verschiebungsoperators negativ ist oder größer oder gleich der Breite des beförderten linken Operanden ist, ist das Verhalten undefiniert.

Maskierung:

Maskieren bezieht sich auf den Vorgang des Extrahierens der gewünschten Bits aus einer Variablen (oder Umwandeln der gewünschten Bits in eine Variable) unter Verwendung von logischen bitweisen Operationen. Der Operand (eine Konstante oder Variable), der zur Maskierung verwendet wird, wird als *Maske bezeichnet* .

Maskierung wird auf verschiedene Arten verwendet:

- Das Bitmuster einer Ganzzahlvariablen bestimmen.
- Um einen Teil eines bestimmten Bitmusters in eine neue Variable zu kopieren, während der Rest der neuen Variablen mit 0 gefüllt wird (unter Verwendung von bitweisem AND)
- Einen Teil eines bestimmten Bitmusters in eine neue Variable kopieren, während der Rest der neuen Variablen mit 1s gefüllt wird (unter Verwendung von bitweisem ODER).
- Um einen Teil eines bestimmten Bitmusters in eine neue Variable zu kopieren, während der Rest des ursprünglichen Bitmusters innerhalb der neuen Variablen invertiert wird (unter Verwendung des bitweisen Exklusiv-ODER).

Die folgende Funktion verwendet eine Maske, um das Bitmuster einer Variablen anzuzeigen:

```
#include <limits.h>
void bit_pattern(int u)
{
    int i, x, word;
    unsigned mask = 1;
    word = CHAR_BIT * sizeof(int);
    mask = mask << (word - 1);    /* shift 1 to the leftmost position */
    for(i = 1; i <= word; i++)
    {
        x = (u & mask) ? 1 : 0;  /* identify the bit */
        printf("%d", x);         /* print bit value */
        mask >>= 1;              /* shift mask to the right by 1 bit */
    }
}
```

_Alignof

C11

Fragt die Ausrichtungsanforderung für den angegebenen Typ ab. Die Ausrichtungsanforderung ist eine positive integrale Potenz von 2, die die Anzahl der Bytes darstellt, zwischen denen zwei Objekte des Typs zugeordnet werden können. In C wird die Ausrichtungsanforderung in `size_t` gemessen.

Der Typname darf weder ein unvollständiger Typ noch ein Funktionstyp sein. Wenn ein Array als Typ verwendet wird, wird der Typ des Array-Elements verwendet.

Auf diesen Operator wird häufig über das Komfort-Makro `alignof` von `<stdalign.h>` .

```
int main(void)
{
    printf("Alignment of char = %zu\n", alignof(char));
    printf("Alignment of max_align_t = %zu\n", alignof(max_align_t));
    printf("alignof(float[10]) = %zu\n", alignof(float[10]));
    printf("alignof(struct{char c; int n;}) = %zu\n",
            alignof(struct {char c; int n;}));
}
```

Mögliche Ausgabe:

```
Alignment of char = 1
Alignment of max_align_t = 16
alignof(float[10]) = 4
alignof(struct{char c; int n;}) = 4
```

http://de.cppreference.com/w/c/language/_Alignof

Kurzschlussverhalten logischer Operatoren

Kurzschließen ist eine Funktion, die das Auswerten von Teilen einer (wenn / während / ...) Bedingung auslässt, wenn dies möglich ist. Bei einer logischen Operation an zwei Operanden wird der erste Operand ausgewertet (wahr oder falsch), und wenn ein Ergebnis vorliegt (dh der erste Operand ist bei Verwendung von && falsch, der erste Operand ist bei Verwendung von || der zweite Operand.) nicht bewertet.

Beispiel:

```
#include <stdio.h>

int main(void) {
  int a = 20;
  int b = -5;

  /* here 'b == -5' is not evaluated,
     since a 'a != 20' is false. */
  if (a != 20 && b == -5) {
    printf("I won't be printed!\n");
  }

  return 0;
}
```

Überzeugen Sie sich selbst:

```
#include <stdio.h>

int print(int i) {
  printf("print function %d\n", i);
  return i;
}

int main(void) {
  int a = 20;

  /* here 'print(a)' is not called,
     since a 'a != 20' is false. */
  if (a != 20 && print(a)) {
    printf("I won't be printed!\n");
  }

  /* here 'print(a)' is called,
     since a 'a == 20' is true. */
  if (a == 20 && print(a)) {
    printf("I will be printed!\n");
  }
```

```
  return 0;
}
```

Ausgabe:

```
$ ./a.out
print function 20
I will be printed!
```

Kurzschlüsse sind wichtig, wenn Sie vermeiden möchten, Begriffe zu berechnen, die (rechenmäßig) teuer sind. Darüber hinaus kann es den Ablauf Ihres Programms wie in diesem Fall stark beeinträchtigen: Warum wird dieses Programm "verzerrt!" 4 Mal?

Operatoren online lesen: https://riptutorial.com/de/c/topic/256/operatoren

Kapitel 40: Preprozessor und Makros

Einführung

Alle Präprozessorbefehle beginnen mit dem Hash-Symbol `#` . AC-Makro ist nur ein Präprozessorbefehl, der mit der `#define` Präprozessor-Direktive definiert wird. Während der Vorverarbeitungsphase ersetzt der C-Präprozessor (ein Teil des C-Compilers) einfach den Hauptteil des Makros, wo immer sein Name erscheint.

Bemerkungen

Wenn ein Compiler auf ein Makro im Code stößt, führt er einen einfachen String-Austausch durch. Es werden keine zusätzlichen Operationen ausgeführt. Aus diesem Grund berücksichtigen Änderungen durch den Präprozessor nicht den Geltungsbereich von C-Programmen. Beispielsweise ist eine Makrodefinition nicht darauf beschränkt, innerhalb eines Blocks zu liegen, und wird daher nicht durch ein `'}'` , das eine Blockanweisung beendet.

Der Präprozessor ist vom Entwurf her nicht vollständig - es gibt verschiedene Arten von Berechnungen, die nicht vom Präprozessor allein ausgeführt werden können.

Normalerweise verfügen Compiler über ein Befehlszeilenflag (oder eine Konfigurationseinstellung), mit dem wir die Kompilierung nach der Vorverarbeitungsphase stoppen und das Ergebnis überprüfen können. Auf POSIX-Plattformen ist dieses Flag `-E` . Wenn Sie gcc mit diesem Flag ausführen, wird die erweiterte Quelle in stdout ausgegeben:

```
$ gcc -E cprog.c
```

Oft ist der Präprozessor als separates Programm implementiert, das vom Compiler aufgerufen wird. Der gebräuchliche Name für dieses Programm lautet `cpp` . Eine Reihe von Präprozessoren gibt unterstützende Informationen aus, z. B. Informationen zu Zeilennummern, die von den nachfolgenden Kompilierungsphasen verwendet werden, um Debugging-Informationen zu generieren. Wenn der Präprozessor auf gcc basiert, unterdrückt die Option -P diese Informationen.

```
$ cpp -P cprog.c
```

Examples

Bedingter Einschluss und Änderung der bedingten Funktionssignatur

Bedingt einen Codeblock enthalten, weist der Vorprozessor mehrere Richtlinien (zB `#if` , `#ifdef` , `#else` , `#endif` , etc).

```
/* Defines a conditional `printf` macro, which only prints if `DEBUG`
```

```
 * has been defined
 */
#ifdef DEBUG
#define DLOG(x) (printf(x))
#else
#define DLOG(x)
#endif
```

Für die `#if` Bedingung können normale C-Vergleichsoperatoren verwendet werden

```
#if __STDC_VERSION__ >= 201112L
/* Do stuff for C11 or higher */
#elif __STDC_VERSION__ >= 199901L
/* Do stuff for C99 */
#else
/* Do stuff for pre C99 */
#endif
```

Die `#if` Direktive verhält sich ähnlich wie die C `if` `#if` Sie enthält nur integrale konstante Ausdrücke und keine Casts. Es unterstützt einen zusätzlichen unären Operator (`defined( identifier )` , der `1` zurückgibt, wenn der Bezeichner definiert ist, andernfalls `0` .

```
#if defined(DEBUG) && !defined(QUIET)
#define DLOG(x) (printf(x))
#else
#define DLOG(x)
#endif
```

Bedingte Funktionssignaturänderung

In den meisten Fällen wird erwartet, dass ein Release-Build einer Anwendung so wenig Overhead wie möglich hat. Beim Testen eines vorläufigen Builds können jedoch zusätzliche Protokolle und Informationen zu gefundenen Problemen hilfreich sein.

`SHORT SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd)` es gibt eine Funktion `SHORT SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd)` die bei einem Test-Build gewünscht wird, erzeugt ein Protokoll über ihre Verwendung. Diese Funktion wird jedoch an mehreren Stellen verwendet, und es ist erwünscht, dass bei der Erzeugung des Protokolls ein Teil der Informationen darin besteht, zu wissen, von wo die Funktion aufgerufen wird.

Wenn Sie die bedingte Kompilierung verwenden, können Sie in der Include-Datei so etwas wie das Folgende definieren, das die Funktion deklariert. Dies ersetzt die Standardversion der Funktion durch eine Debug-Version der Funktion. Der Präprozessor wird verwendet, um Aufrufe der Funktion `SerOpPluAllRead()` mit Aufrufen der Funktion `SerOpPluAllRead_Debug()` durch zwei zusätzliche Argumente zu ersetzen, den Dateinamen und die Zeilennummer der Funktion, in der die Funktion verwendet wird.

Bei der bedingten Kompilierung wird ausgewählt, ob die Standardfunktion mit einer Debugversion überschrieben werden soll oder nicht.

```
#if 0
```

```
// function declaration and prototype for our debug version of the function.
SHORT   SerOpPluAllRead_Debug(PLUIF *pPif, USHORT usLockHnd, char *aszFilePath, int nLineNo);

// macro definition to replace function call using old name with debug function with
additional arguments.
#define SerOpPluAllRead(pPif,usLock) SerOpPluAllRead_Debug(pPif,usLock,__FILE__,__LINE__)
#else
// standard function declaration that is normally used with builds.
SHORT   SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd);
#endif
```

Auf diese Weise können Sie die Standardversion der Funktion `SerOpPluAllRead()` mit einer Version überschreiben, die den Namen der Datei und die Zeilennummer in der Datei `SerOpPluAllRead()` in der die Funktion aufgerufen wird.

Es gibt eine wichtige Überlegung: *Jede Datei, die diese Funktion verwendet, muss die Header-Datei enthalten, bei der dieser Ansatz verwendet wird, damit der Präprozessor die Funktion ändern kann. Andernfalls wird ein Linker-Fehler angezeigt.*

Die Definition der Funktion würde ungefähr wie folgt aussehen. Diese Quelle fordert an, dass der Präprozessor die Funktion `SerOpPluAllRead()` in `SerOpPluAllRead_Debug()` umbenennt und die Argumentliste so ändert, dass sie zwei zusätzliche Argumente enthält, einen Zeiger auf den Namen der Datei, in der die Funktion aufgerufen wurde, und die Zeilennummer in der Datei, in der die Funktion verwendet wird.

```
#if defined(SerOpPluAllRead)
// forward declare the replacement function which we will call once we create our log.
SHORT    SerOpPluAllRead_Special(PLUIF *pPif, USHORT usLockHnd);

SHORT    SerOpPluAllRead_Debug(PLUIF *pPif, USHORT usLockHnd, char *aszFilePath, int nLineNo)
{
    int iLen = 0;
    char  xBuffer[256];

    // only print the last 30 characters of the file name to shorten the logs.
    iLen = strlen (aszFilePath);
    if (iLen > 30) {
        iLen = iLen - 30;
    }
    else {
        iLen = 0;
    }

    sprintf (xBuffer, "SerOpPluAllRead_Debug(): husHandle = %d, File %s, lineno = %d", pPif-
>husHandle, aszFilePath + iLen, nLineNo);
    IssueDebugLog(xBuffer);

    // now that we have issued the log, continue with standard processing.
    return SerOpPluAllRead_Special(pPif, usLockHnd);
}

// our special replacement function name for when we are generating logs.
SHORT    SerOpPluAllRead_Special(PLUIF *pPif, USHORT usLockHnd)
#else
// standard, normal function name (signature) that is replaced with our debug version.
SHORT   SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd)
#endif
```

```
{
    if (STUB_SELF == SstReadAsMaster()) {
        return OpPluAllRead(pPif, usLockHnd);
    }
    return OP_NOT_MASTER;
}
```

Quelldatei-Aufnahme

Die häufigsten Verwendungen von `#include` Vorverarbeitungsrichtlinien sind wie folgt:

```
#include <stdio.h>
#include "myheader.h"
```

`#include` ersetzt die Anweisung durch den Inhalt der Datei, auf die verwiesen wird. Eckige Klammern (<>) beziehen sich auf im System installierte Header-Dateien, während Anführungszeichen ("") für vom Benutzer angegebene Dateien stehen.

Makros selbst können andere Makros einmal erweitern, wie dieses Beispiel veranschaulicht:

```
#if VERSION == 1
    #define INCFILE  "vers1.h"
#elif VERSION == 2
    #define INCFILE  "vers2.h"
    /*  and so on */
#else
    #define INCFILE  "versN.h"
#endif
/* ... */
#include INCFILE
```

Makro-Ersatz

Die einfachste Form der Makroersetzung besteht darin, eine `manifest constant` wie in zu definieren

```
#define ARRSIZE 100
int array[ARRSIZE];
```

Dies definiert ein *funktionsähnliches* Makro, das eine Variable mit `10` multipliziert und den neuen Wert speichert:

```
#define TIMES10(A) ((A) *= 10)

double b = 34;
int c = 23;

TIMES10(b);   // good: ((b) *= 10);
TIMES10(c);   // good: ((c) *= 10);
TIMES10(5);   // bad:  ((5) *= 10);
```

Die Ersetzung erfolgt vor jeder anderen Interpretation des Programmtextes. Beim ersten Aufruf von `TIMES10` der Name `A` aus der Definition durch `b` und der so erweiterte Text wird an die Stelle

des Aufrufs gesetzt. Beachten Sie, dass diese Definition von `TIMES10` nicht äquivalent zu ist

```
#define TIMES10(A) ((A) = (A) * 10)
```

weil dies den Ersatz von `A` zweimal auswerten könnte, was unerwünschte Nebenwirkungen haben kann.

Im Folgenden wird ein funktionsähnliches Makro definiert, dessen Wert das Maximum seiner Argumente ist. Es hat die Vorteile, für alle kompatiblen Typen der Argumente zu arbeiten und Inline-Code zu generieren, ohne den Funktionsaufruf zu verursachen. Es hat den Nachteil, das eine oder andere seiner Argumente ein zweites Mal zu bewerten (einschließlich Nebenwirkungen) und bei mehrmaligem Aufruf mehr Code als eine Funktion zu generieren.

```
#define max(a, b) ((a) > (b) ? (a) : (b))

int maxVal = max(11, 43);              /* 43 */
int maxValExpr = max(11 + 36, 51 - 7); /* 47 */

/* Should not be done, due to expression being evaluated twice */
int j = 0, i = 0;
int sideEffect = max(++i, ++j);       /* i == 4 */
```

Aus diesem Grund werden Makros, die ihre Argumente mehrfach auswerten, im Produktionscode normalerweise vermieden. Seit C11 gibt es die `_Generic` Funktion, mit der solche mehrfachen `_Generic` können.

Die reichlichen Klammern in den Makroerweiterungen (rechte Seite der Definition) stellen sicher, dass die Argumente und der resultierende Ausdruck ordnungsgemäß gebunden sind und gut in den Kontext passen, in dem das Makro aufgerufen wird.

Fehleranweisung

Wenn der Präprozessor auf eine `#error` Anweisung `#error` , wird die Kompilierung angehalten und die enthaltene Diagnosemeldung wird gedruckt.

```
#define DEBUG

#ifdef DEBUG
#error "Debug Builds Not Supported"
#endif

int main(void) {
    return 0;
}
```

Mögliche Ausgabe:

```
$ gcc error.c
error.c: error: #error "Debug Builds Not Supported"
```

#if 0, um Codeabschnitte auszublenden

Wenn Sie Codeabschnitte in Betracht ziehen, die Sie entfernen möchten oder vorübergehend deaktivieren möchten, können Sie sie mit einem Blockkommentar auskommentieren.

```
/* Block comment around whole function to keep it from getting used.
 * What's even the purpose of this function?
int myUnusedFunction(void)
{
    int i = 5;
    return i;
}
*/
```

Wenn der Quellcode, den Sie mit einem Blockkommentar umgeben haben, im Quellcode Blockkommentare enthält, kann die Endung * / der vorhandenen Blockkommentare dazu führen, dass Ihr neuer Blockkommentar ungültig wird und zu Kompilierungsproblemen führt.

```
/* Block comment around whole function to keep it from getting used.
 * What's even the purpose of this function?
int myUnusedFunction(void)
{
    int i = 5;

    /* Return 5 */
    return i;
}
*/
```

Im vorherigen Beispiel werden die letzten beiden Zeilen der Funktion und die letzten '* /' vom Compiler gesehen, sodass sie mit Fehlern kompiliert werden könnten. Eine sicherere Methode ist die Verwendung einer `#if 0` Direktive um den Code, den Sie blockieren möchten.

```
#if 0
/* #if 0 evaluates to false, so everything between here and the #endif are
 * removed by the preprocessor. */
int myUnusedFunction(void)
{
    int i = 5;
    return i;
}
#endif
```

Dies hat den Vorteil, dass es einfacher ist, nach "#if 0" zu suchen, als nach allen Kommentaren zu suchen, wenn Sie den Code wiederfinden möchten.

Ein weiterer sehr wichtiger Vorteil ist, dass Sie den Code mit `#if 0` . Dies ist nicht mit Kommentaren möglich.

Eine Alternative zur Verwendung von `#if 0` besteht darin, einen Namen zu verwenden, der nicht `#defined` aber mehr beschreibt, warum der Code blockiert wird. Wenn es beispielsweise eine Funktion gibt, die scheinbar unbrauchbar ist, können Sie `#if defined(POSSIBLE_DEAD_CODE)` oder `#if defined(FUTURE_CODE_REL_020201)` für Code verwenden, der benötigt wird, wenn eine andere

Funktionalität vorhanden ist oder ähnliches. Wenn Sie dann diese Quelle erneut entfernen oder aktivieren, sind diese Quellbereiche leicht zu finden.

Token-Einfügen

Beim Token-Einfügen können zwei Makro-Argumente miteinander verbunden werden. Zum Beispiel ergibt `front##back frontback front##back frontback` . Ein bekanntes Beispiel ist der `<TCHAR.H>` -Header von Win32. In Standard C kann man `L"string"` schreiben, um eine breite Zeichenkette zu deklarieren. Allerdings ermöglicht Windows - API einen weiten Zeichenketten und schmalen Zeichenketten einfach durch konvertieren `#define` ing `UNICODE` . Um die String-Literale zu implementieren, verwendet `TCHAR.H` dies

```
#ifdef UNICODE
#define TEXT(x) L##x
#endif
```

Wenn ein Benutzer `TEXT("hello, world")` schreibt `TEXT("hello, world")` und UNICODE definiert ist, verkettet der C-Präprozessor `L` und das Makro-Argument. `L` mit `"hello, world"` verkettet gibt `L"hello, world"` .

Vordefinierte Makros

Ein vordefiniertes Makro ist ein Makro, das vom C-Pre-Prozessor bereits verstanden wird, ohne dass das Programm ihn definieren muss. Beispiele beinhalten

Obligatorische vordefinierte Makros

- `__FILE__` , die den Dateinamen der aktuellen Quelldatei (ein String-Literal) `__FILE__` ,
- `__LINE__` für die aktuelle Zeilennummer (eine Ganzzahlkonstante),
- `__DATE__` für das Kompilierungsdatum (ein String-Literal),
- `__TIME__` für die Kompilierungszeit (ein String-Literal).

Es gibt auch einen verwandten vordefinierten Bezeichner `__func__` (ISO / IEC 9899: 2011 §6.4.2.2), der *kein* Makro ist:

> Der Bezeichner `__func__` ist vom Übersetzer implizit zu deklarieren, als würde die Deklaration unmittelbar auf die öffnenden Klammern jeder Funktionsdefinition folgen:
>
> ```
> static const char __func__[] = "function-name";
> ```
>
> erschienen, wobei *Funktionsname* der Name der lexikalisch einschließenden Funktion ist.

`__FILE__` , `__LINE__` und `__func__` sind besonders für Debugging-Zwecke nützlich. Zum Beispiel:

```
fprintf(stderr, "%s: %s: %d: Denominator is 0", __FILE__, __func__, __LINE__);
```

Pre-C99-Compiler unterstützen möglicherweise `__func__` oder verfügen möglicherweise nicht über ein Makro, das dasselbe `__func__` und anders benannt wird. Beispielsweise verwendete gcc `__FUNCTION__` im C89-Modus.

Mit den folgenden Makros können Sie Details zur Implementierung abfragen:

- `__STDC_VERSION__` Die Version des C-Standards wurde implementiert. Dies ist eine konstante Ganzzahl, die das Format `yyyymmL` (der Wert `201112L` für C11, der Wert `199901L` für C99; es wurde nicht für C89 / C90 definiert).
- `__STDC_HOSTED__` `1` wenn es sich um eine gehostete Implementierung handelt, andernfalls `0` .
- `__STDC__` Wenn `1` , entspricht die Implementierung dem C-Standard.

Andere vordefinierte Makros (nicht obligatorisch)

ISO / IEC 9899: 2011 §6.10.9.2 Umgebungsmakros:

- `__STDC_ISO_10646__` Eine Ganzzahlkonstante der Form `yyyymmL` (zum Beispiel 199712L). Wenn dieses Symbol definiert ist, hat jedes Zeichen in der erforderlichen Unicode-Gruppe, wenn es in einem Objekt vom Typ `wchar_t` gespeichert ist, denselben Wert wie der Kurzbezeichner dieses Zeichens. Das erforderliche Unicode-Set besteht aus allen Zeichen, die in der ISO / IEC 10646 definiert sind, einschließlich aller Änderungen und technischen Korrelationen ab dem angegebenen Jahr und Monat. Wenn eine andere Codierung verwendet wird, darf das Makro nicht definiert werden und die tatsächliche Codierung ist implementierungsdefiniert.

- `__STDC_MB_MIGHT_NEQ_WC__` Die Integer-Konstante 1, die angeben soll, dass ein Member des Basis-Zeichensatzes in der Codierung für `wchar_t` keinen Codewert haben muss, der seinem Wert entspricht, wenn er als Einzelzeichen in einer Integer-Zeichenkonstante verwendet wird.

- `__STDC_UTF_16__` Die Ganzzahlkonstante 1, die anzeigen soll, dass Werte vom Typ `char16_t` UTF-16-codiert sind. Wenn eine andere Codierung verwendet wird, darf das Makro nicht definiert werden und die tatsächliche Codierung ist implementierungsdefiniert.

- `__STDC_UTF_32__` Die Ganzzahlkonstante 1, die anzeigen soll, dass Werte vom Typ `char32_t` UTF-32-codiert sind. Wenn eine andere Codierung verwendet wird, darf das Makro nicht definiert werden und die tatsächliche Codierung ist implementierungsdefiniert.

ISO / IEC 9899: 2011 §6.10.8.3 Makros für bedingte Merkmale

- `__STDC_ANALYZABLE__` Die Ganzzahlkonstante 1, die die Übereinstimmung mit den Spezifikationen in Anhang L (Analysierbarkeit) anzeigen soll.
- `__STDC_IEC_559__` Die Ganzzahlkonstante 1, die die Übereinstimmung mit den Spezifikationen in Anhang F (IEC 60559-Gleitkomma-Arithmetik) anzeigen soll.
- `__STDC_IEC_559_COMPLEX__` Die Ganzzahlkonstante 1, die die Einhaltung der

Spezifikationen in Anhang G (IEC 60559-kompatible komplexe Arithmetik) anzeigen soll.

- `__STDC_LIB_EXT1__` Die Ganzzahlkonstante `201112L` , die die Unterstützung für die in Anhang K definierten Erweiterungen (Bounds-testing Interfaces) `201112L` soll.
- `__STDC_NO_ATOMICS__` Die Ganzzahlkonstante 1, die darauf hinweist, dass die Implementierung atomare Typen (einschließlich des `_Atomic` Typ-Qualifiers) und den Header `<stdatomic.h>` nicht unterstützt.
- `__STDC_NO_COMPLEX__` Die Ganzzahlkonstante 1, die darauf hinweist, dass die Implementierung keine komplexen Typen oder den Header `<complex.h>` .
- `__STDC_NO_THREADS__` Die Ganzzahlkonstante 1, die darauf hinweisen soll, dass die Implementierung den Header `<threads.h>` nicht unterstützt.
- `__STDC_NO_VLA__` Die Ganzzahlkonstante 1, die darauf hinweist, dass die Implementierung Arrays mit variabler Länge oder variabel modifizierte Typen nicht unterstützt.

Header Include Guards

So ziemlich jede Header-Datei sollte dem Include Guard- Idiom folgen:

meine-Header-Datei.h

```
#ifndef MY_HEADER_FILE_H
#define MY_HEADER_FILE_H

// Code body for header file

#endif
```

Dadurch wird sichergestellt, dass Sie, wenn Sie `#include "my-header-file.h"` an mehreren Stellen `#include "my-header-file.h"` , keine doppelten Deklarationen von Funktionen, Variablen usw. erhalten. Stellen Sie sich die folgende Hierarchie der Dateien vor:

header-1.h

```
typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);
```

header-2.h

```
#include "header-1.h"

int myFunction2(MyStruct *value);
```

Haupt c

```
#include "header-1.h"
#include "header-2.h"
```

```
int main() {
    // do something
}
```

Dieser Code hat ein schwerwiegendes Problem: Der detaillierte Inhalt von `MyStruct` ist zweimal definiert, was nicht zulässig ist. Dies würde zu einem Kompilierungsfehler führen, der schwer zu finden ist, da eine Headerdatei eine andere enthält. Wenn Sie es stattdessen mit Header Guards getan haben:

header-1.h

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#endif
```

header-2.h

```
#ifndef HEADER_2_H
#define HEADER_2_H

#include "header-1.h"

int myFunction2(MyStruct *value);

#endif
```

Haupt c

```
#include "header-1.h"
#include "header-2.h"

int main() {
    // do something
}
```

Dies würde sich dann erweitern auf:

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#endif
```

```
#ifndef HEADER_2_H
#define HEADER_2_H

#ifndef HEADER_1_H // Safe, since HEADER_1_H was #define'd before.
#define HEADER_1_H

typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#endif

int myFunction2(MyStruct *value);

#endif

int main() {
    // do something
}
```

Wenn der Compiler die zweite Aufnahme von **Header-1.h erreicht** , wurde `HEADER_1_H` bereits durch die vorherige Aufnahme definiert. Ergo läuft es auf folgendes hinaus:

```
#define HEADER_1_H

typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#define HEADER_2_H

int myFunction2(MyStruct *value);

int main() {
    // do something
}
```

Und somit gibt es keinen Kompilierungsfehler.

Hinweis: Es gibt mehrere verschiedene Konventionen für die Benennung der Kopfschutzbügel. Einige Leute nennen es `HEADER_2_H_` , andere enthalten den Projektnamen wie `MY_PROJECT_HEADER_2_H` . Es ist wichtig, sicherzustellen, dass die Konvention, an die Sie sich halten, dafür sorgt, dass jede Datei in Ihrem Projekt über einen eindeutigen Header Guard verfügt.

Wenn die Strukturdetails nicht in der Kopfzeile enthalten wären, wäre der deklarierte Typ unvollständig oder ein undurchsichtiger Typ . Solche Typen können nützlich sein, um Implementierungsdetails vor Benutzern der Funktionen zu verbergen. Für viele Zwecke kann der Typ `FILE` in der Standard-C-Bibliothek als undurchsichtiger Typ angesehen werden (obwohl er normalerweise nicht undurchsichtig ist, sodass Makroimplementierungen der Standard-E / A-Funktionen die internen Komponenten der Struktur verwenden können). In diesem Fall könnte der

`header-1.h` enthalten:

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct MyStruct MyStruct;

int myFunction(MyStruct *value);

#endif
```

Beachten Sie, dass die Struktur einen Tag-Namen haben muss (hier `MyStruct` - das ist im Namespace der Tags getrennt vom gewöhnlichen Bezeichner-Namespace des Typedef-Namens `MyStruct`), und dass `{ … }` weggelassen wird. Dies besagt "es gibt einen Strukturtyp `struct MyStruct` und einen Alias für `MyStruct` ".

In der Implementierungsdatei können die Details der Struktur definiert werden, um den Typ zu vervollständigen:

```
struct MyStruct {
    …
};
```

Wenn Sie C11 verwenden, können Sie die `typedef struct MyStruct MyStruct;` wiederholen `typedef struct MyStruct MyStruct;` Deklaration ohne einen Kompilierungsfehler zu verursachen, aber frühere Versionen von C würden sich beschweren. Daher ist es am besten, das Include-Guard-Idiom zu verwenden, obwohl es in diesem Beispiel optional wäre, wenn der Code immer nur mit Compilern kompiliert wurde, die C11 unterstützten.

Viele Compiler unterstützen die `#pragma once` Direktive, die zu den gleichen Ergebnissen führt:

meine-Header-Datei.h

```
#pragma once

// Code for header file
```

`#pragma once` ist jedoch nicht Teil des C-Standards. `#pragma once` ist der Code weniger portierbar, wenn Sie ihn verwenden.

Einige Header verwenden nicht das Include-Guard-Idiom. Ein konkretes Beispiel ist der Standardheader `<assert.h>` . Es kann mehrmals in einer einzigen Übersetzungseinheit enthalten sein, und die Auswirkung davon hängt davon ab, ob das Makro `NDEBUG` bei jeder Aufnahme des Headers definiert wird. Sie haben gelegentlich eine analoge Anforderung. Solche Fälle werden selten sein. Normalerweise sollten Ihre Header durch das Include-Wächter-Idiom geschützt werden.

FOREACH-Implementierung

Wir können auch Makros verwenden, um das Lesen und Schreiben von Code zu erleichtern. Wir können beispielsweise Makros implementieren, um das `foreach` Konstrukt in C für einige Datenstrukturen wie einfach und doppelt verknüpfte Listen, Warteschlangen usw. zu implementieren.

Hier ist ein kleines Beispiel.

```
#include <stdio.h>
#include <stdlib.h>

struct LinkedListNode
{
    int data;
    struct LinkedListNode *next;
};

#define FOREACH_LIST(node, list) \
     for (node=list; node; node=node->next)

/* Usage */
int main(void)
{
    struct LinkedListNode *list, **plist = &list, *node;
    int i;

    for (i=0; i<10; i++)
    {
         *plist = malloc(sizeof(struct LinkedListNode));
         (*plist)->data = i;
         (*plist)->next = NULL;
         plist          = &(*plist)->next;
    }

    /* printing the elements here */
    FOREACH_LIST(node, list)
    {
        printf("%d\n", node->data);
    }
}
```

Sie können eine Standardschnittstelle für solche Datenstrukturen erstellen und eine generische Implementierung von `FOREACH` schreiben als:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct CollectionItem_
{
    int data;
    struct CollectionItem_ *next;
} CollectionItem;

typedef struct Collection_
{
    /* interface functions */
    void* (*first)(void *coll);
    void* (*last) (void *coll);
    void* (*next) (void *coll, CollectionItem *currItem);
```

```
    CollectionItem *collectionHead;
    /* Other fields */
} Collection;

/* must implement */
void *first(void *coll)
{
    return ((Collection*)coll)->collectionHead;
}

/* must implement */
void *last(void *coll)
{
    return NULL;
}

/* must implement */
void *next(void *coll, CollectionItem *curr)
{
    return curr->next;
}

CollectionItem *new_CollectionItem(int data)
{
    CollectionItem *item = malloc(sizeof(CollectionItem));
    item->data = data;
    item->next = NULL;
    return item;
}

void Add_Collection(Collection *coll, int data)
{
    CollectionItem **item = &coll->collectionHead;
    while(*item)
        item = &(*item)->next;
    (*item) = new_CollectionItem(data);
}

Collection *new_Collection()
{
    Collection *nc = malloc(sizeof(Collection));
    nc->first = first;
    nc->last  = last;
    nc->next  = next;
    return nc;
}

/* generic implementation */
#define FOREACH(node, collection)                      \
    for (node  = (collection)->first(collection);      \
         node != (collection)->last(collection);       \
         node  = (collection)->next(collection, node))

int main(void)
{
    Collection *coll = new_Collection();
    CollectionItem *node;
    int i;

    for(i=0; i<10; i++)
```

```
    {
         Add_Collection(coll, i);
    }

    /* printing the elements here */
    FOREACH(node, coll)
    {
        printf("%d\n", node->data);
    }
}
```

Um diese generische Implementierung zu verwenden, implementieren Sie diese Funktionen einfach für Ihre Datenstruktur.

```
1.  void* (*first)(void *coll);
2.  void* (*last) (void *coll);
3.  void* (*next) (void *coll, CollectionItem *currItem);
```

__cplusplus für die Verwendung von C-Externen in C ++ - Code, zusammengestellt mit C ++ - Namensverstümmelung

Es gibt Zeiten, in denen eine Include-Datei unterschiedliche Ausgaben vom Präprozessor erzeugen muss, abhängig davon, ob der Compiler aufgrund von Sprachunterschieden ein C-Compiler oder ein C ++ - Compiler ist.

Beispielsweise ist eine Funktion oder eine andere externe in einer C-Quelldatei definiert, wird jedoch in einer C ++ - Quelldatei verwendet. Da C ++ die Namensveränderung (oder Namensdekoration) verwendet, um eindeutige Funktionsnamen basierend auf Funktionsargumenttypen zu generieren, führt eine in einer C ++ - Quelldatei verwendete Deklaration der C-Funktion zu Verbindungsfehlern. Der C ++ - Compiler ändert den angegebenen externen Namen für die Compiler-Ausgabe unter Verwendung der Regeln zur Namensänderung für C ++. Das Ergebnis sind Verbindungsfehler aufgrund von Externen, die nicht gefunden wurden, wenn die C ++ - Compilerausgabe mit der C-Compilerausgabe verknüpft ist.

Da C-Compiler keine Namensänderung vornehmen, C ++ - Compiler jedoch für alle vom C ++ - Compiler generierten externen Labels (Funktionsnamen oder Variablennamen), wurde ein vordefiniertes Präprozessor-Makro, `__cplusplus` , eingeführt, um die Compiler-Erkennung zu ermöglichen.

Um dieses Problem der inkompatiblen Compilerausgabe für externe Namen zwischen C und C ++ zu `__cplusplus` ist das Makro `__cplusplus` im C ++ - Präprozessor definiert und nicht im C-Präprozessor definiert. Dieser `#ifdef` kann mit der bedingten Präprozessor-Direktive `#ifdef` oder `#if` mit dem Operator `defined()` werden, um festzustellen, ob eine Quellcode- oder Include-Datei als C ++ oder C kompiliert wird.

```
#ifdef __cplusplus
printf("C++\n");
#else
printf("C\n");
#endif
```

Oder du könntest es gebrauchen

```
#if defined(__cplusplus)
printf("C++\n");
#else
printf("C\n");
#endif
```

Um den korrekten Funktionsnamen einer Funktion aus einer C-Quelldatei anzugeben, die mit dem C-Compiler kompiliert wurde, der in einer C ++ - Quelldatei verwendet wird, können Sie nach der definierten Konstante `__cplusplus` , um das `extern "C" { /* ... */ };` zur Deklaration von C externals, wenn die Headerdatei in einer C ++ - Quelldatei enthalten ist. Beim Kompilieren mit einem C-Compiler wird jedoch das `extern "C" { */ ... */ };` ist nicht benutzt. Diese bedingte Kompilierung ist erforderlich, da `extern "C" { /* ... */ };` ist in C ++ gültig, aber nicht in C.

```
#ifdef __cplusplus
// if we are being compiled with a C++ compiler then declare the
// following functions as C functions to prevent name mangling.
extern "C" {
#endif

// exported C function list.
int foo (void);

#ifdef __cplusplus
// if this is a C++ compiler, we need to close off the extern declaration.
};
#endif
```

Funktionsartige Makros

Funktionsähnliche Makros ähneln `inline` Funktionen. Diese sind in einigen Fällen nützlich, beispielsweise im temporären Debug-Protokoll:

```
#ifdef DEBUG
# define LOGFILENAME "/tmp/logfile.log"

# define LOG(str) do {                            \
  FILE *fp = fopen(LOGFILENAME, "a");            \
  if (fp) {                                       \
    fprintf(fp, "%s:%d %s\n", __FILE__, __LINE__, \
                /* don't print null pointer */    \
                str ?str :"<null>");              \
    fclose(fp);                                   \
  }                                               \
  else {                                          \
    perror("Opening '" LOGFILENAME "' failed");   \
  }                                               \
} while (0)
#else
  /* Make it a NOOP if DEBUG is not defined. */
# define LOG(LINE) (void)0
#endif
```

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    if (argc > 1)
        LOG("There are command line arguments");
    else
        LOG("No command line arguments");
    return 0;
}
```

In beiden Fällen verhält sich der Aufruf in beiden Fällen (mit `DEBUG` oder nicht) wie eine Funktion mit `void` Rückgabetyp. Dadurch wird sichergestellt, dass die `if/else` Bedingungen wie erwartet interpretiert werden.

Im `DEBUG` Fall wird dies durch ein `do { ... } while(0) DEBUG` implementiert. Im anderen Fall ist `(void)0` eine Anweisung ohne Nebeneffekt, die einfach ignoriert wird.

Eine Alternative für Letzteres wäre

```
#define LOG(LINE) do { /* empty */ } while (0)
```

so, dass es in allen Fällen dem ersten syntaktisch entspricht.

Wenn Sie GCC verwenden, können Sie auch ein funktionsähnliches Makro implementieren, das das Ergebnis mit nicht standardmäßigen Ausdrücken der GNU-Erweiterungsanweisung zurückgibt. Zum Beispiel:

```
#include <stdio.h>

#define POW(X, Y) \
({ \
        int i, r = 1; \
        for (i = 0; i < Y; ++i) \
                r *= X; \
        r; \ // returned value is result of last operation
})

int main(void)
{
        int result;

        result = POW(2, 3);
        printf("Result: %d\n", result);
}
```

Variadische Argumente Makro

C99

Makros mit variablen Args:

Angenommen, Sie möchten ein Druckmakro erstellen, um Ihren Code zu debuggen. Nehmen wir

dieses Makro als Beispiel:

```
#define debug_print(msg) printf("%s:%d %s", __FILE__, __LINE__, msg)
```

Einige Anwendungsbeispiele:

Die Funktion `somefunc()` gibt -1 zurück, wenn sie nicht bestanden hat, und 0, wenn sie erfolgreich war, und sie wird an vielen Stellen im Code aufgerufen:

```
int retVal = somefunc();

if(retVal == -1)
{
    debug_printf("somefunc() has failed");
}

/* some other code */

 retVal = somefunc();

if(retVal == -1)
{
    debug_printf("somefunc() has failed");
}
```

Was passiert, wenn sich die Implementierung von `somefunc()` ändert und jetzt unterschiedliche Werte zurückgibt, die auf verschiedene mögliche Fehlertypen abgestimmt sind? Sie möchten weiterhin das Debug-Makro verwenden und den Fehlerwert drucken.

```
debug_printf(retVal);      /* this would obviously fail */
debug_printf("%d",retVal); /* this would also fail */
```

Um dieses Problem zu lösen, wurde das Makro `__VA_ARGS__` eingeführt. Dieses Makro erlaubt mehrere Parameter von X-Makros:

Beispiel:

```
 #define debug_print(msg, ...) printf(msg, __VA_ARGS__) \
                               printf("\nError occurred in file:line (%s:%d)\n", __FILE__,
__LINE)
```

Verwendungszweck:

```
int retVal = somefunc();

debug_print("retVal of somefunc() is-> %d", retVal);
```

Mit diesem Makro können Sie mehrere Parameter übergeben und drucken. Jetzt können Sie jedoch keine Parameter mehr senden.

```
debug_print("Hey");
```

Dies würde zu einem Syntaxfehler führen, da das Makro mindestens ein weiteres Argument erwartet und der Vorprozessor das fehlende Komma im Makro `debug_print()` nicht ignoriert. Auch `debug_print("Hey",);` Ein Syntaxfehler wird ausgelöst, da das an Makro übergebene Argument nicht leer bleiben kann.

Um dieses `##__VA_ARGS__` zu lösen, wurde das `##__VA_ARGS__` Makro eingeführt. Dieses Makro besagt, dass das Komma, wenn keine variablen Argumente vorhanden sind, vom Vorprozessor aus dem Code gelöscht wird.

Beispiel:

```
#define debug_print(msg, ...) printf(msg, ##__VA_ARGS__) \
                              printf("\nError occured in file:line (%s:%d)\n", __FILE__,
__LINE)
```

Verwendungszweck:

```
debug_print("Ret val of somefunc()?");
debug_print("%d",somefunc());
```

Preprozessor und Makros online lesen: https://riptutorial.com/de/c/topic/447/preprozessor-und-makros

Kapitel 41: Prozessübergreifende Kommunikation (IPC)

Einführung

Interprozesskommunikationsmechanismen (IPC) ermöglichen die Kommunikation verschiedener unabhängiger Prozesse miteinander. Standard C bietet keine IPC-Mechanismen. Daher werden alle derartigen Mechanismen vom Host-Betriebssystem definiert. POSIX definiert einen umfangreichen Satz von IPC-Mechanismen. Windows definiert ein anderes Set. und andere Systeme definieren ihre eigenen Varianten.

Examples

Semaphore

Semaphore werden verwendet, um Vorgänge zwischen zwei oder mehr Prozessen zu synchronisieren. POSIX definiert zwei verschiedene Sätze von Semaphorfunktionen:

1. 'System V IPC' - `semctl()` , `semop()` , `semget()` .
2. 'POSIX Semaphores' - `sem_close()` , `sem_destroy()` , `sem_getvalue()` , `sem_init()` , `sem_open()` , `sem_post()` , `sem_trywait()` , `sem_unlink()` .

In diesem Abschnitt werden die System V-IPC-Semaphore beschrieben, die so genannt werden, weil sie aus Unix System V stammen.

Zuerst müssen Sie die erforderlichen Header angeben. Alte Versionen von POSIX erforderten `#include <sys/types.h>` ; moderne POSIX und die meisten Systeme benötigen dies nicht.

```
#include <sys/sem.h>
```

Dann müssen Sie einen Schlüssel sowohl im übergeordneten Element als auch im untergeordneten Element definieren.

```
#define KEY 0x1111
```

Dieser Schlüssel muss in beiden Programmen gleich sein, da sie sich nicht auf dieselbe IPC-Struktur beziehen. Es gibt Möglichkeiten, einen vereinbarten Schlüssel zu generieren, ohne dessen Wert fest zu codieren.

Als Nächstes müssen Sie diesen Schritt je nach Compiler möglicherweise nicht ausführen: Deklaration einer Union für Semaphoroperationen.

```
union semun {
    int val;
```

```
    struct semid_ds *buf;
    unsigned short  *array;
};
```

Als nächstes definieren Sie Ihre *try* (`semwait`) und *(erhöhen* `semsignal`) Strukturen. Die Namen P und V stammen aus dem Niederländischen

```
struct sembuf p = { 0, -1, SEM_UNDO}; # semwait
struct sembuf v = { 0, +1, SEM_UNDO}; # semsignal
```

Holen Sie sich zunächst die ID für Ihr IPC-Semaphor.

```
int id;
// 2nd argument is number of semaphores
// 3rd argument is the mode (IPC_CREAT creates the semaphore set if needed)
if ((id = semget(KEY, 1, 0666 | IPC_CREAT) < 0) {
    /* error handling code */
}
```

Initialisieren Sie das Semaphor im übergeordneten Element mit einem Zähler von 1.

```
union semun u;
u.val = 1;
if (semctl(id, 0, SETVAL, u) < 0) { // SETVAL is a macro to specify that you're setting the
value of the semaphore to that specified by the union u
    /* error handling code */
}
```

Jetzt können Sie das Semaphor nach Bedarf dekrementieren oder erhöhen. Zu Beginn Ihres kritischen Abschnitts dekrementieren Sie den Zähler mit der Funktion `semop()` :

```
if (semop(id, &p, 1) < 0) {
    /* error handling code */
}
```

Um das Semaphor zu inkrementieren, verwenden Sie `&v` anstelle von `&p` :

```
if (semop(id, &v, 1) < 0) {
    /* error handling code */
}
```

Beachten Sie, dass jede Funktion bei Erfolg `0` und bei Fehler `-1` zurückgibt. Wenn diese Rückgabestatus nicht geprüft werden, kann dies zu verheerenden Problemen führen.

Beispiel 1.1: Rennen mit Fäden

Das untenstehende Programm hat einen Prozesszweig für ein `fork` Element, und Eltern und Kinder versuchen, Zeichen ohne Synchronisation auf das Terminal zu drucken.

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main()
{
    int pid;
    pid =  fork();
    srand(pid);
    if(pid < 0)
    {
        perror("fork"); exit(1);
    }
    else if(pid)
    {
        char *s = "abcdefgh";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
        }
    }
    else
    {
        char *s = "ABCDEFGH";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
        }
    }
}
```

Ausgabe (1. Lauf):

```
aAABaBCbCbDDcEEcddeFFGGHHeffgghh
```

(2. Lauf):

```
aabbccAABddBCeeCffgDDghEEhFFGGHH
```

Beim Kompilieren und Ausführen dieses Programms sollten Sie jedes Mal eine andere Ausgabe erhalten.

Beispiel 1.2: Vermeiden Sie das Rennen mit Semaphoren

Wenn Sie *Beispiel 1.1* für die Verwendung von Semaphoren modifizieren, haben wir:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define KEY 0x1111

union semun {
    int val;
    struct semid_ds *buf;
    unsigned short  *array;
};

struct sembuf p = { 0, -1, SEM_UNDO};
struct sembuf v = { 0, +1, SEM_UNDO};

int main()
{
    int id = semget(KEY, 1, 0666 | IPC_CREAT);
    if(id < 0)
    {
        perror("semget"); exit(11);
    }
    union semun u;
    u.val = 1;
    if(semctl(id, 0, SETVAL, u) < 0)
    {
        perror("semctl"); exit(12);
    }
    int pid;
    pid =  fork();
    srand(pid);
    if(pid < 0)
    {
        perror("fork"); exit(1);
    }
    else if(pid)
    {
        char *s = "abcdefgh";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            if(semop(id, &p, 1) < 0)
            {
                perror("semop p"); exit(13);
            }
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            if(semop(id, &v, 1) < 0)
            {
                perror("semop p"); exit(14);
            }
```

```
            sleep(rand() % 2);
        }
    }
    else
    {
        char *s = "ABCDEFGH";
        int l = strlen(s);
        for(int i = 0; i < l; ++i)
        {
            if(semop(id, &p, 1) < 0)
            {
                perror("semop p"); exit(15);
            }
            putchar(s[i]);
            fflush(stdout);
            sleep(rand() % 2);
            putchar(s[i]);
            fflush(stdout);
            if(semop(id, &v, 1) < 0)
            {
                perror("semop p"); exit(16);
            }

            sleep(rand() % 2);
        }
    }
}
```

Ausgabe:

```
aabbAABBCCccddeeDDffEEFFGGHHgghh
```

Wenn Sie dieses Programm kompilieren und ausführen, erhalten Sie jedes Mal dieselbe Ausgabe.

Prozessübergreifende Kommunikation (IPC) online lesen:
https://riptutorial.com/de/c/topic/10564/prozessubergreifende-kommunikation--ipc-

Kapitel 42: Signalverarbeitung

Syntax

- void (* signal (int sig, void (* func) (int))) (int);

Parameter

Parameter	Einzelheiten
sig	Das Signal, das den Signal-Handler auf " `SIGABRT` , " `SIGFPE` , " `SIGILL` , " `SIGTERM` , " `SIGINT` , " `SIGSEGV` oder einen durch die Implementierung definierten Wert setzen soll
func	Der Signalhandler, einer der folgenden: `SIG_DFL` für den Standardhandler, `SIG_IGN` zum Ignorieren des Signals oder ein Funktionszeiger mit der Signatur `void foo(int sig);` .

Bemerkungen

Die Verwendung von Signalhandlern mit nur den Garantien des C-Standards unterliegt verschiedenen Einschränkungen, was im benutzerdefinierten Signalhandler möglich oder nicht möglich ist.

- Wenn die benutzerdefinierte Funktion während der Behandlung von `SIGSEGV` , `SIGFPE` , `SIGILL` oder einem anderen implementierungsdefinierten Hardware-Interrupt zurückkehrt, ist das Verhalten durch den C-Standard nicht definiert. Dies liegt daran, dass die Schnittstelle von C keine Mittel zur Änderung des fehlerhaften Zustands gibt (z. B. nach einer Division durch `0`) und daher bei der Rückkehr des Programms genau der gleiche fehlerhafte Zustand ist wie vor dem Auftreten des Hardware-Interrupts.
- Wenn die benutzerdefinierte Funktion als Ergebnis eines `abort` oder `raise` aufgerufen wurde `raise` darf der Signalhandler nicht erneut `raise` aufrufen.
- Signale können während jeder Operation eintreffen, und daher kann die Unteilbarkeit der Operationen im Allgemeinen nicht garantiert werden, und die Signalverarbeitung funktioniert bei der Optimierung nicht gut. Daher müssen alle Änderungen an den Daten in einem Signalhandler an Variablen vorgenommen werden
    - vom Typ `sig_atomic_t` (alle Versionen) oder einem `sig_atomic_t` atomaren Typ (seit C11, optional)
    - das sind `volatile` qualifiziert.
- Bei anderen Funktionen der C-Standardbibliothek werden diese Einschränkungen normalerweise nicht beachtet, da sie möglicherweise Variablen im globalen Status des

Programms ändern. Der C-Standard gibt nur Garantien für `abort` , `_Exit` (seit C99), `quick_exit` (seit C11), `signal` (für dieselbe Signalnummer) und einige atomare Operationen (seit C11).

Das Verhalten ist vom C-Standard nicht definiert, wenn eine der oben genannten Regeln verletzt wird. Plattformen können über bestimmte Erweiterungen verfügen, diese sind jedoch im Allgemeinen über diese Plattform nicht portierbar.

- Normalerweise haben Systeme eine eigene Liste von Funktionen, die *asynchrones Signalsicher sind* , d. H. Von C-Bibliotheksfunktionen, die von einem Signalhandler verwendet werden können. ZB ist oft `printf` eine dieser Funktionen.
- Insbesondere definiert der C-Standard nicht viel über die Interaktion mit seiner Thread-Schnittstelle (seit C11) oder mit plattformspezifischen Thread-Bibliotheken wie POSIX-Threads. Solche Plattformen müssen die Interaktion solcher Thread-Bibliotheken mit eigenen Signalen festlegen.

Examples

Signalbehandlung mit "Signal ()"

Signalnummern können synchron sein (wie `SIGSEGV` - Segmentierungsfehler), wenn sie durch eine Fehlfunktion des Programms selbst ausgelöst werden, oder asynchron (wie `SIGINT` - interaktive Aufmerksamkeit), wenn sie von außerhalb des Programms initiiert werden, z. B. durch Tastendruck als `Cntrl-C` .

Die Funktion `signal()` ist Teil des ISO-C-Standards und kann verwendet werden, um eine Funktion für die Verarbeitung eines bestimmten Signals zuzuweisen

```
#include <stdio.h>  /* printf() */
#include <stdlib.h> /* abort()  */
#include <signal.h> /* signal() */

void handler_nonportable(int sig)
{
    /* undefined behavior, maybe fine on specific platform */
    printf("Catched: %d\n", sig);

    /* abort is safe to call */
    abort();
}

sig_atomic_t volatile finished = 0;

void handler(int sig)
{
    switch (sig) {
    /* hardware interrupts should not return */
    case SIGSEGV:
    case SIGFPE:
    case SIGILL:
```

C11

```
      /* quick_exit is safe to call */
      quick_exit(EXIT_FAILURE);
```

C11

```
      /* use _Exit in pre-C11 */
      _Exit(EXIT_FAILURE);
```

```
    default:
       /* Reset the signal to the default handler,
          so we will not be called again if things go
          wrong on return. */
      signal(sig, SIG_DFL);
      /* let everybody know that we are finished */
      finished = sig;
      return;
   }
}

int main(void)
{

    /* Catch the SIGSEGV signal, raised on segmentation faults (i.e NULL ptr access */
    if (signal(SIGSEGV, &handler) == SIG_ERR) {
        perror("could not establish handler for SIGSEGV");
        return EXIT_FAILURE;
    }

    /* Catch the SIGTERM signal, termination request */
    if (signal(SIGTERM, &handler) == SIG_ERR) {
        perror("could not establish handler for SIGTERM");
        return EXIT_FAILURE;
    }

    /* Ignore the SIGINT signal, by setting the handler to `SIG_IGN`. */
    signal(SIGINT, SIG_IGN);


    /* Do something that takes some time here, and leaves
       the time to terminate the program from the keyboard. */

    /* Then: */

    if (finished) {
       fprintf(stderr, "we have been terminated by signal %d\n", (int)finished);
        return EXIT_FAILURE;
    }


    /* Try to force a segmentation fault, and raise a SIGSEGV */
    {
      char* ptr = 0;
      *ptr = 0;
    }

    /* This should never be executed */
    return EXIT_SUCCESS;
}
```

Die Verwendung von `signal()` wichtige Einschränkungen, was Sie in den Signalhandlern tun dürfen. Weitere Informationen finden Sie in den Anmerkungen.

POSIX empfiehlt die Verwendung von `sigaction()` anstelle von `signal()` , da dessen Verhalten nicht genau angegeben ist und die Implementierung stark variiert. POSIX definiert auch viel mehr Signale als den ISO-C-Standard, einschließlich `SIGUSR1` und `SIGUSR2` , die vom Programmierer für beliebige Zwecke frei verwendet werden können.

Signalverarbeitung online lesen: https://riptutorial.com/de/c/topic/453/signalverarbeitung

Kapitel 43: Speicherklassen

Einführung

Eine Speicherklasse wird verwendet, um den Gültigkeitsbereich einer Variablen oder Funktion festzulegen. Indem wir die Speicherklasse einer Variablen kennen, können wir die Lebensdauer dieser Variablen während der Laufzeit des Programms bestimmen.

Syntax

- [auto | register | static | extern] <Datentyp> <Variablenname> [= <Wert>];
- [static _Thread_local | extern _Thread_local | _Thread_local] <Datentyp> <Variablenname> [= <Wert>]; / * seit> = C11 * /
- Beispiele:
- typedef int *foo* ;
- extern int *foo* [2];

Bemerkungen

Speicherklassenspezifizierer sind die Schlüsselwörter, die neben dem Typ einer Deklaration auf oberster Ebene angezeigt werden können. Die Verwendung dieser Schlüsselwörter wirkt sich auf die Speicherdauer und die Verknüpfung des deklarierten Objekts aus, je nachdem, ob es im Dateibereich oder im Blockbereich deklariert ist:

Stichwort	Lagerdauer	Verknüpfung	Bemerkungen
`static`	Statisch	Intern	Legt die interne Verknüpfung für Objekte im Dateibereich fest. Legt die statische Speicherdauer für Objekte im Blockbereich fest.
`extern`	Statisch	Externe	Impliziert und daher redundant für Objekte, die im Dateibereich definiert sind und auch über einen Initialisierer verfügen. Bei Verwendung in einer Deklaration im Dateibereich ohne Initialisierer weist dies darauf hin, dass die Definition in einer anderen Übersetzungseinheit zu finden ist und zum Zeitpunkt der Verknüpfung aufgelöst wird.
`auto`	Automatik	Irrelevant	Impliziert und daher für Objekte, die im Sperrbereich deklariert sind, redundant.

Stichwort	Lagerdauer	Verknüpfung	Bemerkungen
`register`	Automatik	Irrelevant	Nur relevant für Objekte mit automatischer Speicherdauer. Bietet einen Hinweis, dass die Variable in einem Register gespeichert werden soll. Eine auferlegte Einschränkung ist, dass der unäre `&` "address of" -Operator für ein solches Objekt nicht verwendet werden kann und das Objekt daher nicht als Alias bezeichnet werden kann.
`typedef`	Irrelevant	Irrelevant	In der Praxis kein Speicherklassenspezifizierer, er arbeitet jedoch aus syntaktischer Sicht wie einer. Der einzige Unterschied besteht darin, dass es sich bei dem deklarierten Bezeichner nicht um ein Objekt, sondern um einen Typ handelt.
`_Thread_local`	Faden	Intern extern	In C11 eingeführt, um die *Dauer des Thread-Speichers* darzustellen. Bei Verwendung im Blockumfang muss es auch `extern` oder `static` .

Jedes Objekt hat eine zugeordnete Speicherdauer (unabhängig vom Gültigkeitsbereich) und eine Verknüpfung (nur für Deklarationen im Dateibereich relevant), auch wenn diese Schlüsselwörter nicht angegeben werden.

Die Reihenfolge von Speicherklassenspezifizierern in Bezug auf Typspezifizierer auf oberster Ebene (`int` , `unsigned` , `short` usw.) und `const` oberster Ebene (`const` , `volatile`) wird nicht erzwungen. Daher sind diese beiden Deklarationen gültig:

```
int static const unsigned a = 5; /* bad practice */
static const unsigned int b = 5; /* good practice */
```

Es wird jedoch als bewährte Methode angesehen, zuerst Speicherklassenspezifizierer, dann alle Typqualifizierer und dann den Typbezeichner (`void` , `char` , `int` , `signed long` , `unsigned long long` , `long double` ...) anzugeben.

Nicht alle Speicherklassenspezifizierer sind in einem bestimmten Umfang zulässig:

```
register int x; /* legal at block scope, illegal at file scope */
auto int y; /* same */

static int z; /* legal at both file and block scope */
extern int a; /* same */

extern int b = 5; /* legal and redundant at file scope, illegal at block scope */

/* legal because typedef is treated like a storage class specifier syntactically */
int typedef new_type_name;
```

Lagerdauer

Die Lagerdauer kann entweder statisch oder automatisch sein. Für ein deklariertes Objekt wird es abhängig von seinem Gültigkeitsbereich und den Speicherklassenspezifizierern festgelegt.

Statische Speicherdauer

Variablen mit statischer Speicherdauer leben während der gesamten Ausführung des Programms und können sowohl im Dateibereich (mit oder ohne `static`) als auch im Blockbereich (durch explizites Setzen von `static`) deklariert werden. Sie werden normalerweise vom Betriebssystem beim Programmstart zugewiesen und initialisiert und nach Beendigung des Prozesses wieder freigegeben. In der Praxis haben ausführbare Formate dedizierte Abschnitte für solche Variablen (`data` , `bss` und `rodata`), und diese gesamten Abschnitte aus der Datei werden in bestimmten Bereichen in den Speicher abgebildet.

Thread-Speicherdauer

C11

Diese Lagerdauer wurde in C11 eingeführt. Dies war in früheren C-Standards nicht verfügbar. Einige Compiler bieten eine nicht standardmäßige Erweiterung mit ähnlicher Semantik. Zum Beispiel unterstützt gcc den `__thread` Bezeichner, der in früheren C-Standards verwendet werden kann, die kein `_Thread_local` .

Variablen mit Thread-Speicherdauer können sowohl im Dateibereich als auch im Blockbereich deklariert werden. Wenn für den Blockbereich deklariert, muss er auch einen `static` oder `extern` Speicherbezeichner verwenden. Ihre Lebensdauer ist die gesamte Ausführung des *Threads,* in dem er erstellt wurde. Dies ist der einzige Speicherbezeichner, der neben einem anderen Speicherbezeichner angezeigt werden kann.

Automatische Speicherdauer

Variablen mit automatischer Speicherdauer können nur im Blockbereich deklariert werden (direkt innerhalb einer Funktion oder innerhalb eines Blocks in dieser Funktion). Sie sind nur in der Zeit zwischen dem Betreten und Verlassen der Funktion oder des Blocks verwendbar. Sobald die Variable den Gültigkeitsbereich verlässt (entweder durch Rückkehr von der Funktion oder durch Verlassen des Blocks), wird der Speicher automatisch freigegeben. Alle weiteren Verweise auf dieselbe Variable aus Zeigern sind ungültig und führen zu undefiniertem Verhalten.

In typischen Implementierungen befinden sich automatische Variablen bei bestimmten Offsets im Stapelrahmen einer Funktion oder in Registern.

Externe und interne Verknüpfung

Die Verknüpfung ist nur für Objekte (Funktionen und Variablen) relevant, die im Dateibereich deklariert sind, und beeinflusst deren Sichtbarkeit in verschiedenen Übersetzungseinheiten. Objekte mit externer Verknüpfung sind in jeder anderen Übersetzungseinheit sichtbar (vorausgesetzt, die entsprechende Deklaration ist enthalten). Objekte mit interner Verknüpfung sind nicht für andere Übersetzungseinheiten verfügbar und können nur in der Übersetzungseinheit verwendet werden, in der sie definiert sind.

Examples

Typedef

Definiert einen neuen Typ basierend auf einem vorhandenen Typ. Ihre Syntax spiegelt die einer Variablendeklaration wider.

```
/* Byte can be used wherever `unsigned char` is needed */
typedef unsigned char Byte;

/* Integer is the type used to declare an array consisting of a single int */
typedef int Integer[1];

/* NodeRef is a type used for pointers to a structure type with the tag "node" */
typedef struct node *NodeRef;

/* SigHandler is the function pointer type that gets passed to the signal function. */
typedef void (*SigHandler)(int);
```

Ein Compiler wird zwar technisch keine Speicherklasse sein, aber als eine Klasse behandeln, da keine der anderen Speicherklassen zulässig ist, wenn das Schlüsselwort `typedef` verwendet wird.

Die `typedef` sind wichtig und sollten nicht durch das Makro `#define` .

```
typedef int newType;
newType *ptr;        // ptr is pointer to variable of type 'newType' aka int
```

Jedoch,

```
#define int newType
newType *ptr;        // Even though macros are exact replacements to words, this doesn't
result to a pointer to variable of type 'newType' aka int
```

Auto

Diese Speicherklasse gibt an, dass eine Kennung eine automatische Speicherdauer hat. Das heißt, sobald der Gültigkeitsbereich, in dem der Bezeichner definiert wurde, endet, ist das durch den Bezeichner gekennzeichnete Objekt nicht mehr gültig.

Da alle Objekte, die nicht im globalen Geltungsbereich leben oder als `static` deklariert werden, standardmäßig eine automatische Speicherdauer aufweisen, ist dieses Schlüsselwort meist von historischem Interesse und sollte nicht verwendet werden:

```
int foo(void)
{
    /* An integer with automatic storage duration. */
    auto int i = 3;

    /* Same */
    int j = 5;

    return 0;
} /* The values of i and j are no longer able to be used. */
```

statisch

Die `static` Speicherklasse erfüllt unterschiedliche Zwecke, abhängig vom Speicherort der Deklaration in der Datei:

1. Um die Kennung nur auf diese Übersetzungseinheit zu beschränken (Bereich = Datei).

```
/* No other translation unit can use this variable. */
static int i;

/* Same; static is attached to the function type of f, not the return type int. */
static int f(int n);
```

2. So speichern Sie Daten zur Verwendung beim nächsten Aufruf einer Funktion (Bereich = Block):

```
void foo()
{
    static int a = 0; /* has static storage duration and its lifetime is the
                       * entire execution of the program; initialized to 0 on
                       * first function call */
    int b = 0; /* b has block scope and has automatic storage duration and
                * only "exists" within function */

    a += 10;
    b += 10;

    printf("static int a = %d, int b = %d\n", a, b);
}

int main(void)
{
    int i;
    for (i = 0; i < 5; i++)
    {
        foo();
    }

    return 0;
}
```

Dieser Code druckt:

```
static int a = 10, int b = 10
```

```
static int a = 20, int b = 10
static int a = 30, int b = 10
static int a = 40, int b = 10
static int a = 50, int b = 10
```

Statische Variablen behalten ihren Wert, auch wenn sie von mehreren verschiedenen Threads aufgerufen werden.

C99

3. Wird in Funktionsparametern verwendet, um ein Array zu kennzeichnen, wird erwartet, dass es eine konstante Mindestanzahl von Elementen und einen Nicht-Null-Parameter aufweist:

```
/* a is expected to have at least 512 elements. */
void printInts(int a[static 512])
{
    size_t i;
    for (i = 0; i < 512; ++i)
        printf("%d\n", a[i]);
}
```

Die erforderliche Anzahl von Elementen (oder sogar ein Zeiger, der nicht Null ist) wird vom Compiler nicht unbedingt geprüft. Compiler müssen Sie nicht benachrichtigen, wenn Sie nicht genügend Elemente haben. Wenn ein Programmierer weniger als 512 Elemente oder einen Nullzeiger übergibt, ist das Ergebnis ein undefiniertes Verhalten. Da dies nicht erzwungen werden kann, muss besondere Sorgfalt angewandt werden, wenn ein Wert für diesen Parameter an eine solche Funktion übergeben wird.

extern

Wird verwendet **, um ein Objekt oder eine Funktion** zu **deklarieren** , die an anderer Stelle definiert ist (und die über eine *externe Verknüpfung verfügt*). Im Allgemeinen wird ein Objekt oder eine Funktion für die Verwendung in einem Modul deklariert, in dem das entsprechende Objekt oder die entsprechende Funktion nicht definiert ist:

```
/* file1.c */
int foo = 2;  /* Has external linkage since it is declared at file scope. */
```

```
/* file2.c */
#include <stdio.h>
int main(void)
{
    /* `extern` keyword refers to external definition of `foo`. */
    extern int foo;
    printf("%d\n", foo);
    return 0;
}
```

C99

Etwas interessanter wird es mit der Einführung des `inline` Keywords in C99:

```
/* Should usually be place in a header file such that all users see the definition */
/* Hints to the compiler that the function `bar` might be inlined */
/* and suppresses the generation of an external symbol, unless stated otherwise. */
inline void bar(int drink)
{
    printf("You ordered drink no.%d\n", drink);
}

/* To be found in just one .c file.
   Creates an external function definition of `bar` for use by other files.
   The compiler is allowed to choose between the inline version and the external
   definition when `bar` is called. Without this line, `bar` would only be an inline
   function, and other files would not be able to call it. */
extern void bar(int);
```

registrieren

Hinweise an den Compiler, dass der Zugriff auf ein Objekt so schnell wie möglich sein sollte. Ob der Compiler den Hinweis tatsächlich verwendet, ist durch die Implementierung definiert. es kann es einfach als äquivalent zu `auto` .

Die einzige Eigenschaft, die sich definitiv für alle mit `register` deklarierten Objekte unterscheidet, besteht darin, dass ihre Adresse nicht berechnet werden kann. Dabei kann `register` ein gutes Werkzeug sein, um bestimmte Optimierungen sicherzustellen:

```
register size_t size = 467;
```

ist ein Objekt, das niemals als *Alias bezeichnet werden kann,* da kein Code seine Adresse an eine andere Funktion übergeben kann, wo sie möglicherweise unerwartet geändert wird.

Diese Eigenschaft impliziert auch ein Array

```
register int array[5];
```

kann nicht in einen Zeiger auf sein erstes Element zerfallen (dh aus `array` in `&array[0]`). Dies bedeutet, dass auf die Elemente eines solchen Arrays nicht zugegriffen werden kann und das Array selbst nicht an eine Funktion übergeben werden kann.

Tatsächlich ist die einzige rechtliche Verwendung eines Arrays, das mit einer `register` deklariert wurde, der Operator `sizeof` . Jeder andere Operator würde die Adresse des ersten Elements des Arrays benötigen. Aus diesem Grund sollten Arrays im Allgemeinen nicht mit dem `register` Schlüsselwort deklariert werden, da sie für andere Zwecke als die Größenberechnung des gesamten Arrays unbrauchbar sind. Dies ist genauso einfach ohne das `register` Schlüsselwort.

Die `register` eignet sich besser für Variablen, die innerhalb eines Blocks definiert sind und auf die mit hoher Häufigkeit zugegriffen wird. Zum Beispiel,

```
/* prints the sum of the first 5 integers*/
/* code assumed to be part of a function body*/
{
    register int k, sum;
```

```
    for(k = 1, sum = 0; k < 6; sum += k, k++);
        printf("\t%d\n",sum);
}
```

C11

Der `_Alignof` Operator darf auch mit `register` Arrays verwendet werden.

_Thread_local

C11

Dies war ein neuer Speicherspezifizierer, der in C11 zusammen mit Multithreading eingeführt wurde. Dies ist in früheren C-Standards nicht verfügbar.

Gibt die *Thread-Speicherdauer an* . Eine mit `_Thread_local` Speicherkennzeichner deklarierte Variable gibt an, dass das Objekt *lokal für diesen Thread ist* und seine Lebensdauer die gesamte Ausführung des Threads ist, in dem es erstellt wurde. Es kann auch zusammen mit `static` oder `extern` .

```
#include <threads.h>
#include <stdio.h>
#define SIZE 5

int thread_func(void *id)
{
    /* thread local variable i. */
    static _Thread_local int i;

    /* Prints the ID passed from main() and the address of the i.
     * Running this program will print different addresses for i, showing
     * that they are all distinct objects. */
    printf("From thread:[%d], Address of i (thread local): %p\n", *(int*)id, (void*)&i);

    return 0;
}

int main(void)
{
    thrd_t id[SIZE];
    int arr[SIZE] = {1, 2, 3, 4, 5};

    /* create 5 threads. */
    for(int i = 0; i < SIZE; i++) {
        thrd_create(&id[i], thread_func, &arr[i]);
    }

    /* wait for threads to complete. */
    for(int i = 0; i < SIZE; i++) {
        thrd_join(id[i], NULL);
    }
}
```

Speicherklassen online lesen: https://riptutorial.com/de/c/topic/3597/speicherklassen

Kapitel 44: Speicherverwaltung

Einführung

Für die Verwaltung von dynamisch zugewiesenem Speicher stellt die Standard-C-Bibliothek die Funktionen `malloc()` , `calloc()` , `realloc()` und `free()` `calloc()` . In C99 und später gibt es auch `aligned_alloc()` . Einige Systeme bieten auch `alloca()` .

Syntax

- void * align_alloc (size_t-Ausrichtung, size_t-Größe); / * Erst seit C11 * /
- void * calloc (size_t nelements, size_t size);
- void free (void * ptr);
- void * malloc (size_t size);
- void * realloc (void * ptr, size_t size);
- void * alloca (size_t size); / * von alloca.h, nicht standardmäßig, nicht portabel, gefährlich. * /

Parameter

Name	Beschreibung
Größe (`malloc` , `realloc` und `aligned_alloc`)	Gesamtgröße des Speichers in Byte. Für `aligned_alloc` muss die Größe ein ganzzahliges Vielfaches der Ausrichtung sein.
Größe (`calloc`)	Größe jedes Elements
nelements	Anzahl der Elemente
ptr	Zeiger auf den zugewiesenen Speicher, der zuvor von `malloc` , `calloc` , `realloc` oder `aligned_alloc`
Ausrichtung	Ausrichtung des zugewiesenen Speichers

Bemerkungen

C11

Beachten Sie, dass `aligned_alloc()` nur für C11 oder höher definiert ist.

Systeme wie die auf POSIX basierenden Systeme bieten andere Möglichkeiten zum `posix_memalign()` von ausgerichtetem Speicher (z. B. `posix_memalign()`) sowie andere Speicherverwaltungsoptionen (z. B. `mmap()`).

Examples

Speicher freigeben

Sie können dynamisch zugewiesenen Speicher freigeben, indem Sie free () aufrufen.

```
int *p = malloc(10 * sizeof *p); /* allocation of memory */
if (p == NULL)
{
    perror("malloc failed");
    return -1;
}

free(p); /* release of memory */
/* note that after free(p), even using the *value* of the pointer p
   has undefined behavior, until a new value is stored into it. */

/* reusing/re-purposing the pointer itself */
int i = 42;
p = &i; /* This is valid, has defined behaviour */
```

Der Speicher, auf den `p` wird nach dem Aufruf von `free()` entweder von der libc-Implementierung oder vom zugrunde liegenden Betriebssystem zurückgefordert, sodass der Zugriff auf den freigegebenen Speicherblock über `p` zu undefiniertem Verhalten führt . Zeiger, die auf freigegebene Speicherelemente verweisen, werden im Allgemeinen als Schlenker bezeichnet und stellen ein Sicherheitsrisiko dar. Darüber hinaus gibt der C-Standard an, dass selbst der Zugriff auf den Wert eines baumelnden Zeigers undefiniertes Verhalten hat. Der Zeiger `p` selbst kann wie oben gezeigt neu bestimmt werden.

Bitte beachten Sie, dass Sie `free()` für Zeiger aufrufen können, die direkt von den Funktionen `malloc()` , `calloc()` , `realloc()` und `aligned_alloc()` , oder wo die Dokumentation Ihnen sagt, dass der Speicher auf diese Weise zugewiesen wurde (Funktionen wie `strdup ()` sind bemerkenswerte Beispiele). Einen Zeiger freigeben, der

- mit dem Operator `&` für eine Variable erhalten oder
- in der Mitte eines zugewiesenen Blocks

ist verboten. Ein solcher Fehler wird normalerweise nicht von Ihrem Compiler diagnostiziert, führt jedoch die Programmausführung in einen undefinierten Zustand.

Es gibt zwei gängige Strategien, um solche Fälle von undefiniertem Verhalten zu verhindern.

Der erste und vorzuziehende ist einfach - haben `p` selbst nicht mehr zu existieren, wenn es nicht mehr benötigt wird, zum Beispiel:

```
if (something_is_needed())
{

    int *p = malloc(10 * sizeof *p);
    if (p == NULL)
    {
        perror("malloc failed");
```

```
        return -1;
    }

    /* do whatever is needed with p */

    free(p);
}
```

Durch Aufrufen von `free()` direkt vor dem Ende des übergeordneten Blocks (dh des `}`) ist `p` nicht mehr vorhanden. Der Compiler gibt bei jedem Versuch, `p` danach zu verwenden, einen Kompilierungsfehler aus.

Ein zweiter Ansatz besteht darin, den Zeiger selbst nach dem Freigeben des Speichers, auf den er zeigt, ungültig zu machen:

```
free(p);
p = NULL;     // you may also use 0 instead of NULL
```

Argumente für diesen Ansatz:

- Auf vielen Plattformen führt der Versuch, einen Nullzeiger zu dereferenzieren, sofort zum Absturz: Segmentierungsfehler. Hier erhalten wir mindestens eine Stack-Ablaufverfolgung, die auf die Variable zeigt, die nach der Freigabe verwendet wurde.

  Ohne den Zeiger auf `NULL` , haben wir den Schlenker. Das Programm wird höchstwahrscheinlich immer noch abstürzen, aber später, weil der Speicher, auf den der Zeiger zeigt, unbemerkt beschädigt wird. Solche Fehler sind schwer zu verfolgen, da sie zu einem Aufrufstack führen können, der völlig unabhängig von dem ursprünglichen Problem ist.

  Dieser Ansatz folgt somit dem Fail-Fast-Konzept .

- Es ist sicher, einen Nullzeiger freizugeben. Der C-Standard legt fest, dass `free(NULL)` keine Auswirkung hat:

  > Die Funktion free bewirkt, dass der Raum, auf den ptr zeigt, freigegeben wird, dh für die weitere Zuweisung verfügbar ist. Wenn ptr ein Nullzeiger ist, wird keine Aktion ausgeführt. Andernfalls ist das Verhalten undefiniert, wenn das Argument nicht mit einem Zeiger `calloc` , der zuvor von der Funktion `calloc` , `malloc` oder `realloc` , oder wenn der Speicherplatz durch einen Aufruf von `free` oder `realloc` freigegeben wurde.

- Manchmal kann der erste Ansatz nicht verwendet werden (z. B. Speicher wird in einer Funktion zugewiesen und viel später in einer völlig anderen Funktion freigegeben).

Speicher zuweisen

Standardzuteilung

Die dynamischen Speicherzuordnungsfunktionen für C sind im Header `<stdlib.h>` definiert. Wenn Sie Speicherplatz für ein Objekt dynamisch zuweisen möchten, kann der folgende Code verwendet werden:

```
int *p = malloc(10 * sizeof *p);
if (p == NULL)
{
    perror("malloc() failed");
    return -1;
}
```

Dieser berechnet die Anzahl der Bytes, die zehn `int` s im Speicher belegen, fordert dann die Anzahl der Bytes von `malloc` und ordnet das Ergebnis (dh die Startadresse des gerade mit `malloc` erstellten `malloc`) einem Zeiger mit dem Namen `p` .

Es ist `sizeof` , die Größe des zu verwendenden Speichers mithilfe von `sizeof` zu berechnen, da das Ergebnis von `sizeof` die Implementierung definiert wird (mit Ausnahme der *Zeichentypen* `char` , `signed char` und `unsigned char` , für die `sizeof` so definiert ist, dass sie immer `1` ergibt.

Da `malloc` die Anforderung möglicherweise nicht bedienen kann, gibt es möglicherweise einen Nullzeiger zurück. Es ist wichtig, dies zu überprüfen, um spätere Versuche zur Dereferenzierung des Nullzeigers zu verhindern.

Der mit `malloc()` dynamisch zugewiesene `malloc()` kann mit `realloc()` verkleinert oder mit `free()` freigegeben werden, wenn er nicht mehr benötigt wird.

Alternativ deklarieren Sie das `int array[10];` würde die gleiche Menge an Speicher zuweisen. Wenn sie jedoch innerhalb einer Funktion ohne das Schlüsselwort `static` deklariert wird, ist sie nur in der Funktion, in der sie deklariert ist, und in den aufgerufenen Funktionen verwendbar (da das Array auf dem Stack zugewiesen wird und der Speicherplatz zur Wiederverwendung freigegeben wird die Funktion kehrt zurück). Wenn sie alternativ mit `static` innerhalb einer Funktion definiert ist oder außerhalb einer Funktion definiert ist, dann ist ihre Lebensdauer die Lebensdauer des Programms. Zeiger können auch von einer Funktion zurückgegeben werden, eine Funktion in C kann jedoch kein Array zurückgeben.

Nullspeicher

Der von `malloc` zurückgegebene `malloc` kann nicht mit einem angemessenen Wert initialisiert werden. Es sollte darauf geachtet werden, den Speicher mit `memset` auf Null zu `memset` oder einen geeigneten Wert sofort in ihn zu kopieren. Alternativ gibt `calloc` einen Block der gewünschten Größe zurück, in dem alle Bits auf `0` initialisiert werden. Dies muss nicht der Darstellung der Gleitkommazahl Null oder einer Nullzeiger-Konstante entsprechen.

```
int *p = calloc(10, sizeof *p);
if (p == NULL)
{
    perror("calloc() failed");
    return -1;
}
```

Ein Hinweis zu `calloc` *:* Die meisten (häufig verwendeten) Implementierungen optimieren die Leistung von `calloc()` , sodass es schneller ist als das Aufrufen von `malloc()` und dann `memset()` , obwohl der Nettoeffekt identisch ist.

Ausgerichteter Speicher

C11

Mit C11 wurde eine neue Funktion `aligned_alloc()` die bei der angegebenen Ausrichtung Speicherplatz `aligned_alloc()` . Es kann verwendet werden, wenn der zuzuweisende Speicher an bestimmten Grenzen ausgerichtet werden muss, die von `malloc()` oder `calloc()` nicht erfüllt werden können. `malloc()` und `calloc()` weisen Speicher zu, der für *jeden* Objekttyp geeignet ausgerichtet ist (dh, die Ausrichtung ist `alignof(max_align_t)`). Mit `aligned_alloc()` können jedoch größere Ausrichtungen angefordert werden.

```
/* Allocates 1024 bytes with 256 bytes alignment. */
char *ptr = aligned_alloc(256, 1024);
if (ptr) {
    perror("aligned_alloc()");
    return -1;
}
free(ptr);
```

Der C11-Standard enthält zwei Einschränkungen: 1) Die angeforderte *Größe* (zweites Argument) muss ein ganzzahliges Vielfaches der *Ausrichtung sein* (erstes Argument) und 2) der Wert der *Ausrichtung* sollte eine gültige Ausrichtung sein, die von der Implementierung unterstützt wird. Wenn eine der beiden Bedingungen nicht erfüllt wird, führt dies zu undefiniertem Verhalten .

Speicher neu zuordnen

Möglicherweise müssen Sie den Zeigerspeicherplatz vergrößern oder verkleinern, nachdem Sie ihm Speicher zugewiesen haben. Die Funktion `void *realloc(void *ptr, size_t size)` hebt das alte Objekt auf, auf das `ptr` und gibt einen Zeiger auf ein Objekt mit der durch size angegebenen `size` . `ptr` ist der Zeiger auf einen Speicherblock, der zuvor mit `malloc` , `calloc` oder `realloc` (oder einem Nullzeiger) zugewiesen wurde, um neu zuzuordnen. Der maximal mögliche Inhalt des ursprünglichen Speichers bleibt erhalten. Wenn die neue Größe größer ist, wird zusätzlicher Speicher, der über die alte Größe hinausgeht, nicht initialisiert. Wenn die neue Größe kürzer ist, geht der Inhalt des geschrumpften Teils verloren. Wenn `ptr` NULL ist, wird ein neuer Block zugewiesen und ein Zeiger darauf wird von der Funktion zurückgegeben.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *p = malloc(10 * sizeof *p);
    if (NULL == p)
    {
        perror("malloc() failed");
        return EXIT_FAILURE;
    }
```

```
    p[0] = 42;
    p[9] = 15;

    /* Reallocate array to a larger size, storing the result into a
     * temporary pointer in case realloc() fails. */
    {
        int *temporary = realloc(p, 1000000 * sizeof *temporary);

        /* realloc() failed, the original allocation was not free'd yet. */
        if (NULL == temporary)
        {
            perror("realloc() failed");
            free(p); /* Clean up. */
            return EXIT_FAILURE;
        }

        p = temporary;
    }

    /* From here on, array can be used with the new size it was
     * realloc'ed to, until it is free'd. */

    /* The values of p[0] to p[9] are preserved, so this will print:
       42 15
    */
    printf("%d %d\n", p[0], p[9]);

    free(p);

    return EXIT_SUCCESS;
}
```

Das neu zugewiesene Objekt kann dieselbe Adresse wie `*p` haben oder nicht. Daher ist es wichtig, den Rückgabewert von `realloc` zu erfassen, der die neue Adresse enthält, wenn der Aufruf erfolgreich ist.

Stellen Sie sicher, dass Sie den Rückgabewert von `realloc` einem `temporary` statt dem ursprünglichen `p` zuweisen. `realloc` gibt bei einem Fehler den `realloc` null zurück, wodurch der Zeiger überschrieben wird. Dies würde Ihre Daten verlieren und einen Speicherverlust verursachen.

Mehrdimensionale Arrays mit variabler Größe

C99

Seit C99 besitzt C Arrays mit variabler Länge (VLA), die Arrays mit Grenzen bilden, die nur zur Initialisierungszeit bekannt sind. Während Sie darauf achten müssen, nicht zu große VLA zuzuordnen (sie könnten Ihren Stack zerstören), ist die Verwendung von *Zeigern auf VLA* und deren Verwendung in `sizeof` Ausdrücken in Ordnung.

```
double sumAll(size_t n, size_t m, double A[n][m]) {
    double ret = 0.0;
    for (size_t i = 0; i < n; ++i)
       for (size_t j = 0; j < m; ++j)
          ret += A[i][j]
```

```
    return ret;
}

int main(int argc, char *argv[argc+1]) {
   size_t n = argc*10;
   size_t m = argc*8;
   double (*matrix)[m] = malloc(sizeof(double[n][m]));
   // initialize matrix somehow
   double res = sumAll(n, m, matrix);
   printf("result is %g\n", res);
   free(matrix);
}
```

Hier ist die `matrix` ein Zeiger auf Elemente vom Typ `double[m]` , und der Ausdruck `sizeof` mit `double[n][m]` stellt sicher, dass in ihm Platz für `n` solche Elemente ist.

Dieser gesamte Speicherplatz wird zusammenhängend zugewiesen und kann daher durch einen einzigen Anruf für `free` freigegeben werden.

Das Vorhandensein von VLA in der Sprache beeinflusst auch die möglichen Deklarationen von Arrays und Zeigern in Funktionsheadern. In den Array-Parametern `[]` ist jetzt ein allgemeiner ganzzahliger Ausdruck zulässig. Für beide Funktionen verwenden die Ausdrücke in `[]` Parameter, die zuvor in der Parameterliste deklariert wurden. Für `sumAll` sind dies die Längen, die der Benutzercode für die Matrix erwartet. Wie bei allen Array-Funktionsparametern in C wird die innerste Dimension in einen Zeigertyp umgeschrieben. Dies entspricht der Deklaration

```
  double sumAll(size_t n, size_t m, double (*A)[m]);
```

Das heißt, `n` ist nicht wirklich Teil der Funktionsschnittstelle, aber die Informationen können für die Dokumentation nützlich sein und könnten auch von Compilern zur Überprüfung von Grenzen verwendet werden, um vor einem Zugriff außerhalb der Grenzen zu warnen.

Ebenso ist für `main` der Ausdruck `argc+1` die minimale Länge, die der C-Standard für das Argument `argv` vorschreibt.

Beachten Sie, dass die Unterstützung für VLA in C11 offiziell optional ist, aber wir kennen keinen Compiler, der C11 implementiert und für den sie nicht verfügbar ist. Sie können mit dem Makro `__STDC_NO_VLA__` wenn Sie müssen.

Realloc (Ptr, 0) ist nicht gleich "Free" (Ptr)

`realloc` ist *konzeptionell gleichbedeutend* mit `malloc + memcpy + free` auf dem anderen Zeiger.

Wenn der angeforderte Speicherplatz null ist, ist das Verhalten von `realloc` implementierungsdefiniert. Dies ist ähnlich für alle Speicherzuordnungsfunktionen , die eine Empfangs `size` Parameter Wert `0` . Solche Funktionen können zwar einen Nicht-Null-Zeiger zurückgeben, dieser darf jedoch niemals dereferenziert werden.

Daher ist `realloc(ptr,0)` nicht mit `free(ptr)` gleichwertig. Es kann

- Seien Sie eine "faule" Implementierung und geben Sie einfach `ptr`

- `free(ptr)` , ordnen Sie ein Dummy-Element zu und geben Sie dieses zurück
- `free(ptr)` und `0`
- Gib einfach `0` für einen Fehler zurück und tue nichts anderes.

So sind insbesondere die beiden letztgenannten Fälle durch den Anwendungscode nicht zu unterscheiden.

Dies bedeutet, dass `realloc(ptr,0)` den Speicher möglicherweise nicht wirklich freigibt / freigibt. Daher sollte er niemals als Ersatz für `free` .

Benutzerdefinierte Speicherverwaltung

`malloc()` ruft häufig zugrunde liegende Betriebssystemfunktionen auf, um Speicherseiten zu erhalten. Die Funktion ist jedoch nichts Besonderes, und sie kann in Straight C implementiert werden, indem ein großes statisches Array deklariert und von diesem zugewiesen wird (es gibt eine leichte Schwierigkeit, die korrekte Ausrichtung zu gewährleisten. In der Praxis ist das Anpassen an 8 Byte fast immer ausreichend).

Um ein einfaches Schema zu implementieren, wird ein Steuerblock in dem Speicherbereich unmittelbar vor dem Zeiger gespeichert, der vom Anruf zurückgegeben werden soll. Dies bedeutet, dass `free()` implementiert werden kann, indem von dem zurückgegebenen Zeiger subtrahiert wird und die Steuerinformationen abgelesen werden. Hierbei handelt es sich normalerweise um die Blockgröße plus einige Informationen, die es ermöglichen, dass sie in die freie Liste zurückgegeben werden - eine verknüpfte Liste nicht zugeordneter Blöcke.

Wenn der Benutzer eine Zuweisung anfordert, wird die freie Liste durchsucht, bis ein Block gefunden wird, der mit dem angeforderten Betrag identisch oder größer ist. Anschließend wird er gegebenenfalls aufgeteilt. Dies kann zu einer Fragmentierung des Speichers führen, wenn der Benutzer ständig viele Zuordnungen und Freigaben von unvorhersehbarer Größe vornimmt und in unvorhersehbaren Intervallen (nicht alle realen Programme verhalten sich so, das einfache Schema ist für kleine Programme oft ausreichend).

```
/* typical control block */
struct block
{
   size_t size;         /* size of block */
   struct block *next;  /* next block in free list */
   struct block *prev;  /* back pointer to previous block in memory */
   void *padding;       /* need 16 bytes to make multiple of 8 */
}

static struct block arena[10000]; /* allocate from here */
static struct block *firstfree;
```

Viele Programme erfordern eine große Anzahl von Zuweisungen von kleinen Objekten gleicher Größe. Dies ist sehr einfach zu implementieren. Verwenden Sie einfach einen Block mit einem nächsten Zeiger. Wenn also ein Block von 32 Bytes erforderlich ist:

```
union block
{
```

```
    union block * next;
    unsigned char payload[32];
}

static union block arena[100];
static union block * head;
void init(void)
{
    int i;
    for (i = 0; i < 100 - 1; i++)
        arena[i].next = &arena[i + 1];
    arena[i].next = 0; /* last one, null */
    head = &block[0];
}

void *block_alloc()
{
    void *answer = head;
    if (answer)
        head = head->next;
    return answer;
}

void block_free(void *ptr)
{
    union block *block = ptr;
    block->next = head;
    head - block;
}
```

Dieses Schema ist extrem schnell und effizient und kann mit einem gewissen Verlust an Klarheit generisch gemacht werden.

Allocationa: Speicher auf Stack zuweisen

`alloca` : `alloca` wird hier nur der Vollständigkeit halber erwähnt. Es ist vollständig nicht portabel (nicht durch eine der gängigen Standards abgedeckt) und verfügt über eine Reihe potenziell gefährlicher Funktionen, die es für Unbewusste unsicher machen. Moderner C-Code sollte ihn durch *Arrays* mit *variabler Länge* (VLA) ersetzen.

Manuelle Seite

```
#include <alloca.h>
// glibc version of stdlib.h include alloca.h by default

void foo(int size) {
    char *data = alloca(size);
    /*
      function body;
    */
    // data is automatically freed
}
```

Weisen Sie dem Stack-Frame des Aufrufers Speicher zu. Der vom zurückgegebenen Zeiger referenzierte Speicherplatz wird automatisch freigegeben , wenn die Caller-Funktion abgeschlossen ist.

Während diese Funktion für die automatische Speicherverwaltung geeignet ist, sollten Sie beachten, dass das Anfordern einer großen Zuweisung einen Stapelüberlauf verursachen kann und dass Sie den mit `alloca` zugewiesenen Speicher nicht `free` `alloca` (was zu mehr `alloca` beim Stapelüberlauf führen kann).

Aus diesem Grund wird die Verwendung von `alloca` innerhalb einer Schleife oder eine rekursive Funktion nicht empfohlen.

Und da der Speicher bei der Funktionsrückgabe `free` , können Sie den Zeiger nicht als Funktionsergebnis zurückgeben (das Verhalten wäre undefiniert).

Zusammenfassung

- Call identisch mit `malloc`
- automatisch bei Funktionsrückkehr frei
- nicht kompatibel mit `free` `realloc` Funktionen (undefiniertes Verhalten)
- Zeiger kann nicht als Funktionsergebnis zurückgegeben werden (undefiniertes Verhalten)
- Zuordnungsgröße begrenzt durch Stack-Speicherplatz, der (bei den meisten Computern) viel kleiner ist als der für `malloc()` verfügbare Heap- `malloc()`
- Vermeiden Sie die Verwendung `alloca()` und VLAs (Arrays mit variabler Länge) in einer einzigen Funktion
- `alloca()` ist nicht so portabel wie `malloc()` et al

Empfehlung

- Verwenden `alloca()` in neuem Code nicht `alloca()`

C99

Moderne Alternative.

```
void foo(int size) {
    char data[size];
    /*
      function body;
    */
    // data is automatically freed
}
```

Dies funktioniert an den `alloca()` denen `alloca()` , und funktioniert an Stellen, an denen die `alloca()` nicht innerhalb von Schleifen `alloca()` . Es wird davon ausgegangen, dass entweder eine C99-Implementierung oder eine C11-Implementierung `__STDC_NO_VLA__` die `__STDC_NO_VLA__` nicht definiert.

Speicherverwaltung online lesen: https://riptutorial.com/de/c/topic/4726/speicherverwaltung

Kapitel 45: Sprunganweisungen

Syntax

- Rückgabewert; / * Kehrt von der aktuellen Funktion zurück. val kann ein Wert eines beliebigen Typs sein, der in den Rückgabetyp der Funktion konvertiert wird. * /
- Rückkehr; / * Kehrt von der aktuellen Void-Funktion zurück. * /
- brechen; / * Springt unbedingt über das Ende einer Iterationsanweisung (Schleife) oder über die innerste switch-Anweisung. * /
- fortsetzen; / * Springt bedingungslos an den Anfang einer Iterationsanweisung (Schleife). * /
- gehe zu LBL; / * Springt zur Bezeichnung von LBL. * /
- LBL: *Anweisung* / * eine beliebige Anweisung in derselben Funktion. * /

Bemerkungen

Dies sind die Sprünge, die über Schlüsselwörter in C integriert werden.

C hat auch ein weiteres Sprungkonstrukt, einen *langen Sprung* , der mit dem Datentyp `jmp_buf` und C-Bibliotheksaufrufen, `setjmp` und `longjmp` .

Siehe auch

Iterationsanweisungen / -schleifen: für, während, währenddessen

Examples

Mit goto aus verschachtelten Schleifen springen

Um aus verschachtelten Schleifen zu springen, müsste normalerweise eine boolesche Variable mit einer Prüfung dieser Variablen in den Schleifen verwendet werden. Angenommen, wir würden über `i` und `j` iterieren, könnte es so aussehen

```
size_t i,j;
for (i = 0; i < myValue && !breakout_condition; ++i) {
    for (j = 0; j < mySecondValue && !breakout_condition; ++j) {
        ... /* Do something, maybe modifying breakout_condition */
            /* When breakout_condition == true the loops end */
     }
}
```

Die C-Sprache bietet jedoch die `goto` Klausel, die in diesem Fall nützlich sein kann. Durch die Verwendung eines Labels, das nach den Loops deklariert wurde, können wir die Loops leicht ausbrechen.

```
size_t i,j;
```

```
for (i = 0; i < myValue; ++i) {
    for (j = 0; j < mySecondValue; ++j) {
        ...
        if(breakout_condition)
          goto final;
    }
}
final:
```

Wenn jedoch dieses Bedürfnis aufkommt, kann eine `return` stattdessen besser verwendet werden. Dieses Konstrukt wird auch in der strukturellen Programmiertheorie als "unstrukturiert" betrachtet.

Eine andere Situation, in der `goto` nützlich sein könnte, ist das Wechseln zu einem Fehlerbehandler:

```
ptr = malloc(N *  x);
if(!ptr)
  goto out_of_memory;

/* normal processing */
free(ptr);
return SUCCESS;

out_of_memory:
 free(ptr); /* harmless, and necessary if we have further errors */
 return FAILURE;
```

Die Verwendung von `goto` hält den Fehlerfluss vom normalen Programmsteuerungsfluss getrennt. Es wird jedoch auch im technischen Sinne als "unstrukturiert" betrachtet.

Verwenden Sie Zurück

Rückgabe eines Wertes

Ein häufig verwendeter Fall: Rückkehr von `main()`

```
#include <stdlib.h> /* for EXIT_xxx macros */

int main(int argc, char ** argv)
{
  if (2 < argc)
  {
    return EXIT_FAILURE; /* The code expects one argument:
                            leave immediately skipping the rest of the function's code */
  }

  /* Do stuff. */

  return EXIT_SUCCESS;
}
```

Zusätzliche Bemerkungen:

1. Für eine Funktion mit einem Rückgabetyp als `void` (ohne `void *` oder verwandte Typen)

sollte die `return` keinen zugeordneten Ausdruck haben. dh die einzige erlaubte return-Anweisung wäre `return;` .

2. Bei einer Funktion mit einem nicht `void` Rückgabetyp darf die `return` Anweisung nicht ohne einen Ausdruck erscheinen.

3. Für `main()` (und nur für `main()`) ist keine *explizite* `return` Anweisung (in C99 oder höher) erforderlich. Wenn die Ausführung den Abbruch `}` , wird ein impliziter Wert von `0` zurückgegeben. Einige Leute denken, diese `return` wegzulassen `return` ist eine schlechte Praxis. andere schlagen aktiv vor, es wegzulassen.

Nichts zurückgeben

Rückkehr von einer `void` Funktion

```
void log(const char * message_to_log)
{
  if (NULL == message_to_log)
  {
    return; /* Nothing to log, go home NOW, skip the logging. */
  }

  fprintf(stderr, "%s:%d %s\n", __FILE__, _LINE__, message_to_log);

  return; /* Optional, as this function does not return a value. */
}
```

Verwenden Sie break und fahren Sie fort

Lesen Sie `continue` Lesen bei ungültiger Eingabe oder `break` auf Benutzeranfrage oder Dateiendung sofort `continue` :

```
#include <stdlib.h> /* for EXIT_xxx macros */
#include <stdio.h>  /* for printf() and getchar() */
#include <ctype.h> /* for isdigit() */

void flush_input_stream(FILE * fp);


int main(void)
{
  int sum = 0;
  printf("Enter digits to be summed up or 0 to exit:\n");

  do
  {
    int c = getchar();
    if (EOF == c)
    {
      printf("Read 'end-of-file', exiting!\n");

      break;
    }
```

```
    if ('\n' != c)
    {
      flush_input_stream(stdin);
    }

    if (!isdigit(c))
    {
      printf("%c is not a digit! Start over!\n", c);

      continue;
    }

    if ('0' == c)
    {
      printf("Exit requested.\n");

      break;
    }

    sum += c - '0';

    printf("The current sum is %d.\n", sum);
  } while (1);

  return EXIT_SUCCESS;
}

void flush_input_stream(FILE * fp)
{
  size_t i = 0;
  int c;
  while ((c = fgetc(fp)) != '\n' && c != EOF) /* Pull all until and including the next new-
line. */
  {
    ++i;
  }

  if (0 != i)
  {
    fprintf(stderr, "Flushed %zu characters from input.\n", i);
  }
}
```

Sprunganweisungen online lesen: https://riptutorial.com/de/c/topic/5568/sprunganweisungen

Kapitel 46: Standard Math

Syntax

- #include <math.h>
- Doppelschlag (doppeltes x, doppeltes y);
- float powf (float x, float y);
- langes doppeltes powl (langes doppeltes x, langes doppeltes y);

Bemerkungen

1. Um eine Verknüpfung mit der Mathematikbibliothek `-lm` verwenden Sie `-lm` mit gcc-Flags.
2. Ein tragbares Programm, das nach einem Fehler in einer mathematischen Funktion `errno` , sollte `errno` auf null setzen und den folgenden Aufruf `feclearexcept(FE_ALL_EXCEPT);` bevor Sie eine mathematische Funktion aufrufen. Bei der Rückkehr von der mathematischen Funktion wird, wenn `errno` ungleich Null ist, oder der folgende Aufruf einen ungleich null `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW);` dann trat ein Fehler in der mathematischen Funktion auf. Lesen Sie die Manpage von math_error für weitere Informationen.

Examples

Fließkomma-Rest mit doppelter Genauigkeit: fmod ()

Diese Funktion gibt den Fließkomma-Rest der Division von `x/y` . Der zurückgegebene Wert hat das gleiche Vorzeichen wie x.

```
#include <math.h> /* for fmod() */
#include <stdio.h> /* for printf() */

int main(void)
{
    double x = 10.0;
    double y = 5.1;

    double modulus = fmod(x, y);

    printf("%lf\n", modulus); /* f is the same as lf. */

    return 0;
}
```

Ausgabe:

```
4.90000
```

Wichtig: Verwenden Sie diese Funktion mit Vorsicht, da unerwartete Werte aufgrund der

Verwendung von Gleitkommawerten zurückgegeben werden können.

```
#include <math.h>
#include <stdio.h>

int main(void)
{
    printf("%f\n", fmod(1, 0.1));
    printf("%19.17f\n", fmod(1, 0.1));
    return 0;
}
```

Ausgabe:

```
0.1
0.09999999999999995
```

Gleitkomma-Rest mit einfacher Genauigkeit und langer doppelter Genauigkeit: fmodf (), fmodl ()

C99

Diese Funktionen geben den Fließkomma-Rest der Division von `x/y` . Der zurückgegebene Wert hat das gleiche Vorzeichen wie x.

Mit einfacher Genauigkeit:

```
#include <math.h> /* for fmodf() */
#include <stdio.h> /* for printf() */

int main(void)
{
    float x = 10.0;
    float y = 5.1;

    float modulus = fmodf(x, y);

    printf("%f\n", modulus); /* lf would do as well as modulus gets promoted to double. */
}
```

Ausgabe:

```
4.90000
```

Doppelte Doppelpräzision:

```
#include <math.h> /* for fmodl() */
#include <stdio.h> /* for printf() */

int main(void)
{
    long double x = 10.0;
    long double y = 5.1;
```

```
    long double modulus = fmodl(x, y);

    printf("%Lf\n", modulus); /* Lf is for long double. */
}
```

Ausgabe:

```
4.90000
```

Power-Funktionen - pow (), powf (), powl ()

Der folgende Beispielcode berechnet die Summe der Reihen *1 + 4 (3 + 3 ^ 2 + 3 ^ 3 + 3 ^ 4 + ... + 3 ^ N)* unter Verwendung der pow () - Familie der Standard-Mathematik-Bibliothek.

```
#include <stdio.h>
#include <math.h>
#include <errno.h>
#include <fenv.h>

int main()
{
        double pwr, sum=0;
        int i, n;

        printf("\n1+4(3+3^2+3^3+3^4+...+3^N)=?\nEnter N:");
        scanf("%d",&n);
        if (n<=0) {
                printf("Invalid power N=%d", n);
                return -1;
        }

        for (i=0; i<n+1; i++) {
                errno = 0;
                feclearexcept(FE_ALL_EXCEPT);
                pwr = powl(3,i);
                if (fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW |
                        FE_UNDERFLOW)) {
                        perror("Math Error");
                }
                sum += i ? pwr : 0;
                printf("N= %d\tS= %g\n", i, 1+4*sum);
        }

        return 0;
}
```

Beispielausgabe:

```
1+4(3+3^2+3^3+3^4+...+3^N)=?
Enter N:10
N= 0    S= 1
N= 1    S= 13
N= 2    S= 49
N= 3    S= 157
N= 4    S= 481
```

```
N= 5    S= 1453
N= 6    S= 4369
N= 7    S= 13117
N= 8    S= 39361
N= 9    S= 118093
N= 10    S= 354289
```

Standard Math online lesen: https://riptutorial.com/de/c/topic/3170/standard-math

Kapitel 47: Structs

Einführung

Strukturen bieten die Möglichkeit, eine Gruppe verwandter Variablen verschiedener Typen in einer einzigen Speichereinheit zu gruppieren. Die Struktur als Ganzes kann durch einen einzelnen Namen oder Zeiger referenziert werden. Die Strukturelemente können auch einzeln aufgerufen werden. Strukturen können an Funktionen übergeben und von Funktionen zurückgegeben werden. Sie werden mit dem Schlüsselwort `struct` .

Examples

Einfache Datenstrukturen

Strukturdatentypen sind eine nützliche Methode, um verwandte Daten zu packen und sich wie eine einzelne Variable zu verhalten.

Eine einfache `struct` , die zwei `int` Mitglieder enthält:

```
struct point
{
    int x;
    int y;
};
```

`x` und `y` werden die *Mitglieder* (oder *Felder*) der `point` .

Definieren und Verwenden von Strukturen:

```
struct point p;    // declare p as a point struct
p.x = 5;           // assign p member variables
p.y = 3;
```

Strukturen können bei der Definition initialisiert werden. Das obige ist äquivalent zu:

```
struct point p = {5, 3};
```

Strukturen können auch mit bestimmten Initialisierern initialisiert werden .

Der Zugriff auf Felder erfolgt ebenfalls mit `.` Operator

```
printf("point is (x = %d, y = %d)", p.x, p.y);
```

Typedef-Strukturen

Die Kombination von `typedef` mit `struct` kann den Code klarer machen. Zum Beispiel:

```
typedef struct
{
    int x, y;
} Point;
```

im Gegensatz zu:

```
struct Point
{
    int x, y;
};
```

könnte erklärt werden als:

```
Point point;
```

anstatt:

```
struct Point point;
```

Noch besser ist es, folgendes zu verwenden

```
typedef struct Point Point;

struct Point
{
    int x, y;
};
```

beide mögliche Definitionen von `point` . Eine solche Deklaration ist am bequemsten, wenn Sie zuerst C ++ gelernt haben, wobei Sie das Schlüsselwort `struct` auslassen können, wenn der Name nicht eindeutig ist.

`typedef` Namen für Strukturen können mit anderen Bezeichnern anderer Programmteile in Konflikt stehen. Einige halten dies für einen Nachteil, aber für die meisten Menschen, die über eine `struct` und eine andere Kennung `struct` ist dies ziemlich störend. Notorious ist zB POSIX‘ `stat`

```
int stat(const char *pathname, struct stat *buf);
```

wo sehen Sie eine Funktion `stat` , das ein Argument hat , das ist `struct stat` .

`typedef` -Strukturen ohne Tag-Namen ist immer die gesamte `struct` Deklaration für den Code sichtbar, der sie verwendet. Die gesamte `struct` Deklaration muss dann in einer Headerdatei abgelegt werden.

Erwägen:

```
#include "bar.h"

struct foo
```

```
{
    bar *aBar;
};
```

Bei einer `typedef` d- `struct` , die keinen Tag-Namen hat, muss die Datei `bar.h` immer die gesamte Definition der `bar` . Wenn wir verwenden

```
typedef struct bar bar;
```

In `bar.h` können die Details der `bar` ausgeblendet werden.

Siehe Typedef

Verweise auf structs

Wenn Sie über eine Variable verfügen, die eine `struct` , können Sie mit dem Punktoperator (`.`) Auf ihre Felder zugreifen. Wenn Sie jedoch einen Zeiger auf eine `struct` , funktioniert dies nicht. Sie müssen den Pfeiloperator (`->`) verwenden, um auf die Felder zuzugreifen. Hier ist ein Beispiel einer schrecklich einfachen (manche sagen "schrecklich und einfach") Implementierung eines Stacks, der Zeiger für `struct` und den Pfeiloperator veranschaulicht.

```
#include <stdlib.h>
#include <stdio.h>

/* structs */
struct stack
{
    struct node *top;
    int size;
};

struct node
{
    int data;
    struct node *next;
};

/* function declarations */
int push(int, struct stack*);
int pop(struct stack*);
void destroy(struct stack*);

int main(void)
{
    int result = EXIT_SUCCESS;

    size_t i;

    /* allocate memory for a struct stack and record its pointer */
    struct stack *stack = malloc(sizeof *stack);
    if (NULL == stack)
    {
        perror("malloc() failed");
        return EXIT_FAILURE;
    }
```

```
    /* initialize stack */
    stack->top = NULL;
    stack->size = 0;

    /* push 10 ints */
    {
        int data = 0;
        for(i = 0; i < 10; i++)
        {
            printf("Pushing: %d\n", data);
            if (-1 == push(data, stack))
            {
                perror("push() failed");
                result = EXIT_FAILURE;
                break;
            }

            ++data;
        }
    }

    if (EXIT_SUCCESS == result)
    {
        /* pop 5 ints */
        for(i = 0; i < 5; i++)
        {
            printf("Popped: %i\n", pop(stack));
        }
    }

    /* destroy stack */
    destroy(stack);

    return result;
}

/* Push a value onto the stack. */
/* Returns 0 on success and -1 on failure. */
int push(int data, struct stack *stack)
{
    int result = 0;

    /* allocate memory for new node */
    struct node *new_node = malloc(sizeof *new_node);
    if (NULL == new_node)
    {
        result = -1;
    }
    else
    {
        new_node->data = data;
        new_node->next = stack->top;
        stack->top = new_node;
        stack->size++;
    }

    return result;
}

/* Pop a value off of the stack. */
```

```
/* Returns the value popped off the stack */
int pop(struct stack *stack)
{
    struct node *top = stack->top;
    int data = top->data;
    stack->top = top->next;
    stack->size--;
    free(top);
    return data;
}

/* destroy the stack */
void destroy(struct stack *stack)
{
    /* free all pointers */
    while(stack->top != NULL)
    {
        pop(stack);
    }
}
```

Flexible Array-Mitglieder

C99

Typenerklärung

Eine Struktur *mit mindestens einem Member* kann zusätzlich ein einzelnes Array-Member mit nicht näher bezeichneter Länge am Ende der Struktur enthalten. Dies wird als flexibles Arraymitglied bezeichnet:

```
struct ex1
{
    size_t foo;
    int flex[];
};

struct ex2_header
{
    int foo;
    char bar;
};

struct ex2
{
    struct ex2_header hdr;
    int flex[];
};

/* Merged ex2_header and ex2 structures. */
struct ex3
{
    int foo;
    char bar;
    int flex[];
};
```

Auswirkungen auf Größe und Polsterung

Ein flexibles Array-Member wird bei der Berechnung der Größe einer Struktur als ohne Größe behandelt. Es kann jedoch immer noch eine Auffüllung zwischen diesem Member und dem vorherigen Member der Struktur vorhanden sein:

```
/* Prints "8,8" on my machine, so there is no padding. */
printf("%zu,%zu\n", sizeof(size_t), sizeof(struct ex1));

/* Also prints "8,8" on my machine, so there is no padding in the ex2 structure itself. */
printf("%zu,%zu\n", sizeof(struct ex2_header), sizeof(struct ex2));

/* Prints "5,8" on my machine, so there are 3 bytes of padding. */
printf("%zu,%zu\n", sizeof(int) + sizeof(char), sizeof(struct ex3));
```

Das flexible Array- `sizeof` hat einen unvollständigen Array-Typ, daher kann seine Größe nicht mit `sizeof` berechnet werden.

Verwendungszweck

Sie können ein Objekt mit einem Strukturtyp deklarieren und initialisieren, der ein flexibles Arraymitglied enthält. Sie dürfen jedoch nicht versuchen, das flexible Arraymitglied zu initialisieren, da es so behandelt wird, als wäre es nicht vorhanden. Es ist verboten, dies zu versuchen, und es werden Kompilierungsfehler ausgegeben.

Ebenso sollten Sie nicht versuchen, einem Element eines flexiblen Arraymitglieds einen Wert zuzuweisen, wenn Sie eine Struktur auf diese Weise deklarieren, da am Ende der Struktur möglicherweise nicht genügend Auffüllung vorhanden ist, um alle vom flexiblen Arraymitglied benötigten Objekte zuzulassen. Der Compiler hindert Sie jedoch nicht unbedingt daran, dies kann zu undefiniertem Verhalten führen.

```
/* invalid: cannot initialize flexible array member */
struct ex1 e1 = {1, {2, 3}};
/* invalid: hdr={foo=1, bar=2} OK, but cannot initialize flexible array member */
struct ex2 e2 = {{1, 2}, {3}};
/* valid: initialize foo=1, bar=2 members */
struct ex3 e3 = {1, 2};

e1.flex[0] = 3; /* undefined behavior, in my case */
e3.flex[0] = 2; /* undefined behavior again */
e2.flex[0] = e3.flex[0]; /* undefined behavior */
```

Sie können stattdessen `malloc` , `calloc` oder `realloc` , um die Struktur mit zusätzlichem Speicher zuzuordnen und später `calloc` , `calloc` Sie das flexible Array- `calloc` nach Belieben verwenden können:

```
/* valid: allocate an object of structure type `ex1` along with an array of 2 ints */
struct ex1 *pe1 = malloc(sizeof(*pe1) + 2 * sizeof(pe1->flex[0]));

/* valid: allocate an object of structure type ex2 along with an array of 4 ints */
```

```
struct ex2 *pe2 = malloc(sizeof(struct ex2) + sizeof(int[4]));

/* valid: allocate 5 structure type ex3 objects along with an array of 3 ints per object */
struct ex3 *pe3 = malloc(5 * (sizeof(*pe3) + sizeof(int[3])));

pe1->flex[0] = 3; /* valid */
pe3[0]->flex[0] = pe1->flex[0]; /* valid */
```

C99

Der "struct hack"

Flexible Array-Mitglieder waren vor C99 nicht vorhanden und werden als Fehler behandelt. Eine gängige Problemumgehung besteht darin, ein Array der Länge 1 zu deklarieren, eine Technik, die als "struct hack" bezeichnet wird:

```
struct ex1
{
    size_t foo;
    int flex[1];
};
```

Dies hat jedoch Auswirkungen auf die Größe der Struktur, im Gegensatz zu einem echten flexiblen Arraymitglied:

```
/* Prints "8,4,16" on my machine, signifying that there are 4 bytes of padding. */
printf("%d,%d,%d\n", (int)sizeof(size_t), (int)sizeof(int[1]), (int)sizeof(struct ex1));
```

Um die Verwendung `flex` als ein flexibles Element - Array, würde man es mit zuteilen `malloc` , wie oben gezeigt, mit der Ausnahme , dass `sizeof(*pe1)` (oder das Äquivalent `sizeof(struct ex1)`) würde ersetzt werden durch `offsetof(struct ex1, flex)` oder die längere `sizeof(*pe1)-sizeof(pe1->flex)` Expressionsgröße von `sizeof(*pe1)-sizeof(pe1->flex)` . Alternativ können Sie 1 von der gewünschten Länge des "flexiblen" Arrays subtrahieren, da es bereits in der Strukturgröße enthalten ist, sofern die gewünschte Länge größer als 0 ist. Die gleiche Logik kann auf die anderen Verwendungsbeispiele angewendet werden.

Kompatibilität

Wenn Kompatibilität mit Compilern gewünscht wird, die keine flexiblen Array-Mitglieder unterstützen, können Sie ein Makro verwenden, das wie `FLEXMEMB_SIZE` definiert ist: `FLEXMEMB_SIZE` :

```
#if __STDC_VERSION__ < 199901L
#define FLEXMEMB_SIZE 1
#else
#define FLEXMEMB_SIZE /* nothing */
#endif

struct ex1
{
    size_t foo;
    int flex[FLEXMEMB_SIZE];
```

```
};
```

Beim `offsetof(struct ex1, flex)` Objekten sollten Sie das `offsetof(struct ex1, flex)` verwenden, um auf die Strukturgröße zu verweisen (mit Ausnahme des flexiblen Array-Mitglieds), da dies der einzige Ausdruck ist, der zwischen Compilern, die flexible Array-Mitglieder unterstützen, und den Compilern, die dies tun, konsistent bleibt nicht:

```
struct ex1 *pe10 = malloc(offsetof(struct ex1, flex) + n * sizeof(pe10->flex[0]));
```

Die Alternative ist, den Präprozessor zu verwenden, um 1 von der angegebenen Länge bedingt abzuziehen. Aufgrund des erhöhten Potenzials für Inkonsistenz und allgemeines menschliches Versagen in dieser Form habe ich die Logik in eine separate Funktion verschoben:

```
struct ex1 *ex1_alloc(size_t n)
{
    struct ex1 tmp;
#if __STDC_VERSION__ < 199901L
    if (n != 0)
        n--;
#endif
    return malloc(sizeof(tmp) + n * sizeof(tmp.flex[0]));
}
...

/* allocate an ex1 object with "flex" array of length 3 */
struct ex1 *pe1 = ex1_alloc(3);
```

Übergabe von Strukturen an Funktionen

In C werden alle Argumente nach Wert an Funktionen übergeben, einschließlich Strukturen. Für kleine Strukturen ist dies eine gute Sache, da dies bedeutet, dass kein Zugriff auf die Daten durch einen Zeiger entsteht. Es macht es jedoch auch sehr leicht, versehentlich eine große Struktur zu übergeben, was zu einer schlechten Leistung führt, insbesondere wenn der Programmierer an andere Sprachen gewöhnt ist, in denen Argumente als Referenz übergeben werden.

```
struct coordinates
{
    int x;
    int y;
    int z;
};

// Passing and returning a small struct by value, very fast
struct coordinates move(struct coordinates position, struct coordinates movement)
{
    position.x += movement.x;
    position.y += movement.y;
    position.z += movement.z;
    return position;
}

// A very big struct
struct lotsOfData
```

```
{
    int param1;
    char param2[80000];
};

// Passing and returning a large struct by value, very slow!
// Given the large size of the struct this could even cause stack overflow
struct lotsOfData doubleParam1(struct lotsOfData value)
{
    value.param1 *= 2;
    return value;
}

// Passing the large struct by pointer instead, fairly fast
void doubleParam1ByPtr(struct lotsOfData *value)
{
    value->param1 *= 2;
}
```

Objektbasierte Programmierung mit Strukturen

Strukturen können verwendet werden, um Code objektorientiert zu implementieren. Eine Struktur ähnelt einer Klasse, es fehlen jedoch die Funktionen, die normalerweise auch Teil einer Klasse sind. Wir können diese als Funktionszeiger-Member-Variablen hinzufügen. Um bei unserem Koordinatenbeispiel zu bleiben:

```
/* coordinates.h */

typedef struct coordinate_s
{
    /* Pointers to method functions */
    void (*setx)(coordinate *this, int x);
    void (*sety)(coordinate *this, int y);
    void (*print)(coordinate *this);
    /* Data */
    int x;
    int y;
} coordinate;

/* Constructor */
coordinate *coordinate_create(void);
/* Destructor */
void coordinate_destroy(coordinate *this);
```

Und nun die implementierende C-Datei:

```
/* coordinates.c */

#include "coordinates.h"
#include <stdio.h>
#include <stdlib.h>

/* Constructor */
coordinate *coordinate_create(void)
{
    coordinate *c = malloc(sizeof(*c));
    if (c != 0)
```

```
    {
        c->setx = &coordinate_setx;
        c->sety = &coordinate_sety;
        c->print = &coordinate_print;
        c->x = 0;
        c->y = 0;
    }
    return c;
}

/* Destructor */
void coordinate_destroy(coordinate *this)
{
    if (this != NULL)
    {
        free(this);
    }
}

/* Methods */
static void coordinate_setx(coordinate *this, int x)
{
    if (this != NULL)
    {
        this->x = x;
    }
}

static void coordinate_sety(coordinate *this, int y)
{
    if (this != NULL)
    {
        this->y = y;
    }
}

static void coordinate_print(coordinate *this)
{
    if (this != NULL)
    {
        printf("Coordinate: (%i, %i)\n", this->x, this->y);
    }
    else
    {
        printf("NULL pointer exception!\n");
    }
}
```

Ein Beispiel für die Verwendung unserer Koordinatenklasse wäre:

```
/* main.c */

#include "coordinates.h"
#include <stddef.h>

int main(void)
{
    /* Create and initialize pointers to coordinate objects */
    coordinate *c1 = coordinate_create();
    coordinate *c2 = coordinate_create();
```

```
    /* Now we can use our objects using our methods and passing the object as parameter */
    c1->setx(c1, 1);
    c1->sety(c1, 2);

    c2->setx(c2, 3);
    c2->sety(c2, 4);

    c1->print(c1);
    c2->print(c2);

    /* After using our objects we destroy them using our "destructor" function */
    coordinate_destroy(c1);
    c1 = NULL;
    coordinate_destroy(c2);
    c2 = NULL;

    return 0;
}
```

Structs online lesen: https://riptutorial.com/de/c/topic/1119/structs

Kapitel 48: Strukturpolsterung und Verpackung

Einführung

Standardmäßig legen C-Compiler Strukturen fest, so dass auf jedes Mitglied schnell zugegriffen werden kann, ohne dass Strafen für nicht ausgerichteten Zugriff, ein Problem mit RISC-Computern wie DEC Alpha und einigen ARM-CPUs entstehen.

Abhängig von der CPU-Architektur und dem Compiler benötigt eine Struktur möglicherweise mehr Speicherplatz als die Summe der Größen ihrer Komponenten. Der Compiler kann Padding zwischen Mitgliedern oder am Ende der Struktur hinzufügen, jedoch nicht am Anfang.

Beim Packen wird die Standardauffüllung überschrieben.

Bemerkungen

Eric Raymond hat einen Artikel über The Lost Art of C Structure Packing, der nützlich ist, um zu lesen.

Examples

Verpackungsstrukturen

Standardmäßig werden Strukturen in C aufgefüllt. Wenn Sie dieses Verhalten vermeiden möchten, müssen Sie es explizit anfordern. Unter GCC ist es `__attribute__((__packed__))` . Betrachten Sie dieses Beispiel auf einem 64-Bit-Computer:

```
struct foo {
    char *p;  /* 8 bytes */
    char c;   /* 1 byte  */
    long x;   /* 8 bytes */
};
```

Die Struktur wird automatisch mit einer `8-byte` Ausrichtung aufgefüllt und sieht folgendermaßen aus:

```
struct foo {
    char *p;     /* 8 bytes */
    char c;      /* 1 byte  */

    char pad[7]; /* 7 bytes added by compiler */

    long x;      /* 8 bytes */
};
```

`sizeof(struct foo)` gibt uns also `24` statt `17` . Dies geschah aufgrund eines 64-Bit-Compilers, der in jedem Schritt aus 8 Bytes aus dem Lese- / Schreibspeicher aus dem / in den Speicher ablegte, und offensichtlich, wenn versucht wurde, `char c;` zu schreiben `char c;` Ein Byte im Speicher, ein komplettes 8 Byte (dh Wort) wird abgerufen und verbraucht nur das erste Byte, und die sieben aufeinanderfolgenden Bytes bleiben leer und können für keine Lese- und Schreiboperation zum Strukturauffüllen verwendet werden.

Strukturverpackung

Wenn Sie das Attribut `packed` hinzufügen, fügt der Compiler keine Auffüllung hinzu:

```
struct __attribute__((__packed__)) foo {
    char *p;   /* 8 bytes */
    char c;    /* 1 byte  */
    long x;    /* 8 bytes */
};
```

`sizeof(struct foo)` gibt jetzt `17` .

Im Allgemeinen werden gepackte Strukturen verwendet:

- Um platz zu sparen.
- Formatieren einer Datenstruktur für die Übertragung über ein Netzwerk, ohne von der jeweiligen Architekturausrichtung jedes Knotens des Netzwerks abhängig zu sein.

Es muss berücksichtigt werden, dass einige Prozessoren wie der ARM Cortex-M0 keinen unausgerichteten Speicherzugriff zulassen. In solchen Fällen kann das Packen von Strukturen zu *undefiniertem Verhalten führen* und die CPU zum Absturz bringen.

Strukturpolsterung

Angenommen, diese `struct` wird mit einem 32-Bit-Compiler definiert und kompiliert:

```
struct test_32 {
    int a;      // 4 byte
    short b;    // 2 byte
    int c;      // 4 byte
} str_32;
```

Wir können erwarten, dass diese `struct` nur 10 Bytes Speicher belegt, aber wenn Sie `sizeof(str_32)` , werden 12 Bytes verwendet.

Dies geschah, weil der Compiler Variablen für den schnellen Zugriff ausrichtete. Ein übliches Muster ist, dass, wenn der Basistyp N Bytes belegt (wobei N eine Potenz von 2 ist, wie 1, 2, 4, 8, 16 - und selten größer), die Variable an einer N-Byte-Grenze ausgerichtet sein sollte (ein Vielfaches von N Bytes).

Für die Struktur, die mit `sizeof(int) == 4` und `sizeof(short) == 2` , ist ein allgemeines Layout:

- `int a;` bei Offset 0 gespeichert; Größe 4.
- `short b;` bei Offset 4 gespeichert; Größe 2.
- unbenannte Auffüllung am Offset 6; Größe 2.
- `int c;` bei Offset 8 gespeichert; Größe 4.

`struct test_32` belegt somit 12 Byte Speicher. In diesem Beispiel gibt es keine nachstehende Auffüllung.

Der Compiler stellt sicher, dass alle Variablen von `struct test_32` beginnend mit einer 4-Byte-Grenze gespeichert werden, sodass die Member innerhalb der Struktur für einen schnellen Zugriff ordnungsgemäß ausgerichtet sind. Speicherzuordnungsfunktionen wie `malloc()` , `calloc()` und `realloc()` sind erforderlich, um sicherzustellen, dass der zurückgegebene Zeiger für alle Datentypen ausreichend gut ausgerichtet ist, sodass dynamisch zugewiesene Strukturen auch ordnungsgemäß ausgerichtet werden.

Es kann zu ungewöhnlichen Situationen kommen, wie zum Beispiel auf einem 64-Bit-Prozessor Intel x86_64 (z. B. Intel Core i7 - ein Mac mit MacOS Sierra oder Mac OS X), bei dem die Compiler beim Kompilieren im 32-Bit-Modus `double` ausgerichtet auf einem 4-Byte-Grenze; Auf derselben Hardware setzen die Compiler beim Compilieren im 64-Bit-Modus jedoch `double` ausgerichtet auf eine 8-Byte-Grenze.

Strukturpolsterung und Verpackung online lesen: https://riptutorial.com/de/c/topic/4590/strukturpolsterung-und-verpackung

Kapitel 49: Themen (einheimisch)

Syntax

- `#ifndef __STDC_NO_THREADS__`
- `# include <threads.h>`
- `#endif`
- `void call_once(once_flag *flag, void (*func)(void));`
- `int cnd_broadcast(cnd_t *cond);`
- `void cnd_destroy(cnd_t *cond);`
- `int cnd_init(cnd_t *cond);`
- `int cnd_signal(cnd_t *cond);`
- `int cnd_timedwait(cnd_t *restrict cond, mtx_t *restrict mtx, const struct timespec *restrict ts);`
- `int cnd_wait(cnd_t *cond, mtx_t *mtx);`
- `void mtx_destroy(mtx_t *mtx);`
- `int mtx_init(mtx_t *mtx, int type);`
- `int mtx_lock(mtx_t *mtx);`
- `int mtx_timedlock(mtx_t *restrict mtx, const struct timespec *restrict ts);`
- `int mtx_trylock(mtx_t *mtx);`
- `int mtx_unlock(mtx_t *mtx);`
- `int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);`
- `thrd_t thrd_current(void);`
- `int thrd_detach(thrd_t thr);`
- `int thrd_equal(thrd_t thr0, thrd_t thr1);`
- `_Noreturn void thrd_exit(int res);`
- `int thrd_join(thrd_t thr, int *res);`
- `int thrd_sleep(const struct timespec *duration, struct timespec* remaining);`
- `void thrd_yield(void);`
- `int tss_create(tss_t *key, tss_dtor_t dtor);`
- `void tss_delete(tss_t key);`
- `void *tss_get(tss_t key);`
- `int tss_set(tss_t key, void *val);`

Bemerkungen

C11-Threads sind eine optionale Funktion. Ihre Abwesenheit kann mit `__STDC__NO_THREAD__` getestet `__STDC__NO_THREAD__` . Derzeit (Juli 2016) ist diese Funktion noch nicht von allen C-Bibliotheken implementiert, die ansonsten C11 unterstützen.

C-Bibliotheken, von denen bekannt ist, dass sie C11-Threads unterstützen, sind:

- musl

C-Bibliotheken, die noch keine C11-Threads unterstützen:

- Gnu Libc

Examples

Starten Sie mehrere Threads

```
#include <stdio.h>
#include <threads.h>
#include <stdlib.h>

struct my_thread_data {
   double factor;
};

int my_thread_func(void* a) {
   struct my_thread_data* d = a;
   // do something with d
   printf("we found %g\n", d->factor);
   // return an success or error code
   return d->factor > 1.0;
}


int main(int argc, char* argv[argc+1]) {
    unsigned n = 4;
    if (argc > 1) n = strtoull(argv[1], 0, 0);
    // reserve space for the arguments for the threads
    struct my_thread_data D[n];     // can't be initialized
    for (unsigned i = 0; i < n; ++i) {
         D[i] = (struct my_thread_data){ .factor = 0.5*i, };
    }
    // reserve space for the ID's of the threads
    thrd_t id[4];
    // launch the threads
    for (unsigned i = 0; i < n; ++i) {
         thrd_create(&id[i], my_thread_func, &D[i]);
    }
    // Wait that all threads have finished, but throw away their
    // return values
    for (unsigned i = 0; i < n; ++i) {
         thrd_join(id[i], 0);
    }
    return EXIT_SUCCESS;
}
```

Initialisierung durch einen Thread

In den meisten Fällen sollten alle Daten, auf die mehrere Threads zugreifen, initialisiert werden, bevor die Threads erstellt werden. Dadurch wird sichergestellt, dass alle Threads mit einem eindeutigen Status beginnen und keine *Race-Bedingung* auftritt.

Wenn dies nicht möglich ist, können `once_flag` und `call_once` verwendet werden

```
#include <threads.h>
#include <stdlib.h>

// the user data for this example
double const* Big = 0;
```

```
// the flag to protect big, must be global and/or static
static once_flag onceBig = ONCE_INIT;

void destroyBig(void) {
   free((void*)Big);
}

void initBig(void) {
    // assign to temporary with no const qualification
    double* b = malloc(largeNum);
    if (!b) {
       perror("allocation failed for Big");
       exit(EXIT_FAILURE);
    }
    // now initialize and store Big
    initializeBigWithSophisticatedValues(largeNum, b);
    Big = b;
    // ensure that the space is freed on exit or quick_exit
    atexit(destroyBig);
    at_quick_exit(destroyBig);
}

// the user thread function that relies on Big
int myThreadFunc(void* a) {
   call_once(&onceBig, initBig);
   // only use Big from here on
   ...
   return 0;
}
```

Die `once_flag` wird verwendet , um verschiedene Themen zu koordinieren, die die gleichen Daten initialisieren möchten `Big` . Der Aufruf von `call_once` garantiert das

- `initBig` wird genau einmal aufgerufen
- `call_once` blockiert, bis ein solcher Aufruf von `initBig` vom selben oder einem anderen Thread erfolgt ist.

Typisch für eine solche einmal aufgerufene Funktion ist neben der Zuweisung die dynamische Initialisierung von Thread-Steuerungsdatenstrukturen wie `mtx_t` oder `cnd_t` , die nicht statisch mit `mtx_init` bzw. `cnd_init` initialisiert werden `mtx_init` .

Themen (einheimisch) online lesen: https://riptutorial.com/de/c/topic/4432/themen--einheimisch-

Kapitel 50: Typ Qualifiers

Bemerkungen

Typqualifizierer sind die Schlüsselwörter, die zusätzliche Semantik eines Typs beschreiben. Sie sind ein integraler Bestandteil von Typunterschriften. Sie können sowohl auf der obersten Ebene einer Deklaration (direkt auf den Bezeichner) als auch auf Unterebenen (nur für Zeiger, die die Zeigerwerte betreffen) angezeigt werden:

Stichwort	Bemerkungen
`const`	Verhindert die Mutation des deklarierten Objekts (indem es auf der obersten Ebene angezeigt wird) oder verhindert die Mutation des angegebenen Werts (indem er neben einem Zeigertyp angezeigt wird).
`volatile`	Informiert den Compiler, dass das deklarierte Objekt (auf oberster Ebene) oder der angegebene Wert (in Zeigeruntertypen) seinen Wert aufgrund externer Bedingungen ändern kann, nicht nur als Ergebnis eines Programmsteuerungsflusses.
`restrict`	Ein Optimierungshinweis, der nur für Zeiger relevant ist. Erklärt die Absicht, dass während der Lebensdauer des Zeigers keine anderen Zeiger verwendet werden, um auf dasselbe Objekt zuzugreifen.

Die Reihenfolge der Typkennzeichner in Bezug auf Speicherklassenspezifizierer (`static` , `extern` , `auto` , `register`), Typmodifizierer (`signed` , `unsigned` , `short` , `long`) und Typbezeichner (`int` , `char` , `double` usw.) wird nicht erzwungen, sondern Es ist empfehlenswert, sie in die oben genannte Reihenfolge zu bringen:

```
static const volatile unsigned long int a = 5; /* good practice */
unsigned volatile long static int const b = 5; /* bad practice */
```

Qualifikationen auf höchstem Niveau

```
/* "a" cannot be mutated by the program but can change as a result of external conditions */
const volatile int a = 5;

/* the const applies to array elements, i.e. "a[0]" cannot be mutated */
const int arr[] = { 1, 2, 3 };

/* for the lifetime of "ptr", no other pointer could point to the same "int" object */
int *restrict ptr;
```

Zeigertyp-Qualifikationen

```
/* "s1" can be mutated, but "*s1" cannot */
const char *s1 = "Hello";

/* neither "s2" (because of top-level const) nor "*s2" can be mutated */
const char *const s2 = "World";

/* "*p" may change its value as a result of external conditions, "**p" and "p" cannot */
char *volatile *p;

/* "q", "*q" and "**q" may change their values as a result of external conditions */
volatile char *volatile *volatile q;
```

Examples

Unveränderbare (const) Variablen

```
const int a = 0; /* This variable is "unmodifiable", the compiler
                    should throw an error when this variable is changed */
int b = 0; /* This variable is modifiable */

b += 10; /* Changes the value of 'b' */
a += 10; /* Throws a compiler error */
```

Die `const` Qualifikation bedeutet nur, dass wir kein Recht haben, die Daten zu ändern. Das bedeutet nicht, dass sich der Wert hinter unserem Rücken nicht ändern kann.

```
_Bool doIt(double const* a) {
   double rememberA = *a;
   // do something long and complicated that calls other functions

   return rememberA == *a;
}
```

Während der Ausführung der anderen Aufrufe hat sich `*a` möglicherweise geändert, so dass diese Funktion entweder `false` oder `true` .

Warnung

Variablen mit `const` Qualifizierung können noch mit Zeigern geändert werden:

```
const int a = 0;

int *a_ptr = (int*)&a; /* This conversion must be explicitly done with a cast */
*a_ptr += 10;          /* This has undefined behavior */

printf("a = %d\n", a); /* May print: "a = 10" */
```

Dies ist jedoch ein Fehler, der zu undefiniertem Verhalten führt. Die Schwierigkeit hier ist, dass sich dies in einfachen Beispielen wie erwartet verhalten kann, dann aber ein Fehler auftritt, wenn der Code wächst.

Flüchtige Variablen

Das `volatile` Schlüsselwort teilt dem Compiler mit, dass sich der Wert der Variablen jederzeit aufgrund von externen Bedingungen ändern kann, nicht nur als Ergebnis eines Programmsteuerungsflusses.

Der Compiler optimiert nichts, was mit der flüchtigen Variablen zu tun hat.

```
volatile int foo; /* Different ways to declare a volatile variable */
int volatile foo;

volatile uint8_t * pReg; /* Pointers to volatile variable */
uint8_t volatile * pReg;
```

Es gibt zwei Hauptgründe, volatile Variablen zu verwenden:

- Schnittstelle zu Hardware, die E / A-Register mit Speicherzuordnung besitzt.
- Bei Verwendung von Variablen, die außerhalb des Programmsteuerungsflusses geändert werden (z. B. in einer Interrupt-Serviceroutine)

Sehen wir uns dieses Beispiel an:

```
int quit = false;

void main()
{
    ...
    while (!quit) {
      // Do something that does not modify the quit variable
    }
    ...
}

void interrupt_handler(void)
{
  quit = true;
}
```

Der Compiler darf feststellen, dass die while-Schleife die `quit` Variable nicht ändert `quit` und konvertiert die Schleife in eine endlose `while (true)` Schleife `while (true)` . Selbst wenn die `quit` Variable für den Signalhandler für `SIGINT` und `SIGTERM` , weiß dies der Compiler nicht.

Wenn Sie `quit` als `volatile` deklarieren `quit` wird der Compiler angewiesen, die Schleife nicht zu optimieren, und das Problem wird gelöst.

Das gleiche Problem tritt beim Zugriff auf Hardware auf, wie wir in diesem Beispiel sehen:

```
uint8_t * pReg = (uint8_t *) 0x1717;

// Wait for register to become non-zero
while (*pReg == 0) { } // Do something else
```

Das Verhalten des Optimierers besteht darin, den Wert der Variablen einmal zu lesen. Ein

erneutes Lesen ist nicht erforderlich, da der Wert immer derselbe ist. So landen wir mit einer Endlosschleife. Um den Compiler zu zwingen, das zu tun, was wir wollen, ändern wir die Deklaration in:

```
uint8_t volatile * pReg = (uint8_t volatile *) 0x1717;
```

Typ Qualifiers online lesen: https://riptutorial.com/de/c/topic/2588/typ-qualifiers

Kapitel 51: Typedef

Einführung

Mit dem `typedef` Mechanismus können Aliase für andere Typen erstellt werden. Es werden keine neuen Typen erstellt. Menschen verwenden `typedef` häufig, um die Portabilität von Code zu verbessern, um Aliase für Struktur- oder `typedef` zu vergeben oder um Aliase für Funktions- (oder Funktionszeiger-) Typen zu erstellen.

In der C-Norm wird `typedef` als "Speicherklasse" klassifiziert. es tritt syntaktisch auf, wo Speicherklassen wie `static` oder `extern` könnten.

Syntax

- typedef Vorhandener_Name Alias_Name;

Bemerkungen

Nachteile von Typedef

`typedef` kann in großen C-Programmen zur Verschmutzung des Namensraums führen.

Nachteile von Typedef-Strukturen

Auch `typedef` 'd-Strukturen ohne Tag-Namen sind eine Hauptursache für unnötige Auferlegung von Ordnungsbeziehungen zwischen Header-Dateien.

Erwägen:

```
#ifndef FOO_H
#define FOO_H 1

#define FOO_DEF (0xDEADBABE)

struct bar; /* forward declaration, defined in bar.h*/

struct foo {
    struct bar *bar;
};

#endif
```

Mit einer solchen Definition nicht mit `typedefs` , ist es möglich , dass eine Übersetzungseinheit umfassen `foo.h` am bekommen `FOO_DEF` Definition. Wenn nicht versucht wird, den `bar` Member der `foo` Struktur zu `bar.h` Datei `bar.h` nicht `bar.h` werden.

Typedef vs #define

`#define` ist eine C-Vorprozessor-Direktive, die auch zur Definition der Aliase für verschiedene Datentypen verwendet wird, die `typedef` ähnlich sind, jedoch mit den folgenden Unterschieden:

- `typedef` kann nur Typen symbolische Namen geben, bei denen mit `#define` auch Alias für Werte definiert werden kann.
- `typedef` Interpretation wird vom Compiler ausgeführt, während die Anweisungen `#define` vom Vorprozessor verarbeitet werden.
- Beachten Sie, dass `#define cptr char *` gefolgt von `cptr a, b;` macht nicht dasselbe wie `typedef char *cptr;` gefolgt von `cptr a, b;` . Mit der `#define` , `b` ist ein einfacher `char` variable, aber es ist auch ein Zeiger mit dem `typedef` .

Examples

Typedef für Strukturen und Vereinigungen

Sie können einer `struct` Aliasnamen geben:

```
typedef struct Person {
    char name[32];
    int age;
} Person;

Person person;
```

Verglichen mit der traditionellen Methode zum Deklarieren von Strukturen müssen Programmierer nicht jedes Mal eine `struct` haben, wenn sie eine Instanz dieser Struktur deklarieren.

Beachten Sie, dass der Name `Person` (im Gegensatz zu `struct Person`) erst beim abschließenden Semikolon definiert wird. Daher müssen Sie für verknüpfte Listen und Baumstrukturen, die einen Zeiger auf denselben Strukturtyp enthalten müssen, entweder Folgendes verwenden:

```
typedef struct Person {
    char name[32];
    int age;
    struct Person *next;
} Person;
```

oder:

```
typedef struct Person Person;

struct Person {
    char name[32];
    int age;
    Person *next;
};
```

Die Verwendung eines `typedef` für eine `union` Typ ist sehr ähnlich.

```
typedef union Float Float;

union Float
{
    float f;
    char  b[sizeof(float)];
};
```

Eine ähnliche Struktur kann verwendet werden, um die Bytes zu analysieren, aus denen ein `float` Wert besteht.

Einfache Verwendung von Typedef

Um einem Datentyp kurze Namen zu geben

Anstatt:

```
long long int foo;
struct mystructure object;
```

man kann verwenden

```
/* write once */
typedef long long ll;
typedef struct mystructure mystruct;

/* use whenever needed */
ll foo;
mystruct object;
```

Dies reduziert den erforderlichen Schreibaufwand, wenn der Typ mehrmals im Programm verwendet wird.

Portabilität verbessern

Die Attribute von Datentypen variieren je nach Architektur. Beispielsweise kann ein `int` in einer Implementierung ein 2-Byte-Typ und in einer anderen ein 4-Byte-Typ sein. Angenommen, ein Programm muss einen 4-Byte-Typ verwenden, um ordnungsgemäß ausgeführt zu werden.

In einer Implementierung sei die Größe von `int` 2 Bytes und die `long` von 4 Byte. In einem anderen Fall sei die Größe von `int` 4 Bytes und die `long` von 8 Bytes. Wenn das Programm mit der zweiten Implementierung geschrieben wurde,

```
/* program expecting a 4 byte integer */
int foo; /* need to hold 4 bytes to work */
/* some code involving many more ints */
```

Damit das Programm in der ersten Implementierung ausgeführt werden kann, müssen alle `int` Deklarationen in `long` geändert werden.

```
/* program now needs long */
long foo; /*need to hold 4 bytes to work */
/* some code involving many more longs - lot to be changed */
```

Um dies zu vermeiden, kann man `typedef`

```
/* program expecting a 4 byte integer */
typedef int myint; /* need to declare once - only one line to modify if needed */
myint foo; /* need to hold 4 bytes to work */
/* some code involving many more myints */
```

Dann muss nur die `typedef` Anweisung jedes Mal geändert werden, anstatt das gesamte Programm zu untersuchen.

C99

Der Header `<stdint.h>` und der zugehörige Header `<inttypes.h>` definieren Standardtypnamen (mithilfe von `typedef`) für Ganzzahlen verschiedener Größen. Diese Namen sind oft die beste Wahl in modernem Code, für den Ganzzahlen mit fester Größe erforderlich sind. Zum Beispiel ist `uint8_t` ein vorzeichenloser 8-Bit-Integer-Typ. `int64_t` ist ein 64-Bit-Integer-Typ mit Vorzeichen. Der Typ `uintptr_t` ist ein vorzeichenloser Integer-Typ, der groß genug ist, um einen Zeiger auf ein Objekt aufzunehmen. Diese Typen sind theoretisch optional - sie sind jedoch selten verfügbar. Es gibt Varianten wie `uint_least16_t` (der kleinste vorzeichenlose Integer-Typ mit mindestens 16 Bit) und `int_fast32_t` (der am schnellsten vorzeichenbehaftete Integer-Typ mit mindestens 32 Bit). Außerdem sind `intmax_t` und `uintmax_t` die größten Integer-Typen, die von der Implementierung unterstützt werden. Diese Typen sind obligatorisch.

Zur Angabe einer Verwendung oder zur Verbesserung der Lesbarkeit

Wenn eine Gruppe von Daten einen bestimmten Zweck `typedef` , können Sie mit `typedef` einen aussagekräftigen Namen vergeben. Wenn die Eigenschaft der Daten sich so ändert, dass der Basistyp geändert werden muss, müsste außerdem nur die Anweisung `typedef` geändert werden, anstatt das gesamte Programm zu untersuchen.

Typedef für Funktionszeiger

Wir können `typedef` , um die Verwendung von Funktionszeigern zu vereinfachen. Stellen Sie sich vor, wir haben einige Funktionen, die alle dieselbe Signatur haben und deren Argument verwendet, um etwas auf unterschiedliche Weise auszudrucken:

```
#include<stdio.h>

void print_to_n(int n)
{
    for (int i = 1; i <= n; ++i)
        printf("%d\n", i);
}

void print_n(int n)
{
    printf("%d\n, n);
```

```
}
```

Jetzt können wir mit einem `typedef` einen benannten Funktionszeigertyp namens Drucker erstellen:

```
typedef void (*printer_t)(int);
```

Dadurch wird ein Typ mit dem Namen `printer_t` für einen Zeiger auf eine Funktion erstellt, die ein einzelnes `int` Argument `printer_t` und nichts zurückgibt, was der Signatur der oben genannten Funktionen entspricht. Um es zu verwenden, erstellen wir eine Variable des erstellten Typs und weisen ihr einen Zeiger auf eine der fraglichen Funktionen zu:

```
printer_t p = &print_to_n;
void (*p)(int) = &print_to_n; // This would be required without the type
```

Dann rufen Sie die Funktion auf, auf die die Funktionszeigervariable zeigt:

```
p(5);           // Prints 1 2 3 4 5 on separate lines
(*p)(5);        // So does this
```

Somit erlaubt das `typedef` eine einfachere Syntax im Umgang mit Funktionszeigern. Dies wird deutlicher, wenn Funktionszeiger in komplexeren Situationen verwendet werden, z. B. Argumente für Funktionen.

Wenn Sie eine Funktion verwenden, die einen Funktionszeiger als Parameter verwendet, ohne dass ein Funktionszeigertyp definiert ist, wäre die Funktionsdefinition

```
void foo (void (*printer)(int), int y){
    //code
    printer(y);
    //code
}
```

Mit dem `typedef` es jedoch:

```
void foo (printer_t printer, int y){
    //code
    printer(y);
    //code
}
```

Ebenso können Funktionen Funktionszeiger zurückgeben, und die Verwendung eines `typedef` kann dabei die Syntax vereinfachen.

Ein klassisches Beispiel ist die `signal` von `<signal.h>` . Die Deklaration dafür (aus dem C-Standard) lautet:

```
void (*signal(int sig, void (*func)(int)))(int);
```

Dies ist eine Funktion, die zwei Argumente `int` - ein `int` und einen Zeiger auf eine Funktion, die ein `int` als Argument verwendet und nichts zurückgibt - und das einen Zeiger auf die Funktion wie sein zweites Argument zurückgibt.

Wenn wir einen Typ `SigCatcher` als Alias für den Zeiger auf den Funktionstyp definiert haben:

```
typedef void (*SigCatcher)(int);
```

dann könnten wir `signal()` deklarieren mit:

```
SigCatcher signal(int sig, SigCatcher func);
```

Im Großen und Ganzen ist dies einfacher zu verstehen (auch wenn der C-Standard sich nicht dafür entschieden hat, einen Typ zu definieren, der die Aufgabe erfüllt). Die `signal` nimmt zwei Argumente, einen `int` und einen `SigCatcher` , und es gibt eine `SigCatcher` - wo ein `SigCatcher` einen Zeiger auf eine Funktion, die eine dauert `int` Argument und gibt nichts.

Die Verwendung von `typedef` Namen für Zeiger auf Funktionstypen macht das Leben zwar einfacher, kann aber auch für andere `typedef` , die Ihren Code später verwalten werden. `typedef` Sie daher mit Vorsicht und der richtigen Dokumentation vor. Siehe auch Funktionszeiger .

Typedef online lesen: https://riptutorial.com/de/c/topic/2681/typedef

Kapitel 52: Übergeben Sie 2D-Arrays an Funktionen

Examples

Übergeben Sie ein 2D-Array an eine Funktion

Die Übergabe eines 2D-Arrays an eine Funktion erscheint einfach und naheliegend und wir schreiben gerne:

```
#include <stdio.h>
#include <stdlib.h>

#define ROWS 3
#define COLS 2

void fun1(int **, int, int);

int main()
{
  int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
  int n = ROWS;
  int m = COLS;

  fun1(array_2D, n, m);

  return EXIT_SUCCESS;
}

void fun1(int **a, int n, int m)
{
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

Aber der Compiler, hier GCC in Version 4.9.4, schätzt es nicht.

```
$ gcc-4.9 -O3 -g3  -W -Wall -Wextra  -std=c11 passarr.c -o passarr
passarr.c: In function ‘main’:
passarr.c:16:8: warning: passing argument 1 of ‘fun1’ from incompatible pointer type
   fun1(array_2D, n, m);
        ^
passarr.c:8:6: note: expected ‘int **’ but argument is of type ‘int (*)[2]’
 void fun1(int **, int, int);
```

Dafür gibt es zwei Gründe: Das Hauptproblem besteht darin, dass Arrays keine Zeiger sind und der zweite Nachteil ist der sogenannte *Zeigerzerfall* . Durch die Übergabe eines Arrays an eine Funktion wird das Array in einen Zeiger auf das erste Element des Arrays zerlegt - im Falle eines

2d-Arrays zerfällt es in einen Zeiger auf die erste Zeile, da in C-Arrays die Zeilen zuerst sortiert werden.

```
#include <stdio.h>
#include <stdlib.h>

#define ROWS 3
#define COLS 2

void fun1(int (*)[COLS], int, int);

int main()
{
  int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
  int n = ROWS;
  int m = COLS;

  fun1(array_2D, n, m);

  return EXIT_SUCCESS;
}

void fun1(int (*a)[COLS], int n, int m)
{
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

Es *ist* notwendig, die Anzahl der Zeilen zu übergeben, sie können nicht berechnet werden.

```
#include <stdio.h>
#include <stdlib.h>

#define ROWS 3
#define COLS 2

void fun1(int (*)[COLS], int);

int main()
{
  int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
  int rows = ROWS;

  /* works here because array_2d is still in scope and still an array */
  printf("MAIN: %zu\n",sizeof(array_2D)/sizeof(array_2D[0]));

  fun1(array_2D, rows);

  return EXIT_SUCCESS;
}

void fun1(int (*a)[COLS], int rows)
{
  int i, j;
  int n, m;
```

```
  n = rows;
  /* Works, because that information is passed (as "COLS").
     It is also redundant because that value is known at compile time (in "COLS"). */
  m = (int) (sizeof(a[0])/sizeof(a[0][0]));

  /* Does not work here because the "decay" in "pointer decay" is meant
     literally--information is lost. */
  printf("FUN1: %zu\n",sizeof(a)/sizeof(a[0]));

  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

C99

Die Anzahl der Spalten ist vordefiniert und somit zur Kompilierzeit festgelegt. Der Vorgänger des aktuellen C-Standards (dh ISO / IEC 9899: 1999, derzeit ISO / IEC 9899: 2011) implementierte VLAs (TODO: link it) und Obwohl der aktuelle Standard dies optional macht, unterstützen ihn fast alle modernen C-Compiler (TODO: Prüfen Sie, ob MS Visual Studio es jetzt unterstützt).

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMMITTED!*/

void fun1(int (*)[], int rows, int cols);

int main(int argc, char **argv)
{
  int rows, cols, i, j;

  if(argc != 3){
     fprintf(stderr,"Usage: %s rows cols\n",argv[0]);
     exit(EXIT_FAILURE);
  }

  rows = atoi(argv[1]);
  cols = atoi(argv[2]);

  int array_2D[rows][cols];

  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      array_2D[i][j] = (i + 1) * (j + 1);
      printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
    }
  }

  fun1(array_2D, rows, cols);

  exit(EXIT_SUCCESS);
}

void fun1(int (*a)[], int rows, int cols)
{
```

```
  int i, j;
  int n, m;

  n = rows;
  /* Does not work anymore, no sizes are specified anymore
  m = (int) (sizeof(a[0])/sizeof(a[0][0])); */
  m = cols;


  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

Das funktioniert nicht, der Compiler beschwert sich:

```
$ gcc-4.9 -O3 -g3  -W -Wall -Wextra  -std=c99 passarr.c -o passarr
passarr.c: In function ‘fun1’:
passarr.c:168:7: error: invalid use of array with unspecified bounds
       printf("array[%d][%d]=%d\n", i, j, a[i][j]);
```

Etwas klarer wird es, wenn wir beim Aufruf der Funktion absichtlich einen Fehler machen, indem wir die Deklaration in `void fun1(int **a, int rows, int cols)` . Dies hat zur Folge, dass sich der Compiler auf eine andere, aber ebenso nebulöse Weise beschwert

```
$ gcc-4.9 -O3 -g3  -W -Wall -Wextra  -std=c99 passarr.c -o passarr
passarr.c: In function ‘main’:
passarr.c:208:8: warning: passing argument 1 of ‘fun1’ from incompatible pointer type
   fun1(array_2D, rows, cols);
        ^
passarr.c:185:6: note: expected ‘int **’ but argument is of type ‘int (*)[(sizetype)(cols)]’
 void fun1(int **, int rows, int cols);
```

Wir können auf verschiedene Arten reagieren, eine davon besteht darin, alles davon zu ignorieren und unleserliche Zeigerverknüppelungen durchzuführen:

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMMITTED!*/

void fun1(int (*)[], int rows, int cols);

int main(int argc, char **argv)
{
  int rows, cols, i, j;

  if(argc != 3){
     fprintf(stderr,"Usage: %s rows cols\n",argv[0]);
     exit(EXIT_FAILURE);
  }

  rows = atoi(argv[1]);
  cols = atoi(argv[2]);
```

```
  int array_2D[rows][cols];
  printf("Make array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      array_2D[i][j] = i * cols + j;
      printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
    }
  }

  fun1(array_2D, rows, cols);

  exit(EXIT_SUCCESS);
}

void fun1(int (*a)[], int rows, int cols)
{
  int i, j;
  int n, m;

  n = rows;
  m = cols;

  printf("\nPrint array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, *( (*a) + (i * cols + j)));
    }
  }
}
```

Oder wir machen es richtig und geben die benötigten Informationen an `fun1` . Dazu müssen wir die Argumente in `fun1` : Die Größe der Spalte muss vor der Deklaration des Arrays liegen. Um die Lesbarkeit zu verbessern, hat die Variable, die die Anzahl der Zeilen enthält, ebenfalls ihren Platz geändert und ist jetzt der erste.

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMMITTED!*/

void fun1(int rows, int cols, int (*)[]);

int main(int argc, char **argv)
{
  int rows, cols, i, j;

  if(argc != 3){
     fprintf(stderr,"Usage: %s rows cols\n",argv[0]);
     exit(EXIT_FAILURE);
  }

  rows = atoi(argv[1]);
  cols = atoi(argv[2]);

  int array_2D[rows][cols];
  printf("Make array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
```

```
      array_2D[i][j] = i * cols + j;
      printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
    }
  }

  fun1(rows, cols, array_2D);

  exit(EXIT_SUCCESS);
}

void fun1(int rows, int cols, int (*a)[cols])
{
  int i, j;
  int n, m;

  n = rows;
  m = cols;

  printf("\nPrint array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

Dies scheint einigen Leuten unangenehm zu sein, die der Meinung sind, dass die Reihenfolge der Variablen keine Rolle spielen sollte. Das ist kein großes Problem, deklarieren Sie einfach einen Zeiger und lassen Sie ihn auf das Array zeigen.

```
#include <stdio.h>
#include <stdlib.h>

/* ALL CHECKS OMMITTED!*/

void fun1(int rows, int cols, int **);

int main(int argc, char **argv)
{
  int rows, cols, i, j;


  if(argc != 3){
     fprintf(stderr,"Usage: %s rows cols\n",argv[0]);
     exit(EXIT_FAILURE);
  }

  rows = atoi(argv[1]);
  cols = atoi(argv[2]);

  int array_2D[rows][cols];
  printf("Make array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      array_2D[i][j] = i * cols + j;
      printf("array[%d][%d]=%d\n", i, j, array_2D[i][j]);
    }
  }
  // a "rows" number of pointers to "int". Again a VLA
```

```
  int *a[rows];
  // initialize them to point to the individual rows
  for (i = 0; i < rows; i++) {
      a[i] = array_2D[i];
  }

  fun1(rows, cols, a);

  exit(EXIT_SUCCESS);
}

void fun1(int rows, int cols, int **a)
{
  int i, j;
  int n, m;

  n = rows;
  m = cols;

  printf("\nPrint array with %d rows and %d columns\n", rows, cols);
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

Verwenden von flachen Arrays als 2D-Arrays

Die einfachste Lösung besteht oft darin, 2D- und höhere Arrays als flachen Speicher zu übergeben.

```
/* create 2D array with dimensions determined at runtime */
double *matrix = malloc(width * height * sizeof(double));

/* initialise it (for the sake of illustration we want 1.0 on the diagonal) */
int x, y;
for (y = 0; y < height; y++)
{
    for (x = 0; x < width; x++)
    {
        if (x == y)
            matrix[y * width + x] = 1.0;
        else
            matrix[y * width + x] = 0.0;
    }
}

/* pass it to a subroutine */
 manipulate_matrix(matrix, width, height);


/* do something with the matrix, e.g. scale by 2 */
void manipulate_matrix(double *matrix, int width, int height)
{
    int x, y;

    for (y = 0; y < height; y++)
    {
```

```
        for (x = 0; x < width; x++)
        {
            matrix[y * width + x] *= 2.0;
        }
    }
}
```

Übergeben Sie 2D-Arrays an Funktionen online lesen:
https://riptutorial.com/de/c/topic/6862/ubergeben-sie-2d-arrays-an-funktionen

Kapitel 53: Undefiniertes Verhalten

Einführung

In C führen einige Ausdrücke zu *undefiniertem Verhalten* . Der Standard wählt explizit aus, nicht zu definieren, wie sich ein Compiler verhalten soll, wenn er auf einen solchen Ausdruck stößt. Daher kann ein Compiler das tun, was er für richtig hält, und kann nützliche Ergebnisse, unerwartete Ergebnisse oder sogar einen Absturz verursachen.

Code, der UB aufruft, funktioniert möglicherweise auf einem bestimmten System mit einem bestimmten Compiler wie beabsichtigt, wird aber wahrscheinlich nicht auf einem anderen System oder mit einem anderen Compiler, einer Compilerversion oder Compilereinstellungen funktionieren.

Bemerkungen

Was ist undefiniertes Verhalten (UB)?

Nicht definiertes Verhalten ist ein Begriff, der im C-Standard verwendet wird. Der C11-Standard (ISO / IEC 9899: 2011) definiert den Begriff undefiniertes Verhalten als

> Verhalten bei Verwendung eines nicht portablen oder fehlerhaften Programmkonstrukts oder fehlerhafter Daten, für das diese Internationale Norm keine Anforderungen auferlegt

Was passiert, wenn mein Code UB enthält?

Dies sind die Ergebnisse, die aufgrund von undefiniertem Verhalten gemäß Standard auftreten können:

> ANMERKUNG Das mögliche undefinierte Verhalten reicht vom Ignorieren der Situation mit unvorhersehbaren Ergebnissen über das dokumentierte Verhalten der Übersetzung (mit oder ohne Ausgabe einer Diagnosemeldung) während der Übersetzung oder Programmausführung bis zum Abbruch einer Übersetzung oder Ausführung (mit der Option Ausgabe einer Diagnosemeldung).

Das folgende Zitat wird häufig verwendet, um (weniger formell) Ergebnisse zu beschreiben, die auf undefiniertes Verhalten zurückzuführen sind:

> "Wenn der Compiler [einem bestimmten undefinierten Konstrukt] begegnet, ist es legal, Dämonen aus der Nase fliegen zu lassen."

Warum gibt es UB?

Wenn es so schlimm ist, warum haben sie es nicht einfach definiert oder durch Implementierung definiert?

Undefiniertes Verhalten bietet mehr Möglichkeiten zur Optimierung; Der Compiler kann zu Recht davon ausgehen, dass jeder Code kein undefiniertes Verhalten enthält, wodurch Laufzeitprüfungen vermieden und Optimierungen durchgeführt werden können, deren Gültigkeit kostspielig ist oder nicht nachgewiesen werden kann.

Warum ist UB schwer zu finden?

Es gibt mindestens zwei Gründe, warum undefiniertes Verhalten Fehler verursacht, die schwer zu erkennen sind:

- Der Compiler muss nicht - und kann Sie im Allgemeinen nicht zuverlässig - vor undefiniertem Verhalten warnen. Ein entsprechendes Erfordernis würde direkt gegen den Grund für undefiniertes Verhalten verstoßen.
- Die unvorhersehbaren Ergebnisse beginnen sich möglicherweise nicht genau an der Stelle zu entwickeln, an der das Konstrukt auftritt, dessen Verhalten undefiniert ist. Undefiniertes Verhalten verfälscht die gesamte Ausführung und ihre Auswirkungen können jederzeit auftreten: Während, nach oder sogar *vor* dem undefinierten Konstrukt.

Betrachten Sie eine Null-Zeiger-Dereferenzierung: Der Compiler ist nicht für die Diagnose einer Null-Zeiger-Dereferenzierung erforderlich und könnte dies auch nicht, da zur Laufzeit jeder in eine Funktion übergebene Zeiger oder in einer globalen Variablen Null sein kann. *Und wenn die Null-Zeiger-Dereferenzierung auftritt, schreibt der Standard nicht vor, dass das Programm abstürzen muss.* Das Programm kann vielmehr früher, später oder überhaupt nicht abstürzen. Es könnte sich sogar so verhalten, als ob der Nullzeiger auf ein gültiges Objekt zeigte, und sich völlig normal verhält, nur unter anderen Umständen zum Absturz.

Im Falle von Null-Zeiger-Dereferenzierung unterscheidet sich die Sprache C von verwalteten Sprachen wie Java oder C #, in denen das Verhalten der Null-Zeiger-Dereferenzierung *definiert ist* : Es wird genau zu der Zeit eine Ausnahme ausgelöst (`NullPointerException` in Java, `NullReferenceException` in C #) Diejenigen, die aus Java oder C # stammen, könnten daher *fälschlicherweise glauben, dass ein C-Programm mit oder ohne Ausgabe einer Diagnosemeldung abstürzen muss* .

Zusätzliche Information

Es gibt mehrere solcher Situationen, die klar unterschieden werden sollten:

- Ausdrücklich undefiniertes Verhalten. Hier weist der C-Standard explizit darauf hin, dass Sie sich nicht im Grenzbereich befinden.
- Implizit undefiniertes Verhalten, bei dem der Standard einfach keinen Text enthält, der ein Verhalten für die Situation vorsieht, in die Sie Ihr Programm eingebracht haben.

Bedenken Sie auch, dass das Verhalten bestimmter Konstrukte an vielen Stellen durch den C-Standard absichtlich undefiniert ist, um Raum für Compiler- und Bibliotheksimplementierer zu lassen, um eigene Definitionen zu entwickeln. Ein gutes Beispiel sind Signale und Signalhandler, bei denen Erweiterungen von C, wie beispielsweise der Betriebssystemstandard POSIX, weitaus detailliertere Regeln definieren. In solchen Fällen müssen Sie nur die Dokumentation Ihrer Plattform überprüfen. Der C-Standard kann Ihnen nichts sagen.

Beachten Sie auch, dass wenn undefiniertes Verhalten in einem Programm auftritt, das nicht bedeutet, dass nur der Punkt, an dem undefiniertes Verhalten aufgetreten ist, problematisch ist, vielmehr wird das gesamte Programm bedeutungslos.

Aufgrund solcher Bedenken ist es für die Personenprogrammierung in C wichtig (vor allem, da Compiler uns nicht immer vor UB warnen), zumindest mit den Dingen vertraut zu sein, die undefiniertes Verhalten auslösen.

Es sei darauf hingewiesen, dass es einige Tools gibt (z. B. statische Analysewerkzeuge wie PC-Lint), die das Erkennen von undefiniertem Verhalten unterstützen, aber auch hier nicht alle Vorkommen von undefiniertem Verhalten erkennen können.

Examples

Dereferenzieren Sie einen Nullzeiger

Dies ist ein Beispiel für die Dereferenzierung eines NULL-Zeigers, wodurch undefiniertes Verhalten verursacht wird.

```
int * pointer = NULL;
int value = *pointer; /* Dereferencing happens here */
```

Der C-Standard garantiert einen `NULL` Zeiger, um ungleiche Zeiger auf ein gültiges Objekt zu vergleichen, und dereferenziert es, undefiniert dieses Verhalten.

Objekte mehr als einmal zwischen zwei Sequenzpunkten ändern

```
int i = 42;
i = i++; /* Assignment changes variable, post-increment as well */
int a = i++ + i--;
```

Code wie dieser führt oft zu Spekulationen über den "Ergebniswert" von `i` . Anstatt ein Ergebnis anzugeben, geben die C-Standards jedoch an, dass die Auswertung eines solchen Ausdrucks zu *undefiniertem Verhalten führt* . Vor C2011 formalisierte der Standard diese Regeln in Form sogenannter *Sequenzpunkte* :

> Zwischen dem vorherigen und dem nächsten Sequenzpunkt soll ein gespeicherter Wert eines skalaren Objekts höchstens einmal durch die Auswertung eines Ausdrucks geändert werden. Außerdem soll der vorherige Wert nur gelesen werden, um den zu speichernden Wert zu bestimmen.

(Standard C99, Abschnitt 6.5, Absatz 2)

Dieses Schema erwies sich als etwas zu grob, was dazu führte, dass einige Ausdrücke in Bezug auf C99 undefiniertes Verhalten zeigten, das plausibel nicht funktionieren sollte. C2011 behält die Sequenzpunkte bei, führt jedoch einen differenzierteren Ansatz für diesen Bereich basierend auf der *Sequenzierung* und einer Beziehung ein, die als "Sequenzierung zuvor" bezeichnet wird:

> Wenn ein Nebeneffekt auf ein Skalarobjekt relativ zu einem anderen Nebeneffekt auf demselben Skalarobjekt oder einer Wertberechnung mit dem Wert desselben Skalarobjekts ohne Folgen bleibt, ist das Verhalten undefiniert. Wenn es mehrere zulässige Anordnungen der Unterausdrücke eines Ausdrucks gibt, ist das Verhalten undefiniert, wenn ein solcher nicht aufeinanderfolgender Nebeneffekt in einer der Anordnungen auftritt.

(Standard C2011, Abschnitt 6.5, Absatz 2)

Die vollständigen Details der "Vorher-sequenzierten" Relation sind zu lang, um sie hier zu beschreiben, ergänzen jedoch Sequenzpunkte, anstatt sie zu ersetzen, so dass sie Verhalten für einige Auswertungen definieren, deren Verhalten zuvor undefiniert war. Wenn sich zwischen zwei Auswertungen ein Sequenzpunkt befindet, wird der vor dem Sequenzpunkt vor dem nachfolgenden Sequenzpunkt "sequenziert".

Das folgende Beispiel hat ein genau definiertes Verhalten:

```
int i = 42;
i = (i++, i+42); /* The comma-operator creates a sequence point */
```

Das folgende Beispiel hat ein undefiniertes Verhalten:

```
int i = 42;
printf("%d %d\n", i++, i++); /* commas as separator of function arguments are not comma-
operators */
```

Wie bei jeder Form von undefiniertem Verhalten ist das Beobachten des tatsächlichen Verhaltens bei der Auswertung von Ausdrücken, die gegen die Sequenzierungsregeln verstoßen, nicht aufschlussreich, es sei denn, dies ist rückblickend. Der Sprachstandard bietet keine Grundlage, um zu erwarten, dass solche Beobachtungen sogar das zukünftige Verhalten desselben Programms vorhersagen.

Fehlende return-Anweisung in der Funktion zur Wertrückgabe

```
int foo(void) {
  /* do stuff */
  /* no return here */
}

int main(void) {
  /* Trying to use the (not) returned value causes UB */
  int value = foo();
  return 0;
}
```

Wenn für eine Funktion ein Wert zurückgegeben wird, muss dies für jeden möglichen Codepfad erfolgen. Ein undefiniertes Verhalten tritt auf, sobald der Aufrufer (der einen Rückgabewert erwartet) den Rückgabewert [1 verwendet] .

Beachten Sie, dass das undefinierte Verhalten *nur* auftritt *, wenn* der Aufrufer versucht, den Wert

der Funktion zu verwenden / darauf zuzugreifen. Zum Beispiel,

```
int foo(void) {
  /* do stuff */
  /* no return here */
}

int main(void) {
  /* The value (not) returned from foo() is unused. So, this program
   * doesn't cause *undefined behaviour*. */
  foo();
  return 0;
}
```

C99

Die `main()` Funktion ist eine Ausnahme von dieser Regel, da sie ohne return-Anweisung beendet werden kann, da in diesem Fall automatisch ein angenommener Rückgabewert von `0` verwendet wird [2] .

---

[1] (*ISO / IEC 9899: 201x* , 6.9.1 / 12)

> Wenn das}, das eine Funktion beendet, erreicht wird und der Wert des Funktionsaufrufs vom Aufrufer verwendet wird, ist das Verhalten undefiniert.

[2] (*ISO / IEC 9899: 201x* , 5.1.2.2.3 / 1)

> Wenn Sie das} erreichen, das die Hauptfunktion beendet, wird der Wert 0 zurückgegeben.

Überlauf der signierten Ganzzahl

Gemäß Absatz 6.5 / 5 von C99 und C11 führt die Auswertung eines Ausdrucks zu undefiniertem Verhalten, wenn das Ergebnis kein darstellbarer Wert des Typs des Ausdrucks ist. Für arithmetische Typen wird dies als *Überlauf bezeichnet* . Die Ganzzahl-Arithmetik ohne Vorzeichen läuft nicht über, weil Absatz 6.2.5 / 9 gilt. Dies führt dazu, dass vorzeichenlose Ergebnisse, die ansonsten außerhalb des Bereichs liegen, auf einen Wert innerhalb des Bereichs reduziert werden. Es gibt jedoch keine analoge Bestimmung für *vorzeichenbehaftete* Integer-Typen. Diese können und machen einen Überlauf und erzeugen undefiniertes Verhalten. Zum Beispiel,

```
#include <limits.h>      /* to get INT_MAX */

int main(void) {
    int i = INT_MAX + 1; /* Overflow happens here */
    return 0;
}
```

Die meisten Fälle dieser Art von undefiniertem Verhalten sind schwieriger zu erkennen oder vorherzusagen. Ein Überlauf kann prinzipiell aus jeder Addition, Subtraktion oder Multiplikation von vorzeichenbehafteten Ganzzahlen (vorbehaltlich der üblichen arithmetischen Konvertierungen) entstehen, bei denen keine wirksamen Grenzen oder eine Beziehung zwischen

den Operanden bestehen, um dies zu verhindern. Zum Beispiel diese Funktion:

```
int square(int x) {
    return x * x;  /* overflows for some values of x */
}
```

ist vernünftig und tut das Richtige für Argumentwerte, die klein genug sind, aber für größere Argumentwerte ist das Verhalten nicht definiert. Sie können nicht allein anhand der Funktion beurteilen, ob Programme, die sie aufrufen, als Ergebnis undefiniertes Verhalten zeigen. Es hängt davon ab, mit welchen Argumenten sie darüber sprechen.

Betrachten Sie andererseits dieses triviale Beispiel für eine überlaufsichere Ganzzahlarithmetik:

```
int zero(int x) {
    return x - x;  /* Cannot overflow */
}
```

Die Beziehung zwischen den Operanden des Subtraktionsoperators stellt sicher, dass die Subtraktion niemals überläuft. Oder betrachten Sie dieses etwas praktischere Beispiel:

```
int sizeDelta(FILE *f1, FILE *f2) {
    int count1 = 0;
    int count2 = 0;
    while (fgetc(f1) != EOF) count1++;  /* might overflow */
    while (fgetc(f2) != EOF) count2++;  /* might overflow */

    return count1 - count2; /* provided no UB to this point, will not overflow */
}
```

Solange die Zähler nicht einzeln überlaufen, sind die Operanden der letzten Subtraktion beide nicht negativ. Alle Unterschiede zwischen zwei beliebigen Werten können als `int` .

Verwendung einer nicht initialisierten Variablen

```
int a;
printf("%d", a);
```

Die Variable `a` ist ein `int` mit automatischer Speicherdauer. Der obige Beispielcode versucht, den Wert einer nicht initialisierten Variablen zu drucken (`a` wurde nie initialisiert). Automatische Variablen, die nicht initialisiert werden, haben unbestimmte Werte. Der Zugriff auf diese kann zu undefiniertem Verhalten führen.

Anmerkung: Variablen mit lokalem statischem oder Thread-Speicher, einschließlich globaler Variablen ohne das Schlüsselwort `static` , werden entweder auf Null oder auf ihren initialisierten Wert gesetzt. Daher ist das Folgende legal.

```
static int b;
printf("%d", b);
```

Ein sehr häufiger Fehler ist es , nicht die Variablen zu initialisieren , die als Zähler auf 0. Sie dienen Werte zu ergänzen, aber da der Anfangswert Müll ist, werden Sie **nicht definiertes Verhalten,** wie zum Beispiel in der Frage aufrufen Kompilierung auf Terminal verströmt Zeiger Warnung und seltsame Symbole .

Beispiel:

```
#include <stdio.h>

int main(void) {
    int i, counter;
    for(i = 0; i < 10; ++i)
        counter += i;
    printf("%d\n", counter);
    return 0;
}
```

Ausgabe:

```
C02QT2UBFVH6-lm:~ gsamaras$ gcc main.c -Wall -o main
main.c:6:9: warning: variable 'counter' is uninitialized when used here [-Wuninitialized]
        counter += i;
        ^~~~~~~
main.c:4:19: note: initialize the variable 'counter' to silence this warning
    int i, counter;
                  ^
                   = 0
1 warning generated.
C02QT2UBFVH6-lm:~ gsamaras$ ./main
32812
```

Die obigen Regeln gelten auch für Zeiger. Das folgende führt beispielsweise zu undefiniertem Verhalten

```
int main(void)
{
    int *p;
    p++; // Trying to increment an uninitialized pointer.
}
```

Beachten Sie, dass der obige Code alleine möglicherweise keinen Fehler oder Segmentierungsfehler verursacht. Wenn Sie diesen Zeiger später dereferenzieren, wird dies jedoch zu einem undefinierten Verhalten führen.

Dereferenzieren eines Zeigers auf eine Variable über ihre Lebensdauer hinaus

```
int* foo(int bar)
{
    int baz = 6;
    baz += bar;
    return &baz; /* (&baz) copied to new memory location outside of foo. */
} /* (1) The lifetime of baz and bar end here as they have automatic storage
   * duration (local variables), thus the returned pointer is not valid! */
```

```
int main (void)
{
    int* p;

    p = foo(5);  /* (2) this expression's behavior is undefined */
    *p = *p - 6; /* (3) Undefined behaviour here */

    return 0;
}
```

Einige Compiler weisen darauf hin. Zum Beispiel warnt `gcc` mit:

```
warning: function returns address of local variable [-Wreturn-local-addr]
```

und `clang` warnt mit:

```
warning: address of stack memory associated with local variable 'baz' returned
[-Wreturn-stack-address]
```

für den obigen Code. Compiler können jedoch möglicherweise nicht in komplexem Code helfen.

(1) Das Zurückgeben eines Verweises auf eine als `static` deklarierte Variable ist ein definiertes Verhalten, da die Variable nach Verlassen des aktuellen Gültigkeitsbereichs nicht gelöscht wird.

(2) Gemäß ISO / IEC 9899: 2011 6.2.4 §2 "Der Wert eines Zeigers wird unbestimmt, wenn das Objekt, auf das er zeigt, das Ende seiner Lebensdauer erreicht."

(3) Der Rückschluss auf den von der Funktion `foo` Zeiger ist undefiniertes Verhalten, da der Speicher, auf den er verweist, einen unbestimmten Wert enthält.

Durch Null teilen

```
int x = 0;
int y = 5 / x;  /* integer division */
```

oder

```
double x = 0.0;
double y = 5.0 / x;  /* floating point division */
```

oder

```
int x = 0;
int y = 5 % x;  /* modulo operation */
```

Für die zweite Zeile in jedem Beispiel, in der der Wert des zweiten Operanden (x) Null ist, ist das Verhalten undefiniert.

Beachten Sie, dass die meisten Implementierungen der Gleitkomma-Mathematik einem Standard

folgen (z. B. IEEE 754). In diesem Fall haben Operationen wie Division durch Null gleichbleibende Ergebnisse (z. B. `INFINITY`), obwohl der C-Standard sagt, dass die Operation undefiniert ist.

Zugriff auf Speicherplatz außerhalb des zugewiesenen Blocks

Ein Zeiger auf einen Speicherbereich, der `n` Elemente enthält, kann nur dereferenziert werden, wenn er sich im Bereich `memory` und `memory + (n - 1)` . Wenn Sie einen Zeiger außerhalb dieses Bereichs referenzieren, führt dies zu undefiniertem Verhalten. Betrachten Sie als Beispiel den folgenden Code:

```
int array[3];
int *beyond_array = array + 3;
*beyond_array = 0; /* Accesses memory that has not been allocated. */
```

Die dritte Zeile greift auf das vierte Element in einem nur 3 Elemente langen Array zu, was zu undefiniertem Verhalten führt. Ebenso ist das Verhalten der zweiten Zeile in dem folgenden Codefragment nicht gut definiert:

```
int array[3];
array[3] = 0;
```

Beachten Sie, dass das Zeigen auf das letzte Element eines Arrays kein undefiniertes Verhalten ist (`beyond_array = array + 3` ist hier gut definiert), aber dereferenzierend ist (`*beyond_array` ist undefiniertes Verhalten). Diese Regel gilt auch für dynamisch zugewiesenen Speicher (z. B. durch `malloc` erzeugte Puffer).

Überlappenden Speicher kopieren

Eine Vielzahl von Standard-Bibliotheksfunktionen hat unter anderem Auswirkungen auf das Kopieren von Bytefolgen von einem Speicherbereich in einen anderen. Die meisten dieser Funktionen haben undefiniertes Verhalten, wenn sich die Quell- und Zielregionen überlappen.

Zum Beispiel das ...

```
#include <string.h> /* for memcpy() */

char str[19] = "This is an example";
memcpy(str + 7, str, 10);
```

... versucht, 10 Bytes zu kopieren, wobei sich der Quell- und der Zielspeicherbereich um drei Bytes überschneiden. Visualisieren:

```
               overlapping area
               |
               _ _
              |   |
              v   v
T h i s   i s   a n   e x a m p l e \0
^             ^
|             |
```

```
|             destination
|
source
```

Aufgrund der Überlappung ist das resultierende Verhalten undefiniert.

Zu den Standard-Bibliotheksfunktionen mit einer Einschränkung dieser Art gehören `memcpy()` , `strcpy()` , `strcat()` , `sprintf()` und `sscanf()` . Der Standard sagt von diesen und einigen anderen Funktionen:

> Wenn zwischen überlappenden Objekten kopiert wird, ist das Verhalten undefiniert.

Die `memmove()` Funktion ist die Hauptausnahme dieser Regel. Seine Definition gibt an, dass sich die Funktion so verhält, als ob die Quelldaten zuerst in einen temporären Puffer kopiert und dann an die Zieladresse geschrieben wurden. Es gibt keine Ausnahme für überlappende Quell- und Zielregionen und auch keine Notwendigkeit. `memmove()` hat `memmove()` in solchen Fällen ein genau definiertes Verhalten.

Die Unterscheidung spiegelt eine Effizienz *vs.* allgemeiner Kompromiss. Kopieren, wie diese Funktionen ausgeführt werden, tritt normalerweise zwischen getrennten Speicherbereichen auf, und es ist oft möglich, zur Entwicklungszeit zu wissen, ob eine bestimmte Instanz des Speicherkopierens in dieser Kategorie liegt. Die Annahme, dass keine Überlappung vorliegt, führt zu vergleichsweise effizienteren Implementierungen, die nicht zuverlässig korrekte Ergebnisse liefern, wenn die Annahme nicht zutrifft. Die meisten C-Bibliotheksfunktionen sind für die effizienteren Implementierungen zugelassen, und `memmove()` füllt die Lücken aus und dient den Fällen, in denen sich Quelle und Ziel möglicherweise überschneiden. Um jedoch in allen Fällen die richtige Wirkung zu erzielen, muss es zusätzliche Tests durchführen und / oder eine vergleichsweise weniger effiziente Implementierung einsetzen.

Lesen eines nicht initialisierten Objekts, das nicht durch den Speicher gesichert wird

C11

Das Lesen eines Objekts führt zu undefiniertem Verhalten, wenn das Objekt [1] ist :

- nicht initialisiert
- definiert mit automatischer Speicherdauer
- Seine Adresse wird nie vergeben

Die Variable a im folgenden Beispiel erfüllt alle diese Bedingungen:

```
void Function( void )
{
    int a;
    int b = a;
}
```

---

[1] (Zitiert aus: ISO: IEC 9899: 201X 6.3.2.1 Werte, Arrays und Funktionsbezeichner 2)

Wenn der Wert "lvalue" ein Objekt mit automatischer Speicherdauer bezeichnet, das mit der Registerspeicherklasse hätte deklariert werden können (seine Adresse wurde nie verwendet), und dieses Objekt ist nicht initialisiert (nicht mit einem Initialisierer deklariert und wurde vor der Verwendung nicht zugewiesen) ist das Verhalten undefiniert.

Datenrennen

C11

Mit C11 wurde die Unterstützung für mehrere Ausführungsthreads eingeführt, wodurch Datenrennen möglich sind. Ein Programm enthält ein Datenrennen, wenn von zwei verschiedenen Threads auf ein Objekt [1] zugegriffen wird, wobei mindestens einer der Zugriffe nicht atomar ist, mindestens einer das Objekt ändert und die Semantik des Programms nicht gewährleistet, dass sich die beiden Zugriffe nicht überlappen zeitlich. [2] Beachten Sie, dass die tatsächliche Parallelität der beteiligten Zugriffe keine Bedingung für ein Datenrennen ist. Datenrennen decken eine breitere Klasse von Problemen ab, die sich aus (zulässigen) Inkonsistenzen in den Speicheransichten verschiedener Threads ergeben.

Betrachten Sie dieses Beispiel:

```
#include <threads.h>

int a = 0;

int Function( void* ignore )
{
    a = 1;

    return 0;
}

int main( void )
{
    thrd_t id;
    thrd_create( &id , Function , NULL );

    int b = a;

    thrd_join( id , NULL );
}
```

Der Haupt-Thread ruft `thrd_create` auf, um eine neue Thread-Funktion `Function` zu starten. Der zweite Thread ändert `a` und der Haupt-Thread liest `a` . Keiner dieser Zugriffe ist atomar, und die beiden Threads tun weder einzeln noch gemeinsam, um sicherzustellen, dass sie sich nicht überlappen, sodass es zu einem Datenrennen kommt.

Dieses Programm könnte den Datenwettlauf vermeiden

- der Haupt - Thread könnte seinen Lesevorgang auszuführen `a` vor dem anderen Thread beginnen;
- Der Haupt-Thread könnte ein Lesen von `a` nachdem er via `thrd_join` sichergestellt `thrd_join` dass der andere beendet wurde.

- Die Threads konnten ihre Zugriffe über einen Mutex synchronisieren, wobei jeder diesen Mutex sperrte, bevor er auf `a` zugreift und ihn anschließend entsperrt.

Wie die Mutex-Option zeigt, muss beim Vermeiden eines Datenrennens nicht eine bestimmte Reihenfolge von Vorgängen sichergestellt werden, z. B. wenn der untergeordnete Thread `a` ändert, bevor der Hauptthread es liest. es genügt (um ein Datenrennen zu vermeiden), um sicherzustellen, dass für eine gegebene Ausführung ein Zugriff vor dem anderen erfolgt.

---

[1] Objekt ändern oder lesen.

[2] (Zitat aus ISO: IEC 9889: 201x, Abschnitt 5.1.2.4 "Ausführungen mit mehreren Threads und Datenrennen")
Die Ausführung eines Programms enthält ein Datenrennen, wenn es zwei widersprüchliche Aktionen in verschiedenen Threads enthält, von denen mindestens eine nicht atomar ist und keine der beiden vor dem anderen auftritt. Ein solches Datenrennen führt zu undefiniertem Verhalten.

Lese den Wert des freigegebenen Zeigers

Selbst wenn Sie nur den Wert eines Zeigers **lesen,** der freigegeben wurde (dh ohne den Zeiger dereferenzieren zu wollen), ist undefiniertes Verhalten (UB), z

```
char *p = malloc(5);
free(p);
if (p == NULL) /* NOTE: even without dereferencing, this may have UB */
{

}
```

Zitieren von **ISO / IEC 9899: 2011** , Abschnitt 6.2.4 §2:

> […] Der Wert eines Zeigers wird unbestimmt, wenn das Objekt, auf das er zeigt (oder gerade vorbei ist), das Ende seiner Lebensdauer erreicht.

Die Verwendung von unbestimmtem Speicher für irgendetwas, einschließlich scheinbar harmlosen Vergleichs oder Arithmetik, kann undefiniertes Verhalten aufweisen, wenn der Wert eine Trap-Darstellung für den Typ sein kann.

Ändern Sie das String-Literal

In diesem Codebeispiel wird der Zeichenzeiger `p` auf die Adresse eines Zeichenfolgenlitals initialisiert. Der Versuch, das String-Literal zu ändern, hat ein undefiniertes Verhalten.

```
char *p = "hello world";
p[0] = 'H'; // Undefined behavior
```

Das direkte Ändern eines veränderlichen Arrays von `char` oder durch einen Zeiger ist natürlich kein undefiniertes Verhalten, auch wenn der Initialisierer eine Literalzeichenfolge ist. Folgendes ist in Ordnung:

```
char a[] = "hello, world";
char *p = a;

a[0] = 'H';
p[7] = 'W';
```

Das liegt daran, dass das String-Literal bei jeder Initialisierung des Arrays effektiv in das Array kopiert wird (einmal für Variablen mit statischer Dauer, jedes Mal, wenn das Array für Variablen mit automatischer oder Thread-Dauer erstellt wird - Variablen mit zugewiesener Dauer werden nicht initialisiert) und Es ist in Ordnung, den Inhalt des Arrays zu ändern.

Speicherplatz zweimal freigeben

Das doppelte Freigeben von Speicher ist undefiniertes Verhalten, z

```
int * x = malloc(sizeof(int));
*x = 9;
free(x);
free(x);
```

Zitat aus Standard (7.20.3.2. Die freie Funktion von C99):

> Andernfalls ist das Verhalten undefiniert, wenn das Argument nicht mit einem Zeiger übereinstimmt, der zuvor von der Funktion calloc, malloc oder realloc zurückgegeben wurde.

Verwendung eines falschen Formatbezeichners in printf

Die Verwendung eines falschen Formatbezeichners im ersten Argument für `printf` ruft ein undefiniertes Verhalten auf. Der folgende Code ruft beispielsweise ein undefiniertes Verhalten auf:

```
long z = 'B';
printf("%c\n", z);
```

Hier ist ein anderes Beispiel

```
printf("%f\n",0);
```

Über der Codezeile ist undefiniertes Verhalten. `%f` erwartet ein Doppel. 0 ist jedoch vom Typ `int` .

Beachten Sie, dass Ihr Compiler normalerweise dazu beitragen kann, Fälle wie diese zu vermeiden, wenn Sie während des Kompilierens die entsprechenden Flags `-Wformat` (`-Wformat` in `clang` und `gcc`). Aus dem letzten Beispiel:

```
warning: format specifies type 'double' but the argument has type
      'int' [-Wformat]
    printf("%f\n",0);
            ~~    ^
            %d
```

Die Konvertierung zwischen Zeigertypen führt zu einem falsch ausgerichteten Ergebnis

Die folgende *möglicherweise* undefinierten Verhalten durch falsche Zeiger Ausrichtung:

```
char *memory_block = calloc(sizeof(uint32_t) + 1, 1);
uint32_t *intptr = (uint32_t*)(memory_block + 1);  /* possible undefined behavior */
uint32_t mvalue = *intptr;
```

Das undefinierte Verhalten tritt auf, wenn der Zeiger konvertiert wird. Wenn gemäß C11 eine *Konvertierung zwischen zwei Zeigertypen zu einem Ergebnis führt, das falsch ausgerichtet ist (6.3.2.3), ist das Verhalten undefiniert* . Hier kann ein `uint32_t` eine Ausrichtung von 2 oder 4 erfordern.

`calloc` hingegen muss einen Zeiger zurückgeben, der für jeden Objekttyp geeignet ausgerichtet ist. Der `memory_block` ist also richtig ausgerichtet, um einen `uint32_t` in seinem Anfangsteil zu enthalten. In einem System, in dem `uint32_t` eine Ausrichtung von 2 oder 4 erfordert, ist `memory_block + 1` eine *ungerade* Adresse und daher nicht richtig ausgerichtet.

Beachten Sie, dass der C-Standard fordert, dass bereits der Cast-Vorgang undefiniert ist. Dies ist `memory_block + 1` , da auf Plattformen, auf denen Adressen segmentiert sind, die `memory_block + 1` möglicherweise nicht einmal eine korrekte Darstellung als Ganzzahlzeiger hat.

Das Umwandeln von `char *` in Zeiger auf andere Typen ohne Rücksicht auf die Ausrichtungsanforderungen wird manchmal fälschlicherweise zum Dekodieren gepackter Strukturen wie Dateiheader oder Netzwerkpakete verwendet.

Sie können das undefinierte Verhalten vermeiden, das durch eine falsch ausgerichtete Zeigerkonvertierung entsteht, indem Sie `memcpy` :

```
memcpy(&mvalue, memory_block + 1, sizeof mvalue);
```

Hier findet keine Zeigerumwandlung nach `uint32_t*` statt und die Bytes werden nacheinander kopiert.

Dieser Kopiervorgang für unser Beispiel führt nur zu einem gültigen Wert von `mvalue` weil:

- Wir haben `calloc` , damit die Bytes ordnungsgemäß initialisiert werden. In unserem Fall haben alle Bytes den Wert `0` , aber jede andere ordnungsgemäße Initialisierung würde dies tun.
- `uint32_t` ist ein exakter Breitentyp und hat keine Füllbits
- Jedes beliebige Bitmuster ist eine gültige Darstellung für jeden vorzeichenlosen Typ.

Addition oder Subtraktion des Zeigers nicht richtig begrenzt

Der folgende Code hat ein undefiniertes Verhalten:

```
char buffer[6] = "hello";
```

```
char *ptr1 = buffer - 1;  /* undefined behavior */
char *ptr2 = buffer + 5;  /* OK, pointing to the '\0' inside the array */
char *ptr3 = buffer + 6;  /* OK, pointing to just beyond */
char *ptr4 = buffer + 7;  /* undefined behavior */
```

Laut C11 ist das Verhalten undefiniert, wenn die Addition oder Subtraktion eines Zeigers in ein Array-Objekt oder einen Integer-Typ oder etwas darüber hinaus zu einem Ergebnis führt, das nicht auf das gleiche Array-Objekt oder knapp darüber hinausweist (6.5.6).

Außerdem ist es natürlich undefiniertes Verhalten, einen Zeiger zu *demeferenzieren* , der direkt hinter das Array zeigt:

```
char buffer[6] = "hello";
char *ptr3 = buffer + 6;  /* OK, pointing to just beyond */
char value = *ptr3;       /* undefined behavior */
```

Eine const-Variable mit einem Zeiger ändern

```
int main (void)
{
    const int foo_readonly = 10;
    int *foo_ptr;

    foo_ptr = (int *)&foo_readonly; /* (1) This casts away the const qualifier */
    *foo_ptr = 20; /* This is undefined behavior */

    return 0;
}
```

Zitieren von *ISO / IEC 9899: 201x* , Abschnitt 6.7.3, §2:

> Wenn versucht wird, ein mit einem const-qualifiziertem Typ definiertes Objekt durch Verwendung eines lvalue mit einem nicht const-qualifizierten Typ zu ändern, ist das Verhalten nicht definiert. [...]

---

(1) In GCC kann dies die folgende Warnung `warning: assignment discards 'const' qualifier from pointer target type [-Wdiscarded-qualifiers]` : `warning: assignment discards 'const' qualifier from pointer target type [-Wdiscarded-qualifiers]`

Übergabe eines Nullzeigers an die Konvertierung von printf% s

Die `%s` Konvertierung von `printf` besagt, dass das entsprechende Argument *ein Zeiger auf das Anfangselement eines Arrays vom Zeichentyp ist* . Ein Nullzeiger zeigt nicht auf das Anfangselement eines Arrays von Zeichentypen, und daher ist das Verhalten des Folgenden nicht definiert:

```
char *foo = NULL;
printf("%s", foo); /* undefined behavior */
```

Das undefinierte Verhalten bedeutet jedoch nicht immer, dass das Programm abstürzt - einige

Systeme ergreifen Schritte, um Abstürze zu vermeiden, die normalerweise auftreten, wenn ein Nullzeiger dereferenziert wird. Zum Beispiel ist bekannt, dass Glibc druckt

```
(null)
```

für den Code oben. Fügen Sie jedoch (nur) einen Zeilenumbruch zum Formatstring hinzu, und Sie erhalten einen Absturz:

```
char *foo = 0;
printf("%s\n", foo); /* undefined behavior */
```

In diesem Fall geschieht dies, weil GCC über eine Optimierung verfügt, die `printf("%s\n", argument);` in einen Aufruf an `puts` mit `puts(argument)` , und `puts` in Glibc nicht behandelt Null - Zeiger. All dieses Verhalten ist standardkonform.

Beachten Sie, dass sich der *Nullzeiger* von einer *leeren Zeichenfolge unterscheidet* . Das Folgende ist also gültig und hat kein undefiniertes Verhalten. Es wird nur eine *neue Zeile gedruckt* :

```
char *foo = "";
printf("%s\n", foo);
```

Inkonsistente Verknüpfung von Bezeichnern

```
extern int var;
static int var; /* Undefined behaviour */
```

In C11, § 6.2.2, 7 heißt es:

> Wenn innerhalb einer Übersetzungseinheit derselbe Identifizierer sowohl mit interner als auch mit externer Verknüpfung erscheint, ist das Verhalten nicht definiert.

Wenn eine vorherige Deklaration eines Bezeichners sichtbar ist, wird die Verknüpfung der vorherigen Deklaration angezeigt. *C11, §6.2.2, 4* erlaubt es:

> Für einen mit dem Speicherklassenspezifizierer extern deklarierten Identifizierer in einem Bereich, in dem eine vorherige Deklaration dieses Identifizierers sichtbar ist, 31) Wenn die vorherige Deklaration interne oder externe Verknüpfung angibt, ist die Verknüpfung des Identifizierers bei der späteren Deklaration die gleiche wie bei die auf der vorherigen Erklärung angegebene Verbindung. Wenn keine vorherige Deklaration sichtbar ist oder wenn in der vorherigen Deklaration keine Verknüpfung angegeben ist, hat der Identifier eine externe Verknüpfung.

```
/* 1. This is NOT undefined */
static int var;
extern int var;


/* 2. This is NOT undefined */
```

```
static int var;
static int var;

/* 3. This is NOT undefined */
extern int var;
extern int var;
```

Fflush für einen Eingabestrom verwenden

In den `fflush` und C-Standards wird explizit angegeben, dass die Verwendung von `fflush` in einem Eingabestrom undefiniertes Verhalten ist. Die `fflush` ist nur für Ausgabeströme definiert.

```
#include <stdio.h>

int main()
{
    int i;
    char input[4096];

    scanf("%i", &i);
    fflush(stdin); // <-- undefined behavior
    gets(input);

    return 0;
}
```

Es gibt keine Standardmethode, um ungelesene Zeichen aus einem Eingabestrom zu löschen. Auf der anderen Seite verwenden einige Implementierungen `fflush` , um den `stdin` Puffer zu löschen. Microsoft definiert das Verhalten von `fflush` in einem Eingabestrom: Wenn der Stream für die Eingabe `fflush` löscht `fflush` den Inhalt des Puffers. Gemäß POSIX.1-2008 ist das Verhalten von `fflush` definiert, es sei denn, die Eingabedatei kann gesucht werden.

Weitere `fflush(stdin)` .

Bitverschiebung mit negativen Zählwerten oder über die Breite des Typs hinaus

Wenn die *Verschiebung Zählwert* einen **negativen Wert** ist dann sowohl *Linksverschiebung* und *rechte Shift* - Operationen sind nicht definiert [1]:

```
int x = 5 << -3; /* undefined */
int x = 5 >> -3; /* undefined */
```

Wenn die *Linksverschiebung* bei einem **negativen Wert ausgeführt wird** , ist dies undefiniert:

```
int x = -5 << 3; /* undefined */
```

Wenn ein **positiver Wert nach** *links verschoben* wird und das Ergebnis des mathematischen Werts **nicht** im Typ dargestellt werden kann, ist es undefiniert [1] :

```
/* Assuming an int is 32-bits wide, the value '5 * 2^72' doesn't fit
```

```
 * in an int. So, this is undefined. */

int x = 5 << 72;
```

Beachten Sie, dass die *Rechtsverschiebung* bei einem **negativen Wert** (eg `-5 >> 3`) *nicht* undefiniert, sondern *implementierungsdefiniert ist* .

---

[1] Zitat von *ISO / IEC 9899: 201x* , Abschnitt 6.5.7:

> Wenn der Wert des rechten Operanden negativ ist oder größer oder gleich der Breite des beförderten linken Operanden ist, ist das Verhalten undefiniert.

Ändern der Zeichenfolge, die von den Funktionen getenv, strerror und setlocale zurückgegeben wird

Das Ändern der Zeichenfolgen, die von den Standardfunktionen `getenv()` , `strerror()` und `setlocale()` ist undefiniert. Implementierungen verwenden daher möglicherweise statischen Speicher für diese Zeichenfolgen.

Die Funktion getenv (), C11, §7.22.4.7, 4 sagt:

> Die Funktion getenv gibt einen Zeiger auf eine Zeichenfolge zurück, die dem übereinstimmenden Listenmitglied zugeordnet ist. Die Zeichenfolge, auf die verwiesen wird, darf vom Programm nicht geändert werden, kann jedoch durch einen nachfolgenden Aufruf der Funktion getenv überschrieben werden.

Die Funktion strerror (), C11, §7.23.6.3, 4 sagt:

> Die Strerror-Funktion gibt einen Zeiger auf den String zurück, dessen Inhalt localespeci fi c ist. Das Array, auf das gezeigt wird, darf vom Programm nicht geändert werden, kann jedoch durch einen nachfolgenden Aufruf der Strerror-Funktion überschrieben werden.

Die Funktion setlocale (), C11, §7.11.1.1, 8 sagt:

> Der Zeiger auf die Zeichenfolge, die von der Funktion setlocale zurückgegeben wird, ist so, dass ein nachfolgender Aufruf mit diesem Zeichenfolgenwert und der zugehörigen Kategorie diesen Teil des Gebietsschemas des Programms wiederherstellt. Die Zeichenfolge, auf die verwiesen wird, darf vom Programm nicht geändert werden, kann jedoch durch einen nachfolgenden Aufruf der Funktion setlocale überschrieben werden.

Ebenso gibt die Funktion `localeconv()` einen Zeiger auf `struct lconv` der nicht geändert werden soll.

Die Funktion localeconv (), C11, §7.11.2.1, 8 sagt:

> Die Funktion localeconv gibt einen Zeiger auf das ausgefüllte Objekt zurück. Die

Struktur, auf die der Rückgabewert zeigt, wird vom Programm nicht geändert, kann jedoch durch einen nachfolgenden Aufruf der localeconv-Funktion überschrieben werden.

Rückkehr von einer Funktion, die mit dem _ _Noreturn`- oder `Noreturn'-Funktionsbezeichner deklariert wurde

C11

Der Funktionsbezeichner `_Noreturn` wurde in C11 eingeführt. Die Kopfzeile `<stdnoreturn.h>` enthält ein Makro `noreturn` das auf `_Noreturn` erweitert `_Noreturn` . Die Verwendung von `_Noreturn` oder `noreturn` von `<stdnoreturn.h>` ist also in Ordnung und gleichwertig.

Eine mit `_Noreturn` (oder `noreturn`) deklarierte Funktion darf nicht zu ihrem Aufrufer zurückkehren. Wenn eine solche Funktion an seinen Aufrufer zurückkehrt, ist das Verhalten nicht definiert.

Im folgenden Beispiel wird `func()` mit dem `noreturn` deklariert, kehrt jedoch zu seinem Aufrufer zurück.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>

noreturn void func(void);

void func(void)
{
    printf("In func()...\n");
} /* Undefined behavior as func() returns */

int main(void)
{
    func();
    return 0;
}
```

`gcc` und `clang` erzeugen Warnungen für das obige Programm:

```
$ gcc test.c
test.c: In function ‘func’:
test.c:9:1: warning: ‘noreturn’ function does return
 }
 ^
$ clang test.c
test.c:9:1: warning: function declared 'noreturn' should not return [-Winvalid-noreturn]
}
^
```

Ein Beispiel mit `noreturn` , das ein genau definiertes Verhalten aufweist:

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <stdnoreturn.h>

noreturn void my_exit(void);

/* calls exit() and doesn't return to its caller. */
void my_exit(void)
{
    printf("Exiting...\n");
    exit(0);
}

int main(void)
{
    my_exit();
    return 0;
}
```

Undefiniertes Verhalten online lesen: https://riptutorial.com/de/c/topic/364/undefiniertes-verhalten

Kapitel 54: Valgrind

Syntax

- valgrind *Programmname optionale Argumente* < *Testeingabe*

Bemerkungen

Valgrind ist ein Debugging-Tool, mit dem Fehler in Bezug auf die Speicherverwaltung in C-Programmen diagnostiziert werden können. Valgrind kann verwendet werden, um Fehler wie die Verwendung eines ungültigen Zeigers zu erkennen, z. B. das Schreiben oder Lesen über den zugewiesenen Speicherplatz hinaus oder einen ungültigen Aufruf von `free()` . Es kann auch zur Verbesserung von Anwendungen durch Funktionen verwendet werden, die die Speicherprofilerstellung durchführen.

Weitere Informationen finden Sie unter http://valgrind.org .

Examples

Valgrind laufen lassen

```
valgrind ./my-program arg1 arg2 < test-input
```

Dadurch wird Ihr Programm ausgeführt und ein Bericht über allfällige Zuweisungen und Abgaben erstellt. Außerdem werden Sie vor häufigen Fehlern gewarnt, z. B. mit nicht initialisiertem Speicher, der Deferenzierung von Zeigern auf fremde Stellen, dem Abschreiben des mit malloc zugewiesenen Blocks oder der fehlgeschlagenen Freigabe der Blöcke.

Flaggen hinzufügen

Sie können auch weitere Tests aktivieren, z.

```
valgrind -q --tool=memcheck --leak-check=yes ./my-program arg1 arg2 < test-input
```

Weitere Informationen zu den (vielen) Optionen finden Sie unter valgrind --help. Ausführliche Informationen zur Bedeutung der Ausgabe finden Sie in der Dokumentation unter http://valgrind.org .

Bytes verloren - vergessen vergessen

Hier ist ein Programm, das malloc aber nicht frei aufruft:

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, char **argv)
{
    char *s;

    s = malloc(26); // the culprint

    return 0;
}
```

Ohne zusätzliche Argumente sucht valgrind nicht nach diesem Fehler.

Wenn Sie jedoch `--leak-check=yes` oder `--tool=memcheck` , werden die Zeilen für diese Speicherverluste beschwert und angezeigt, wenn das Programm im Debug-Modus kompiliert wurde:

```
$ valgrind -q --leak-check=yes ./missing_free
==4776== 26 bytes in 1 blocks are definitely lost in loss record 1 of 1
==4776==    at 0x4024F20: malloc (vg_replace_malloc.c:236)
==4776==    by 0x80483F8: main (missing_free.c:9)
==4776==
```

Wenn das Programm nicht im Debug-Modus kompiliert wird (beispielsweise mit dem Flag `-g` in GCC), zeigt es uns immer noch an, wo das Leck in Bezug auf die betreffende Funktion aufgetreten ist, nicht jedoch die Zeilen.

Dadurch können wir zurückgehen und sehen, welcher Block in dieser Zeile zugewiesen wurde, und versuchen, vorwärts zu suchen, um zu sehen, warum er nicht freigegeben wurde.

Die häufigsten Fehler bei der Verwendung von Valgrind

Valgrind liefert Ihnen die *Zeilen, in denen der Fehler* am Ende jeder Zeile im Format `(file.c:line_no)` . Fehler in valgrind werden folgendermaßen zusammengefasst:

```
ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Zu den häufigsten Fehlern gehören:

1. **Illegale Lese- / Schreibfehler**

```
==8451== Invalid read of size 2
==8451==    at 0x4E7381D: getenv (getenv.c:84)
==8451==    by 0x4EB1559: __libc_message (libc_fatal.c:80)
==8451==    by 0x4F5256B: __fortify_fail (fortify_fail.c:37)
==8451==    by 0x4F5250F: __stack_chk_fail (stack_chk_fail.c:28)
==8451==    by 0x40059C: main (valg.c:10)
==8451==  Address 0x700000007 is not stack'd, malloc'd or (recently) free'd
```

Dies geschieht, wenn der Code beginnt, auf Speicher zuzugreifen, der nicht zum Programm gehört. Die Größe des Speichers, auf den zugegriffen wird, zeigt auch an, welche Variable verwendet wurde.

2. **Verwendung von nicht initialisierten Variablen**

```
==8795== 1 errors in context 5 of 8:
==8795== Conditional jump or move depends on uninitialised value(s)
==8795==    at 0x4E881AF: vfprintf (vfprintf.c:1631)
==8795==    by 0x4E8F898: printf (printf.c:33)
==8795==    by 0x400548: main (valg.c:7)
```

Entsprechend dem Fehler hat der Aufruf von `printf()` in Zeile 7 des `main` von `valg.c` eine nicht initialisierte Variable an `printf` .

3. **Illegale Befreiung der Erinnerung**

```
==8954== Invalid free() / delete / delete[] / realloc()
==8954==    at 0x4C2EDEB: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8954==    by 0x4005A8: main (valg.c:10)
==8954==  Address 0x5203040 is 0 bytes inside a block of size 240 free'd
==8954==    at 0x4C2EDEB: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8954==    by 0x40059C: main (valg.c:9)
==8954==  Block was alloc'd at
==8954==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8954==    by 0x40058C: main (valg.c:7)
```

Laut valgrind hat der Code den Speicher illegal (ein zweites Mal) in *Zeile 10* von `valg.c` , wohingegen er bereits in *Zeile 9* freigegeben wurde und der Block selbst in *Zeile 7* Speicher zugewiesen wurde.

Valgrind online lesen: https://riptutorial.com/de/c/topic/2674/valgrind

Kapitel 55: Variable Argumente

Einführung

Variablenargumente werden von Funktionen der printf-Familie (`printf` , `fprintf` usw.) und anderen verwendet, um zu ermöglichen, dass eine Funktion jedes Mal mit einer anderen Anzahl von Argumenten aufgerufen wird, daher der Name *varargs* .

Um Funktionen mithilfe der Variablenargumente zu implementieren, verwenden Sie `#include <stdarg.h>` .

Um Funktionen mit einer variablen Anzahl von Argumenten aufzurufen, stellen Sie sicher, dass es einen vollständigen Prototyp mit nachlaufenden Ellipsen im Gültigkeitsbereich gibt: `void err_exit(const char *format, ...);` zum Beispiel.

Syntax

- void va_start (va_list ap, *last*); / * Variadic Argumentverarbeitung starten; *last* ist der letzte Funktionsparameter vor den Ellipsen ("...") * /
- *Typ* va_arg (va_list ap, *type);* / * Nächstes variadisches Argument in der Liste erhalten; Vergewissern Sie sich, dass Sie den richtigen *beworbenen* Typ angeben * /
- void va_end (va_list ap); / * Argumentverarbeitung beenden * /
- va_copy ungültig (va_list dst, va_list src); / * C99 oder höher: Argumentliste, dh aktuelle Position in der Argumentverarbeitung, in eine andere Liste kopieren (zB um Argumente mehrfach zu übergeben) * /

Parameter

Parameter	Einzelheiten
`va_list` ap	Argumentzeiger, aktuelle Position in der Liste der variadischen Argumente
zuletzt	Name des letzten nicht-variadischen Funktionsarguments, damit der Compiler den richtigen Ort für die Verarbeitung von variadischen Argumenten findet; darf nicht als `register` , Funktion oder Array-Typ deklariert werden
Art	**gefördert** Typ des Arguments variadische (zB zum Lesen `int` für ein `short int` Argument)
`va_list` src	aktueller zu kopierender Argumentzeiger
`va_list` dst	neue Argumentliste, die ausgefüllt werden soll

Bemerkungen

Die Funktionen `va_start` , `va_arg` , `va_end` und `va_copy` sind eigentlich Makros.

`va_start` Sie sicher, dass Sie `va_start` *immer* zuerst und nur einmal aufrufen und `va_end` zuletzt und nur einmal sowie an jedem Exit-Punkt der Funktion aufrufen. Wenn Sie dies nicht tun, funktioniert dies *möglicherweise* auf *Ihrem* System, ist aber sicherlich **nicht** portierbar und führt daher zu Fehlern.

Achten Sie darauf, Ihre Funktion korrekt zu deklarieren, dh mit einem Prototyp, und beachten Sie die Einschränkungen für das *letzte* nicht-variadische Argument (nicht `register` , keine Funktion oder einen Array-Typ). Es ist nicht möglich, eine Funktion zu deklarieren, die nur variadische Argumente verwendet, da mindestens ein nicht-variadisches Argument erforderlich ist, um die Argumentverarbeitung starten zu können.

Wenn Sie `va_arg` aufrufen, müssen Sie den **beförderten** Argumenttyp anfordern, d. `va_arg`

- `short` wird zu `int` (und `unsigned short` wird auch zu `int` befördert, es sei denn `sizeof(unsigned short) == sizeof(int)` . In diesem Fall wird es zu `unsigned int` .
- `float` wird `double` .
- `signed char` wird zu `int` ; `unsigned char` wird ebenfalls zu `int` befördert, es sei denn `sizeof(unsigned char) == sizeof(int)` , was selten der Fall ist.
- `char` wird normalerweise zu `int` .
- C99-Typen wie `uint8_t` oder `int16_t` werden ähnlich befördert.

Die Verarbeitung historischer (dh K & R) variadischer Argumente wird in `<varargs.h>` , sollte aber nicht verwendet werden, da sie veraltet ist. Die Verarbeitung variabischer Standardargumente (die hier beschriebene und in `<stdarg.h>`) wurde in C89 eingeführt. `va_copy` Makro `va_copy` wurde in C99 eingeführt, wurde jedoch zuvor von vielen Compilern bereitgestellt.

Examples

Verwenden eines expliziten count-Arguments, um die Länge der va_list zu bestimmen

Bei jeder variadischen Funktion muss die Funktion wissen, wie sie die Liste der Variablenargumente interpretiert. Bei den Funktionen `printf()` oder `scanf()` teilt die `scanf()` der Funktion mit, was zu erwarten ist.

Die einfachste Technik besteht darin, eine explizite Anzahl der anderen Argumente zu übergeben (die normalerweise alle vom gleichen Typ sind). Dies wird in der variadischen Funktion im folgenden Code demonstriert, der die Summe einer Reihe von Ganzzahlen berechnet, wobei eine beliebige Anzahl von Ganzzahlen vorhanden sein kann, die Anzahl jedoch als Argument vor der Liste der Variablenargumente angegeben wird.

```
#include <stdio.h>
#include <stdarg.h>

/* first arg is the number of following int args to sum. */
int sum(int n, ...) {
```

```
    int sum = 0;
    va_list it; /* hold information about the variadic argument list. */

    va_start(it, n); /* start variadic argument processing */
    while (n--)
      sum += va_arg(it, int); /* get and sum the next variadic argument */
    va_end(it); /* end variadic argument processing */

    return sum;
}

int main(void)
{
    printf("%d\n", sum(5, 1, 2, 3, 4, 5)); /* prints 15 */
    printf("%d\n", sum(10, 5, 9, 2, 5, 111, 6666, 42, 1, 43, -6218)); /* prints 666 */
    return 0;
}
```

Terminatorwerte verwenden, um das Ende von va_list zu bestimmen

Bei jeder variadischen Funktion muss die Funktion wissen, wie sie die Liste der Variablenargumente interpretiert. Der "traditionelle" Ansatz (am Beispiel von `printf`) besteht darin, die Anzahl der Argumente im Vorfeld anzugeben. Dies ist jedoch nicht immer eine gute Idee:

```
/* First argument specifies the number of parameters; the remainder are also int */
extern int sum(int n, ...);

/* But it's far from obvious from the code. */
sum(5, 2, 1, 4, 3, 6)

/* What happens if i.e. one argument is removed later on? */
sum(5, 2, 1, 3, 6)  /* Disaster */
```

Manchmal ist es robuster, einen expliziten Terminator hinzuzufügen, wie beispielsweise die POSIX-Funktion `execlp()` . Hier ist eine weitere Funktion zum Berechnen der Summe einer Reihe von `double` Zahlen:

```
#include <stdarg.h>
#include <stdio.h>
#include <math.h>

/* Sums args up until the terminator NAN */
double sum (double x, ...) {
  double sum = 0;
  va_list va;

  va_start(va, x);
  for (; !isnan(x); x = va_arg(va, double)) {
    sum += x;
  }
  va_end(va);

  return sum;
}

int main (void) {
```

```
  printf("%g\n", sum(5., 2., 1., 4., 3., 6., NAN));
  printf("%g\n", sum(1, 0.5, 0.25, 0.125, 0.0625, 0.03125, NAN));
}
```

Gute Terminatorwerte:

- Ganzzahl (soll alle positiv oder nicht negativ sein) - `0` oder `-1`
- Fließkommatypen - `NAN`
- Zeigertypen - `NULL`
- Aufzählertypen - ein besonderer Wert

Funktionen mit einer `printf ()` -ähnlichen Oberfläche implementieren

Argumentlisten mit variabler Länge werden häufig verwendet, um Funktionen zu implementieren, die einen dünnen Wrapper um die `printf()` Funktionsfamilie bilden. Ein solches Beispiel ist eine Reihe von Fehlerberichterstattungsfunktionen.

errmsg.h

```
#ifndef ERRMSG_H_INCLUDED
#define ERRMSG_H_INCLUDED

#include <stdarg.h>
#include <stdnoreturn.h>    // C11

void verrmsg(int errnum, const char *fmt, va_list ap);
noreturn void errmsg(int exitcode, int errnum, const char *fmt, ...);
void warnmsg(int errnum, const char *fmt, ...);

#endif
```

Dies ist ein nacktes Beispiel. Solche Pakete können sehr aufwendig sein. Normalerweise verwenden Programmierer entweder `errmsg()` oder `warnmsg()` , die selbst intern `verrmsg()` . Wenn jedoch jemand die Notwendigkeit hat, mehr zu tun, ist die exponierte `verrmsg()` Funktion nützlich. Sie könnten aussetzt es vermeiden , bis Sie eine Notwendigkeit für sie (YAGNI - Yagni), aber die Notwendigkeit entstehen schließlich (Sie *sind* gonna brauchen es - YAGNI).

errmsg.c

Dieser Code muss nur die variadischen Argumente an die Funktion `vfprintf()` um sie in einen Standardfehler `vfprintf()` . Sie meldet auch die Systemfehlermeldung, die der an die Funktionen übergebenen Systemfehlernummer (`errno`) entspricht.

```
#include "errmsg.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void
verrmsg(int errnum, const char *fmt, va_list ap)
{
    if (fmt)
        vfprintf(stderr, fmt, ap);
```

```
    if (errnum != 0)
        fprintf(stderr, ": %s", strerror(errnum));
    putc('\n', stderr);
}

void
errmsg(int exitcode, int errnum, const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    verrmsg(errnum, fmt, ap);
    va_end(ap);
    exit(exitcode);
}

void
warnmsg(int errnum, const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    verrmsg(errnum, fmt, ap);
    va_end(ap);
}
```

Mit `errmsg.h`

Jetzt können Sie diese Funktionen wie folgt verwenden:

```
#include "errmsg.h"
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    char buffer[BUFSIZ];
    int fd;
    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s filename\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    const char *filename = argv[1];

    if ((fd = open(filename, O_RDONLY)) == -1)
        errmsg(EXIT_FAILURE, errno, "cannot open %s", filename);
    if (read(fd, buffer, sizeof(buffer)) != sizeof(buffer))
        errmsg(EXIT_FAILURE, errno, "cannot read %zu bytes from %s", sizeof(buffer),
filename);
    if (close(fd) == -1)
        warnmsg(errno, "cannot close %s", filename);
    /* continue the program */
    return 0;
}
```

Wenn entweder der Systemaufruf `open()` oder `read()` fehlschlägt, wird der Fehler in den Standardfehler geschrieben und das Programm wird mit dem Exit-Code 1 beendet. Wenn der

Systemaufruf `close()` fehlschlägt, wird der Fehler lediglich als Warnmeldung ausgegeben und Das Programm wird fortgesetzt.

Überprüfen der korrekten Verwendung von `printf()` -Formaten

Wenn Sie GCC (den GNU C-Compiler, der Teil der GNU-Compiler-Collection ist) oder Clang verwenden, können Sie den Compiler überprüfen lassen, ob die Argumente, die Sie an die Fehlermeldungsfunktionen übergeben, mit dem übereinstimmen, was `printf()` erwartet. Da nicht alle Compiler die Erweiterung unterstützen, muss sie bedingt kompiliert werden, was etwas fummelig ist. Der Schutz lohnt sich jedoch.

Zunächst müssen wir wissen, wie der Compiler GCC oder Clang ist, der GCC emuliert. Die Antwort ist, dass GCC `__GNUC__` definiert, um `__GNUC__` anzuzeigen.

Informationen zu den Attributen finden Sie unter Allgemeine Funktionsattribute - insbesondere das `format` .

`errmsg.h` umgeschrieben

```
#ifndef ERRMSG_H_INCLUDED
#define ERRMSG_H_INCLUDED

#include <stdarg.h>
#include <stdnoreturn.h>    // C11

#if !defined(PRINTFLIKE)
#if defined(__GNUC__)
#define PRINTFLIKE(n,m) __attribute__((format(printf,n,m)))
#else
#define PRINTFLIKE(n,m) /* If only */
#endif /* __GNUC__ */
#endif /* PRINTFLIKE */

void verrmsg(int errnum, const char *fmt, va_list ap);
void noreturn errmsg(int exitcode, int errnum, const char *fmt, ...)
        PRINTFLIKE(3, 4);
void warnmsg(int errnum, const char *fmt, ...)
        PRINTFLIKE(2, 3);

#endif
```

Nun, wenn Sie einen Fehler machen wie:

```
errmsg(EXIT_FAILURE, errno, "Failed to open file '%d' for reading", filename);
```

(Wenn `%d` `%s`), wird der Compiler Folgendes bemängeln:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes \
>     -Wold-style-definition -c erruse.c
erruse.c: In function ‘main’:
erruse.c:20:64: error: format ‘%d’ expects argument of type ‘int’, but argument 4 has type
‘const char *’ [-Werror=format=]
         errmsg(EXIT_FAILURE, errno, "Failed to open file '%d' for reading", filename);
                                                                  ~^
```

```
                                                            %s
cc1: all warnings being treated as errors
$
```

Verwenden einer Formatzeichenfolge

Die Verwendung einer Formatzeichenfolge enthält Informationen über die erwartete Anzahl und den Typ der nachfolgenden variadischen Argumente, sodass ein explizites count-Argument oder ein Terminatorwert nicht erforderlich ist.

Das folgende Beispiel zeigt eine Funktion, die die Standardfunktion `printf()` umschließt und nur die Verwendung von variadischen Argumenten des Typs `char` , `int` und `double` (im dezimalen Fließkommaformat) zulässt. Wie bei `printf()` ist hier das erste Argument der Wrapping-Funktion der Formatstring. Da die Formatzeichenfolge analysiert wird, kann die Funktion feststellen, ob ein anderes variadisches Argument erwartet wird und welchen Typ es haben soll.

```
#include <stdio.h>
#include <stdarg.h>

int simple_printf(const char *format, ...)
{
    va_list ap; /* hold information about the variadic argument list. */
    int printed = 0; /* count of printed characters */

    va_start(ap, format); /* start variadic argument processing */

    while (*format != '\0') /* read format string until string terminator */
    {
        int f = 0;

        if (*format == '%')
        {
            ++format;
            switch(*format)
            {
                case 'c' :
                    f = printf("%d", va_arg(ap, int)); /* print next variadic argument, note
type promotion from char to int */
                    break;
                case 'd' :
                    f = printf("%d", va_arg(ap, int)); /* print next variadic argument */
                    break;

                case 'f' :
                    f = printf("%f", va_arg(ap, double)); /* print next variadic argument */
                    break;
                default :
                    f = -1; /* invalid format specifier */
                    break;
            }
        }
        else
        {
            f = printf("%c", *format); /* print any other characters */
        }

        if (f < 0) /* check for errors */
```

```
        {
            printed = f;
            break;
        }
        else
        {
            printed += f;
        }
        ++format; /* move on to next character in string */
    }

    va_end(ap); /* end variadic argument processing */

    return printed;
}

int main (int argc, char *argv[])
{
    int x = 40;
    int y = 0;

    y = simple_printf("There are %d characters in this sentence", x);
    simple_printf("\n%d were printed\n", y);
}
```

Variable Argumente online lesen: https://riptutorial.com/de/c/topic/455/variable-argumente

Kapitel 56: Verknüpfte Listen

Bemerkungen

Die C-Sprache definiert keine Datenstruktur für verknüpfte Listen. Wenn Sie C verwenden und eine verkettete Liste benötigen, müssen Sie entweder eine verkettete Liste aus einer vorhandenen Bibliothek (z. B. GLib) verwenden oder eine eigene verkettete Listenschnittstelle schreiben. In diesem Thema werden Beispiele für verknüpfte Listen und doppelte verknüpfte Listen beschrieben, die als Ausgangspunkt für das Schreiben Ihrer eigenen verknüpften Listen verwendet werden können.

Einfach verknüpfte Liste

Die Liste enthält Knoten, die aus einem Link bestehen, der als Nächstes bezeichnet wird.

Datenstruktur

```
struct singly_node
{
  struct singly_node * next;
};
```

Doppelt verknüpfte Liste

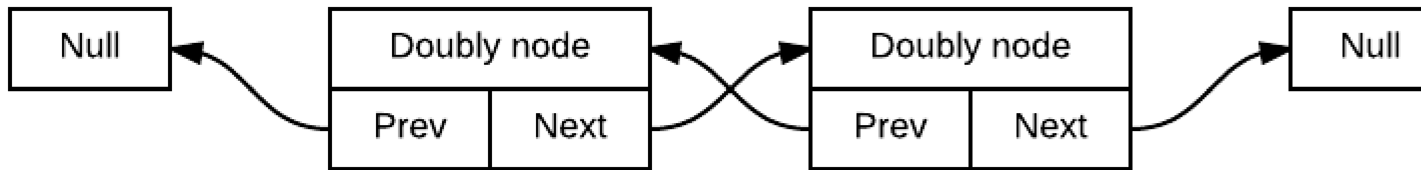
Die Liste enthält Knoten, die aus zwei Verknüpfungen bestehen, die als Vorherige und Nächste bezeichnet werden. Die Links verweisen normalerweise auf einen Knoten mit derselben Struktur.

Datenstruktur

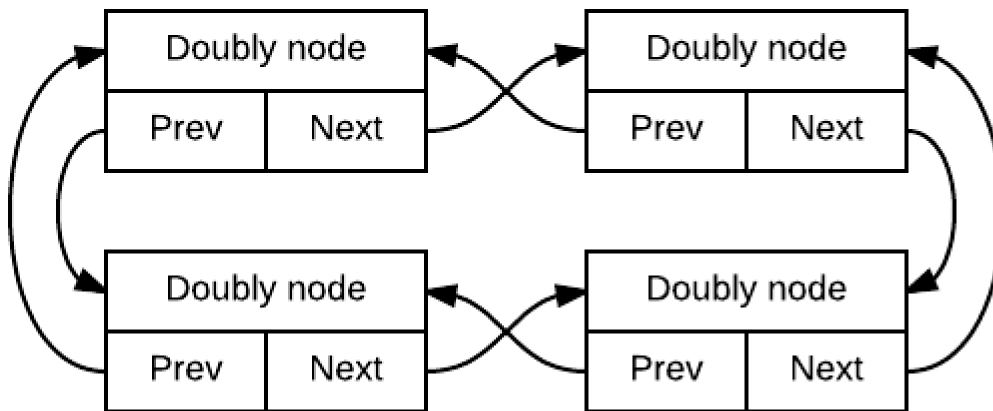
```
struct doubly_node
{
  struct doubly_node * prev;
  struct doubly_node * next;
};
```

Topoliges

Linear oder offen



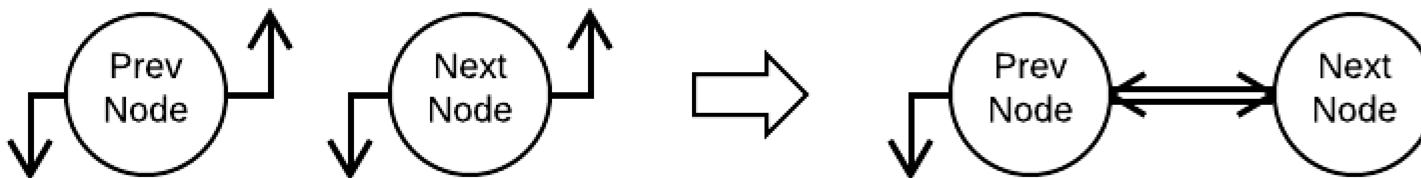
Kreisförmig oder ring



Verfahren

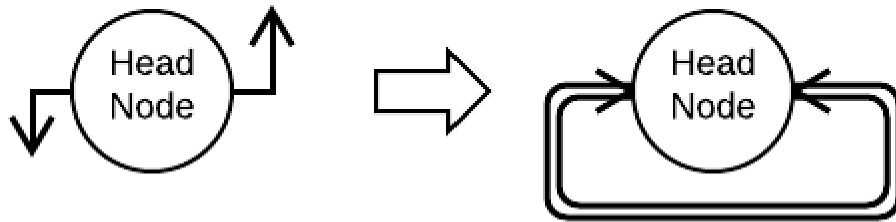
Binden

Binden Sie zwei Knoten miteinander.



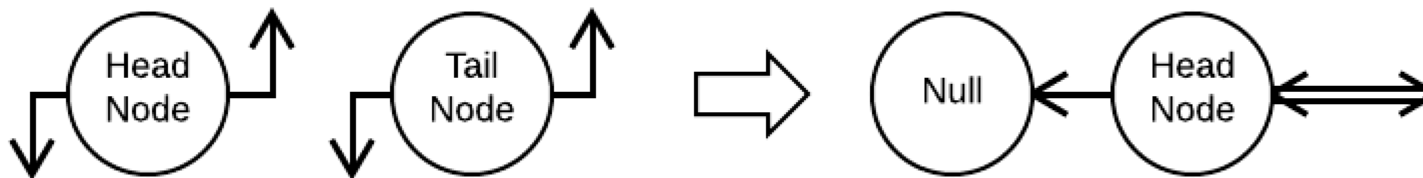

```
void doubly_node_bind (struct doubly_node * prev, struct doubly_node * next)
{
  prev->next = next;
  next->prev = prev;
}
```

Erstellen einer kreisförmig verbundenen Liste




```
void doubly_node_make_empty_circularly_list (struct doubly_node * head)
{
  doubly_node_bind (head, head);
}
```

Erstellen einer linear verknüpften Liste




```
void doubly_node_make_empty_linear_list (struct doubly_node * head, struct doubly_node * tail)
{
  head->prev = NULL;
  tail->next = NULL;
  doubly_node_bind (head, tail);
}
```

Einfügung

Nehmen wir an, eine leere Liste enthält immer einen Knoten anstelle von NULL. Dann müssen Einfügeprozeduren nicht NULL berücksichtigen.

```
void doubly_node_insert_between
(struct doubly_node * prev, struct doubly_node * next, struct doubly_node * insertion)
{
  doubly_node_bind (prev, insertion);
  doubly_node_bind (insertion, next);
}

void doubly_node_insert_before
(struct doubly_node * tail, struct doubly_node * insertion)
{
  doubly_node_insert_between (tail->prev, tail, insertion);
}

void doubly_node_insert_after
```

```
(struct doubly_node * head, struct doubly_node * insertion)
{
  doubly_node_insert_between (head, head->next, insertion);
}
```

Examples

Einfügen eines Knotens am Anfang einer einfach verknüpften Liste

Der nachstehende Code fordert Sie zur Eingabe von Nummern auf und fügt sie am Anfang einer verknüpften Liste hinzu.

```
/* This program will demonstrate inserting a node at the beginning of a linked list */

#include <stdio.h>
#include <stdlib.h>

struct Node {
  int data;
  struct Node* next;
};


void insert_node (struct Node **head, int nodeValue);
void print_list (struct Node *head);

int main(int argc, char *argv[]) {
  struct Node* headNode;
  headNode = NULL; /* Initialize our first node pointer to be NULL. */
  size_t listSize, i;
  do {
    printf("How many numbers would you like to input?\n");
  } while(1 != scanf("%zu", &listSize));

  for (i = 0; i < listSize; i++) {
    int numToAdd;
    do {
      printf("Enter a number:\n");
    } while (1 != scanf("%d", &numToAdd));

    insert_node (&headNode, numToAdd);
    printf("Current list after your inserted node: \n");
    print_list(headNode);
  }

  return 0;
}

void print_list (struct Node *head) {
  struct node* currentNode = head;

  /* Iterate through each link. */
  while (currentNode != NULL) {
      printf("Value: %d\n", currentNode->data);
      currentNode = currentNode -> next;
  }
}
```

```
void insert_node (struct Node **head, int nodeValue) {
  struct Node *currentNode = malloc(sizeof *currentNode);
  currentNode->data = nodeValue;
  currentNode->next = (*head);

  *head = currentNode;
}
```

Erläuterung zum Einfügen von Knoten

Um zu verstehen, wie wir am Anfang Knoten hinzufügen, werfen wir einen Blick auf mögliche Szenarien:

1. Die Liste ist leer, daher müssen wir einen neuen Knoten hinzufügen. In diesem Fall sieht unser Gedächtnis folgendermaßen aus, wobei `HEAD` ein Zeiger auf den ersten Knoten ist:

```
| HEAD | --> NULL
```

Die Zeile `currentNode->next = *headNode;` weist den Wert von `currentNode->next` zu `NULL` da `headNode` ursprünglich mit einem `NULL` Wert beginnt.

Jetzt möchten wir unseren Kopfknotenzeiger so einstellen, dass er auf unseren aktuellen Knoten zeigt.

```
  -----      -------------
 |HEAD | --> |CURRENTNODE| --> NULL /* The head node points to the current node */
  -----      -------------
```

Dies geschieht mit `*headNode = currentNode;`

2. Die Liste ist bereits ausgefüllt. Wir müssen am Anfang einen neuen Knoten hinzufügen. Der Einfachheit halber beginnen wir mit 1 Knoten:

```
 -----    -----------
 HEAD --> FIRST NODE --> NULL
 -----    -----------
```

Bei `currentNode->next = *headNode` sieht die Datenstruktur folgendermaßen aus:

```
 ---------        -----    ---------------------
 currentNode -->  HEAD --> POINTER TO FIRST NODE --> NULL
 ---------        -----    ---------------------
```

Was natürlich geändert werden muss, da `*headNode` auf `currentNode` `*headNode` sollte.

```
 ----    -----------    ---------------
 HEAD -> currentNode -->     NODE       -> NULL
 ----    -----------    ---------------
```

Dies geschieht mit `*headNode = currentNode;`

Einfügen eines Knotens an der n-ten Position

Bisher haben wir uns mit dem Einfügen eines Knotens am Anfang einer einfach verknüpften Liste befasst . Meistens möchten Sie jedoch auch Knoten an anderer Stelle einfügen können. Der folgende Code zeigt, wie es möglich ist, eine `insert()` Funktion zu schreiben, um Knoten an *beliebigen Stellen* in die verknüpften Listen einzufügen.

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
  int data;
  struct Node* next;
};

struct Node* insert(struct Node* head, int value, size_t position);
void print_list (struct Node* head);

int main(int argc, char *argv[]) {
  struct Node *head = NULL; /* Initialize the list to be empty */

  /* Insert nodes at positions with values: */
  head = insert(head, 1, 0);
  head = insert(head, 100, 1);
  head = insert(head, 21, 2);
  head = insert(head, 2, 3);
  head = insert(head, 5, 4);
  head = insert(head, 42, 2);

  print_list(head);
  return 0;
}

struct Node* insert(struct Node* head, int value, size_t position) {
  size_t i = 0;
  struct Node *currentNode;

  /* Create our node */
  currentNode = malloc(sizeof *currentNode);
  /* Check for success of malloc() here! */

  /* Assign data */
  currentNode->data = value;

  /* Holds a pointer to the 'next' field that we have to link to the new node.
     By initializing it to &head we handle the case of insertion at the beginning. */
  struct Node **nextForPosition = &head;
  /* Iterate to get the 'next' field we are looking for.
     Note: Insert at the end if position is larger than current number of elements. */
  for (i = 0; i < position && *nextForPosition != NULL; i++) {
      /* nextForPosition is pointing to the 'next' field of the node.
         So *nextForPosition is a pointer to the next node.
         Update it with a pointer to the 'next' field of the next node. */
      nextForPosition = &(*nextForPosition)->next;
  }
```

```
  /* Here, we are taking the link to the next node (the one our newly inserted node should
  point to) by dereferencing nextForPosition, which points to the 'next' field of the node
  that is in the position we want to insert our node at.
  We assign this link to our next value. */
  currentNode->next = *nextForPosition;

  /* Now, we want to correct the link of the node before the position of our
  new node: it will be changed to be a pointer to our new node. */
  *nextForPosition = currentNode;

  return head;
}

void print_list (struct Node* head) {
  /* Go through the list of nodes and print out the data in each node */
  struct Node* i = head;
  while (i != NULL) {
    printf("%d\n", i->data);
    i = i->next;
  }
}
```

Umkehrung einer verknüpften Liste

Sie können diese Aufgabe auch rekursiv ausführen, aber ich habe mich in diesem Beispiel für einen iterativen Ansatz entschieden. Diese Aufgabe ist hilfreich, wenn Sie alle Knoten am Anfang einer verknüpften Liste einfügen . Hier ist ein Beispiel:

```
#include <stdio.h>
#include <stdlib.h>

#define NUM_ITEMS 10

struct Node {
  int data;
  struct Node *next;
};

void insert_node(struct Node **headNode, int nodeValue, int position);
void print_list(struct Node *headNode);
void reverse_list(struct Node **headNode);


int main(void) {
  int i;
  struct Node *head = NULL;

  for(i = 1; i <= NUM_ITEMS; i++) {
    insert_node(&head, i, i);
  }
  print_list(head);

  printf("I will now reverse the linked list\n");
  reverse_list(&head);
  print_list(head);
  return 0;
}
```

```
void print_list(struct Node *headNode) {
  struct Node *iterator;

  for(iterator = headNode; iterator != NULL; iterator = iterator->next) {
    printf("Value: %d\n", iterator->data);
  }
}

void insert_node(struct Node **headNode, int nodeValue, int position) {
  int i;
  struct Node *currentNode = (struct Node *)malloc(sizeof(struct Node));
  struct Node *nodeBeforePosition = *headNode;

  currentNode->data = nodeValue;

  if(position == 1) {
    currentNode->next = *headNode;
    *headNode = currentNode;
    return;
  }

  for (i = 0; i < position - 2; i++) {
    nodeBeforePosition = nodeBeforePosition->next;
  }

  currentNode->next = nodeBeforePosition->next;
  nodeBeforePosition->next = currentNode;
}

void reverse_list(struct Node **headNode) {
  struct Node *iterator = *headNode;
  struct Node *previousNode = NULL;
  struct Node *nextNode = NULL;

  while (iterator != NULL) {
    nextNode = iterator->next;
    iterator->next = previousNode;
    previousNode = iterator;
    iterator = nextNode;
  }

  /* Iterator will be NULL by the end, so the last node will be stored in
  previousNode.  We will set the last node to be the headNode */
  *headNode = previousNode;
}
```

Erklärung für die Reverse List-Methode

Wir starten den `previousNode` als `NULL` , da wir bei der ersten Iteration der Schleife wissen, dass wir `NULL` suchen, wenn wir vor dem ersten Kopfknoten nach dem Knoten suchen. Der erste Knoten wird der letzte Knoten in der Liste, und die nächste Variable sollte natürlich `NULL` .

Grundsätzlich besteht das Konzept der Umkehrung der verknüpften Liste darin, dass wir die Verknüpfungen selbst umkehren. Das nächste Member jedes Knotens wird der Knoten davor, wie folgt:

```
Head -> 1 -> 2 -> 3 -> 4 -> 5
```

Jede Zahl steht für einen Knoten. Diese Liste würde zu:

```
1 <- 2 <- 3 <- 4 <- 5 <- Head
```

Schließlich sollte der Kopf stattdessen auf den fünften Knoten zeigen, und jeder Knoten sollte auf den vor ihm liegenden Knoten zeigen.

Knoten 1 sollte auf `NULL` da vorher nichts vorlag. Knoten 2 sollte auf Knoten 1 zeigen, Knoten 3 auf Knoten 2 und so weiter.

Es gibt jedoch *ein kleines Problem* mit dieser Methode. Wenn wir die Verbindung zum nächsten Knoten aufheben und zum vorherigen Knoten ändern, können wir nicht zum nächsten Knoten in der Liste wechseln, da die Verbindung zu diesem Knoten weg ist.

Die Lösung für dieses Problem besteht darin, das nächste Element in einer Variablen (`nextNode`) zu `nextNode` bevor Sie die Verknüpfung ändern.

Eine doppelt verknüpfte Liste

Ein Codebeispiel, das zeigt, wie Knoten in eine doppelt verknüpfte Liste eingefügt werden können, wie die Liste leicht umgekehrt werden kann und wie sie umgekehrt gedruckt werden kann.

```
#include <stdio.h>
#include <stdlib.h>

/* This data is not always stored in a structure, but it is sometimes for ease of use */
struct Node {
  /* Sometimes a key is also stored and used in the functions */
  int data;
  struct Node* next;
  struct Node* previous;
};

void insert_at_beginning(struct Node **pheadNode, int value);
void insert_at_end(struct Node **pheadNode, int value);

void print_list(struct Node *headNode);
void print_list_backwards(struct Node *headNode);

void free_list(struct Node *headNode);

int main(void) {
  /* Sometimes in a doubly linked list the last node is also stored */
  struct Node *head = NULL;

  printf("Insert a node at the beginning of the list.\n");
  insert_at_beginning(&head, 5);
  print_list(head);

  printf("Insert a node at the beginning, and then print the list backwards\n");
  insert_at_beginning(&head, 10);
  print_list_backwards(head);

  printf("Insert a node at the end, and then print the list forwards.\n");
```

```
  insert_at_end(&head, 15);
  print_list(head);

  free_list(head);

  return 0;
}

void print_list_backwards(struct Node *headNode) {
  if (NULL == headNode)
  {
    return;
  }
  /*
  Iterate through the list, and once we get to the end, iterate backwards to print
  out the items in reverse order (this is done with the pointer to the previous node).
  This can be done even more easily if a pointer to the last node is stored.
  */
  struct Node *i = headNode;
  while (i->next != NULL) {
    i = i->next; /* Move to the end of the list */
  }

  while (i != NULL) {
    printf("Value: %d\n", i->data);
    i = i->previous;
  }
}

void print_list(struct Node *headNode) {
  /* Iterate through the list and print out the data member of each node */
  struct Node *i;
  for (i = headNode; i != NULL; i = i->next) {
    printf("Value: %d\n", i->data);
  }
}

void insert_at_beginning(struct Node **pheadNode, int value) {
  struct Node *currentNode;

  if (NULL == pheadNode)
  {
    return;
  }
  /*
  This is done similarly to how we insert a node at the beginning of a singly linked
  list, instead we set the previous member of the structure as well
  */
  currentNode = malloc(sizeof *currentNode);

  currentNode->next = NULL;
  currentNode->previous = NULL;
  currentNode->data = value;

  if (*pheadNode == NULL) { /* The list is empty */
    *pheadNode = currentNode;
    return;
  }

  currentNode->next = *pheadNode;
  (*pheadNode)->previous = currentNode;
```

```
  *pheadNode = currentNode;
}

void insert_at_end(struct Node **pheadNode, int value) {
  struct Node *currentNode;

  if (NULL == pheadNode)
  {
    return;
  }

  /*
  This can, again be done easily by being able to have the previous element.  It
  would also be even more useful to have a pointer to the last node, which is commonly
  used.
  */

  currentNode = malloc(sizeof *currentNode);
  struct Node *i = *pheadNode;

  currentNode->data = value;
  currentNode->next = NULL;
  currentNode->previous = NULL;

  if (*pheadNode == NULL) {
    *pheadNode = currentNode;
    return;
  }

  while (i->next != NULL) { /* Go to the end of the list */
    i = i->next;
  }

  i->next = currentNode;
  currentNode->previous = i;
}

void free_list(struct Node *node) {
  while (node != NULL) {
    struct Node *next = node->next;
    free(node);
    node = next;
  }
}
```

Beachten Sie, dass das Speichern eines Zeigers auf den letzten Knoten manchmal nützlich ist (es ist effizienter, einfach direkt zum Ende der Liste springen zu können, als bis zum Ende durchlaufen zu müssen):

```
struct Node *lastNode = NULL;
```

In diesem Fall ist eine Aktualisierung bei Änderungen der Liste erforderlich.

Manchmal wird ein Schlüssel auch zur Identifizierung von Elementen verwendet. Es ist einfach ein Mitglied der Node-Struktur:

```
struct Node {
```

```
  int data;
  int key;
  struct Node* next;
  struct Node* previous;
};
```

Der Schlüssel wird dann verwendet, wenn Aufgaben für ein bestimmtes Element ausgeführt werden, z. B. das Löschen von Elementen.

Verknüpfte Listen online lesen: https://riptutorial.com/de/c/topic/560/verknupfte-listen

Kapitel 57: X-Makros

Einführung

X-Makros sind eine auf dem Präprozessor basierende Technik zum Minimieren von Wiederholungscode und zum Verwalten von Daten- / Code-Entsprechungen. Mehrere verschiedene Makroerweiterungen, die auf einem gemeinsamen Datensatz basieren, werden unterstützt, indem die gesamte Erweiterungsgruppe über ein einziges Master-Makro dargestellt wird. Der Ersetzungstext dieses Makros besteht aus einer Folge von Erweiterungen eines inneren Makros, einer für jedes Datum. Das innere Makro wird traditionell `X()` , daher der Name der Technik.

Bemerkungen

Es wird erwartet, dass der Benutzer eines X-Makro-Master-Makros seine eigene Definition für das innere `X()` -Makro bereitstellt und in seinem Bereich das Master-Makro erweitert. Die inneren Makroreferenzen des Masters werden entsprechend der Definition des Benutzers von `X()` . Auf diese Weise kann die Menge des sich wiederholenden Boilerplate-Codes in der Quelldatei reduziert werden (erscheint nur einmal im Ersetzungstext von `X()`), wie es von Anhängern der "Do not Repeat Yourself" (DRY) Philosophie bevorzugt wird.

Darüber hinaus können X-Makros durch das erneute Definieren von `X()` und Erweitern des Master-Makros ein oder mehrere zusätzliche Male die Pflege entsprechender Daten und Code erleichtern - eine Erweiterung des Makros deklariert die Daten (beispielsweise als Array-Elemente oder Enum-Member) und Die anderen Erweiterungen erzeugen entsprechenden Code.

Obwohl der Name "X-Makro" vom traditionellen Namen des inneren Makros stammt, hängt die Technik nicht von diesem Namen ab. Jeder gültige Makroname kann an seiner Stelle verwendet werden.

Kritikpunkte umfassen

- Quelldateien, die X-Makros verwenden, sind schwieriger zu lesen.
- Wie alle Makros sind X-Makros streng textlich - sie bieten von Natur aus keine Typensicherheit. und
- X-Makros ermöglichen die *Codegenerierung* . Im Vergleich zu auf Aufruffunktionen basierenden Alternativen wird der Code durch X-Makros effektiv vergrößert.

Eine gute Erklärung für X-Makros finden Sie in Randy Meyers 'Artikel [X-Macros] in Dr. Dobbs (http://www.drdobbs.com/the-new-cx-macros/184401387) .

Examples

Trivialer Einsatz von X-Makros für printfs

```
/* define a list of preprocessor tokens on which to call X */
#define X_123 X(1) X(2) X(3)

/* define X to use */
#define X(val) printf("X(%d) made this print\n", val);
X_123
#undef X
/* good practice to undef X to facilitate reuse later on */
```

In diesem Beispiel generiert der Präprozessor den folgenden Code:

```
printf("X(%d) made this print\n", 1);
printf("X(%d) made this print\n", 2);
printf("X(%d) made this print\n", 3);
```

Aufzählungswert und Bezeichner

```
/* declare items of the enum */
#define FOREACH \
      X(item1) \
      X(item2) \
      X(item3) \
/* end of list */

/* define the enum values */
#define X(id) MyEnum_ ## id,
enum MyEnum { FOREACH };
#undef X

/* convert an enum value to its identifier */
const char * enum2string(int enumValue)
{
    const char* stringValue = NULL;
#define X(id) if (enumValue == MyEnum_ ## id) stringValue = #id;
    FOREACH
#undef X
    return stringValue;
}
```

Als Nächstes können Sie den Aufzählungswert in Ihrem Code verwenden und den Bezeichner auf folgende Weise drucken:

```
printf("%s\n", enum2string(MyEnum_item2));
```

Erweiterung: Geben Sie das X-Makro als Argument an

Der X-Makro-Ansatz kann etwas verallgemeinert werden, indem der Name des Makros "X" zu einem Argument des Master-Makros gemacht wird. Dies hat den Vorteil, dass Sie Makronamenskollisionen vermeiden und die Verwendung eines Allzweckmakros als "X" -Makro zulassen.

Wie bei X-Makros stellt das Master-Makro eine Liste von Elementen dar, deren Bedeutung für dieses Makro spezifisch ist. In dieser Variante kann ein solches Makro folgendermaßen definiert

werden:

```
/* declare list of items */
#define ITEM_LIST(X) \
      X(item1) \
      X(item2) \
      X(item3) \
/* end of list */
```

Man könnte dann Code generieren, um die Artikelnamen wie folgt auszudrucken:

```
/* define macro to apply */
#define PRINTSTRING(value) printf( #value "\n");

/* apply macro to the list of items */
ITEM_LIST(PRINTSTRING)
```

Das erweitert sich zu diesem Code:

```
printf( "item1" "\n"); printf( "item2" "\n"); printf( "item3" "\n");
```

Im Gegensatz zu Standard-X-Makros, bei denen der Name "X" ein integriertes Merkmal des Master-Makros ist, kann es bei diesem Stil unnötig oder sogar unerwünscht sein, das als Argument verwendete Makro (in diesem Beispiel `PRINTSTRING`) `PRINTSTRING` .

Codegenerierung

X-Makros können zur Codegenerierung verwendet werden, indem wiederholter Code geschrieben wird: Durchlaufen Sie eine Liste, um bestimmte Aufgaben auszuführen oder eine Reihe von Konstanten, Objekten oder Funktionen zu deklarieren.

Hier verwenden wir X-Makros, um eine Aufzählung mit 4 Befehlen und eine Zuordnung ihrer Namen als Zeichenfolgen zu deklarieren

Dann können wir die Zeichenfolgewerte der Aufzählung drucken.

```
/* All our commands */
#define COMMANDS(OP) OP(Open) OP(Close) OP(Save) OP(Quit)

/* generate the enum Commands: {cmdOpen, cmdClose, cmdSave, cmdQuit, }; */
#define ENUM_NAME(name) cmd##name,
enum Commands {
  COMMANDS(ENUM_NAME)
};
#undef ENUM_NAME

/* generate the string table */
#define COMMAND_OP(name) #name,
const char* const commandNames[] = {
```

```
  COMMANDS(COMMAND_OP)
};
#undef COMMAND_OP

/* the following prints "Quit\n": */
printf("%s\n", commandNames[cmdQuit]());
```

In ähnlicher Weise können wir eine Sprungtabelle erzeugen, um Funktionen anhand des Aufzählungswerts aufzurufen.

Dies erfordert, dass alle Funktionen dieselbe Signatur haben. Wenn sie keine Argumente akzeptieren und ein int zurückgeben, würden wir dies in einen Header mit der enum-Definition einfügen:

```
/* declare all functions as extern */
#define EXTERN_FUNC(name) extern int doCmd##name(void);
COMMANDS(EXTERN_FUNC)
#undef EXTERN_FUNC

/* declare the function pointer type and the jump table  */
typedef int (*CommandFunc)(void);
extern CommandFunc commandJumpTable[];
```

Alle folgenden Elemente können sich in verschiedenen Übersetzungseinheiten befinden, vorausgesetzt, der obige Teil ist als Kopf enthalten:

```
/* generate the jump table */
#define FUNC_NAME(name) doCmd##name,
CommandFunc commandJumpTable[] = {
  COMMANDS(FUNC_NAME)
};
#undef FUNC_NAME

/* call the save command like this: */
int result = commandJumpTable[cmdSave]();

/* somewhere else, we need the implementations of the commands */
int doCmdOpen(void) {/* code performing open command */}
int doCmdClose(void) {/* code performing close command */}
int doCmdSave(void) {/* code performing save command */}
int doCmdQuit(void) {/* code performing quit command */}
```

Ein Beispiel für diese Technik, die in echtem Code verwendet wird, ist die GPU-Befehlsverteilung in Chromium .

X-Makros online lesen: https://riptutorial.com/de/c/topic/628/x-makros

Kapitel 58: Zeichenfolge mit mehreren Zeichen

Bemerkungen

Nicht alle Präprozessoren unterstützen die Verarbeitung von Trigrafesequenzen. Einige Compiler geben eine zusätzliche Option oder einen Schalter für die Verarbeitung an. Andere verwenden ein separates Programm, um Trigrafien zu konvertieren.

Der GCC Compiler erkennt sie nicht , wenn Sie es explizit anfordern so (Verwendung tun `-trigraphs` ihnen zu ermöglichen, verwenden `-Wtrigraphs` , Teil `-Wall` , Warnungen über trigraphs zu bekommen).

Da die meisten heute verwendeten Plattformen den gesamten Bereich der in C verwendeten Einzelzeichen unterstützen, werden Digraphen gegenüber Trigraphen bevorzugt, aber die Verwendung beliebiger Zeichenfolgen mit mehreren Zeichen wird im Allgemeinen nicht empfohlen.

Hüten Sie sich auch vor versehentlichem Trigraph-Gebrauch (zB `puts("What happened??!!");`

Examples

Trigraphen

Die Symbole `[ ] { } ^ \ | ~ #` werden häufig in C-Programmen verwendet. In den späten 1980er Jahren wurden jedoch Codesätze verwendet (ISO 646-Varianten, z. B. in skandinavischen Ländern), bei denen die ASCII-Zeichenpositionen für landessprachige Varianten (z. B. `£` für `#` in Großbritannien; `Æ Å æ å ø Ø` für `{ } { } | \` in Dänemark; in EBCDIC gab es kein `~`). Dies bedeutete, dass es schwierig war, C-Code auf Maschinen zu schreiben, die diese Sets verwendeten.

Um dieses Problem zu lösen, schlug der C-Standard die Verwendung von Kombinationen von drei Zeichen vor, um ein einzelnes Zeichen zu erzeugen, das als Trigraph bezeichnet wird. Ein Trigraph ist eine Folge von drei Zeichen, von denen die ersten beiden Fragezeichen sind.

Das folgende ist ein einfaches Beispiel, das Trigrafesequenzen anstelle von `#` , `{` und `}` :

```
??=include <stdio.h>

int main()
??<
    printf("Hello World!\n");
??>
```

Dies wird vom C-Präprozessor geändert, indem die Trigraphen durch ihre Einzelzeichen-Äquivalente ersetzt werden, als ob der Code geschrieben worden wäre:

```
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
}
```

<table>
<tr><th>Trigraph</th><th>Äquivalent</th></tr>
<tr><td>?? =</td><td>#</td></tr>
<tr><td>??</td><td>\</td></tr>
<tr><td>?? '</td><td>^</td></tr>
<tr><td>?? (</td><td>[</td></tr>
<tr><td>??)</td><td>]</td></tr>
<tr><td>??!</td><td>|</td></tr>
<tr><td>?? <</td><td>{</td></tr>
<tr><td>??></td><td>}</td></tr>
<tr><td>?? -</td><td>~</td></tr>
</table>

Beachten Sie, dass Trigraphen problematisch sind, weil z. B. `??/` ein Backslash ist und die Bedeutung von Fortsetzungszeilen in Kommentaren beeinflussen kann und in Zeichenfolgen und Zeichenliteralen erkannt werden muss (z. B. `'??/??/'` ist eine einzige Zeichen, ein Backslash).

Digraphen

C99

1994 wurden besser lesbare Alternativen zu fünf der Trigraphen geliefert. Diese verwenden nur zwei Zeichen und werden als Digraphen bezeichnet. Im Gegensatz zu Trigraphen sind Digraphen Marken. Wenn ein Digraph in einem anderen Token vorkommt (z. B. Zeichenfolgenliterale oder Zeichenkonstanten), wird er nicht als Digraph behandelt, sondern bleibt unverändert.

Das Folgende zeigt den Unterschied vor und nach der Verarbeitung der Digraphenfolge.

```
#include <stdio.h>

int main()
<%
    printf("Hello %> World!\n"); /* Note that the string contains a digraph */
%>
```

Die werden gleich behandelt wie:

```
#include <stdio.h>

int main()
{
    printf("Hello %> World!\n"); /* Note the unchanged digraph within the string. */
}
```

Digraph	Äquivalent
<:	[
:>	]
<%	{
%>	}
%:	#

Zeichenfolge mit mehreren Zeichen online lesen: https://riptutorial.com/de/c/topic/7111/zeichenfolge-mit-mehreren-zeichen

Kapitel 59: Zeichenketten

Einführung

In C ist eine Zeichenfolge kein intrinsischer Typ. Ein C-String ist die Konvention, um ein eindimensionales Array von Zeichen zu haben, das durch ein Nullzeichen und ein `'\0'` .

Dies bedeutet, dass ein C-String mit dem Inhalt `"abc"` die vier Zeichen `'a'` , `'b'` , `'c'` und `'\0'` .

Siehe die grundlegende Einführung zu Strings .

Syntax

- char str1 [] = "Hallo, Welt!"; / * Modifizierbar * /
- char str2 [14] = "Hallo, Welt!"; / * Modifizierbar * /
- char * str3 = "Hallo, Welt!"; /* Nicht veränderbar*/

Examples

Länge berechnen: strlen ()

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
    /* Exit if no second argument is found. */
    if (argc != 2)
    {
        puts("Argument missing.");
        return EXIT_FAILURE;
    }

    size_t len = strlen(argv[1]);
    printf("The length of the second argument is %zu.\n", len);

    return EXIT_SUCCESS;
}
```

Dieses Programm berechnet die Länge seines zweiten Eingabearrements und speichert das Ergebnis in `len` . Diese Länge wird dann an das Terminal ausgegeben. Wenn Sie zum Beispiel mit den Parametern `program_name "Hello, world!"` , Das Programm gibt aus. `The length of the second argument is 13.` Weil die Zeichenfolge `Hello, world!` ist 13 Zeichen lang.

`strlen` zählt alle **Bytes** vom Anfang des Strings bis zum abschließenden NUL-Zeichen `'\0'` , schließt dieses jedoch nicht ein. Daher kann es nur verwendet werden, wenn *garantiert ist* , dass der String NUL-terminiert ist.

`strlen` Sie außerdem, dass `strlen` Ihnen nicht sagt, wie viele Zeichen die Zeichenfolge enthält, wenn die Zeichenfolge Unicode-Zeichen enthält (da einige Zeichen möglicherweise mehrere Byte lang sind). In solchen Fällen müssen Sie die Zeichen (*dh* Codeeinheiten) selbst zählen. Betrachten Sie die Ausgabe des folgenden Beispiels:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char asciiString[50] = "Hello world!";
    char utf8String[50] = "Γειά σου Κόσμε!"; /* "Hello World!" in Greek */

    printf("asciiString has %zu bytes in the array\n", sizeof(asciiString));
    printf("utf8String has %zu bytes in the array\n", sizeof(utf8String));
    printf("\"%s\" is %zu bytes\n", asciiString, strlen(asciiString));
    printf("\"%s\" is %zu bytes\n", utf8String, strlen(utf8String));
}
```

Ausgabe:

```
asciiString has 50 bytes in the array
utf8String has 50 bytes in the array
"Hello world!" is 12 bytes
"Γειά σου Κόσμε!" is 27 bytes
```

Kopieren und Verketten: strcpy (), strcat ()

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  /* Always ensure that your string is large enough to contain the characters
   * and a terminating NUL character ('\0')!
   */
  char mystring[10];

  /* Copy "foo" into `mystring`, until a NUL character is encountered. */
  strcpy(mystring, "foo");
  printf("%s\n", mystring);

  /* At this point, we used 4 chars of `mystring`, the 3 characters of "foo",
   * and the NUL terminating byte.
   */

  /* Append "bar" to `mystring`. */
  strcat(mystring, "bar");
  printf("%s\n", mystring);

  /* We now use 7 characters of `mystring`: "foo" requires 3, "bar" requires 3
   * and there is a terminating NUL character ('\0') at the end.
   */

  /* Copy "bar" into `mystring`, overwriting the former contents. */
```

```
  strcpy(mystring, "bar");
  printf("%s\n", mystring);

  return 0;
}
```

Ausgänge:

```
foo
foobar
bar
```

Wenn Sie an eine vorhandene Zeichenfolge anhängen oder von dieser kopieren oder kopieren, stellen Sie sicher, dass sie NUL-terminiert ist!

String-Literale (zB `"foo"`) werden vom Compiler immer NUL-terminiert.

Vergleich: strcmp (), strncmp (), strcasecmp (), strncasecmp ()

Die `strcase*` -Funktionen sind nicht Standard C, sondern eine POSIX-Erweiterung.

Die Funktion `strcmp` vergleicht lexikographisch zwei nullterminierte Zeichenarrays. Die Funktionen geben einen negativen Wert zurück, wenn das erste Argument vor dem zweiten in lexikografischer Reihenfolge erscheint, Null, wenn sie gleich sind, oder positiv, wenn das erste Argument nach dem zweiten in lexikografischer Reihenfolge erscheint.

```
#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs)
{
    int result = strcmp(lhs, rhs); // compute comparison once
    if (result < 0) {
        printf("%s comes before %s\n", lhs, rhs);
    } else if (result == 0) {
        printf("%s equals %s\n", lhs, rhs);
    } else { // last case: result > 0
        printf("%s comes after %s\n", lhs, rhs);
    }
}

int main(void)
{
    compare("BBB", "BBB");
    compare("BBB", "CCCCC");
    compare("BBB", "AAAAAA");
    return 0;
}
```

Ausgänge:

```
BBB equals BBB
BBB comes before CCCCC
BBB comes after AAAAAA
```

Als `strcmp` vergleicht die `strcasecmp` Funktion auch lexikographisch ihre Argumente, nachdem sie jedes Zeichen in einen Kleinbuchstaben übersetzt haben:

```
#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs)
{
    int result = strcasecmp(lhs, rhs); // compute case-insensitive comparison once
    if (result < 0) {
        printf("%s comes before %s\n", lhs, rhs);
    } else if (result == 0) {
        printf("%s equals %s\n", lhs, rhs);
    } else { // last case: result > 0
        printf("%s comes after %s\n", lhs, rhs);
    }
}

int main(void)
{
    compare("BBB", "bBB");
    compare("BBB", "ccCCC");
    compare("BBB", "aaaaaa");
    return 0;
}
```

Ausgänge:

```
BBB equals bBB
BBB comes before ccCCC
BBB comes after aaaaaa
```

`strncmp` und `strncasecmp` vergleichen höchstens n Zeichen:

```
#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs, int n)
{
    int result = strncmp(lhs, rhs, n); // compute comparison once
    if (result < 0) {
        printf("%s comes before %s\n", lhs, rhs);
    } else if (result == 0) {
        printf("%s equals %s\n", lhs, rhs);
    } else { // last case: result > 0
        printf("%s comes after %s\n", lhs, rhs);
    }
}

int main(void)
{
    compare("BBB", "Bb", 1);
    compare("BBB", "Bb", 2);
    compare("BBB", "Bb", 3);
    return 0;
}
```

Ausgänge:

```
BBB equals Bb
BBB comes before Bb
BBB comes before Bb
```

Tokenisierung: strtok (), strtok_r () und strtok_s ()

Die Funktion `strtok` zerlegt einen String in kleinere Strings oder Token, wobei ein Satz von Trennzeichen verwendet wird.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    int toknum = 0;
    char src[] = "Hello,, world!";
    const char delimiters[] = ", !";
    char *token = strtok(src, delimiters);
    while (token != NULL)
    {
        printf("%d: [%s]\n", ++toknum, token);
        token = strtok(NULL, delimiters);
    }
    /* source is now "Hello\0, world\0\0" */
}
```

Ausgabe:

```
1: [Hello]
2: [world]
```

Die Begrenzungszeichenfolge kann ein oder mehrere Begrenzungszeichen enthalten, und bei jedem Aufruf von `strtok` können unterschiedliche Begrenzungszeichenfolgen verwendet werden.

Aufrufe von `strtok` , um die Tokenisierung derselben `strtok` fortzusetzen, sollten die Quellzeichenfolge nicht erneut übergeben, sondern `NULL` als erstes Argument. Wenn die gleiche Quelle String übergeben *wird* , dann wird das erste Token stattdessen wieder in Token aufgeteilt werden. Das heißt, mit denselben Trennzeichen würde `strtok` das erste Token einfach wieder zurückgeben.

Beachten Sie, dass als `strtok` keine neue Speicher zuteilt für die Token, *die Quellzeichenfolge ändert.* Das heißt, im obigen Beispiel wird der String `src` so manipuliert, dass er die Token erzeugt, auf die der Zeiger verweist, der von den Aufrufen von `strtok` . Das bedeutet, dass die Quellzeichenfolge nicht `const` (also kein String-Literal sein kann). Dies bedeutet auch, dass die Identität des Begrenzungsbytes verloren geht (dh in dem Beispiel werden die Zeichen "," und "!" Effektiv aus dem Quellstring gelöscht und Sie können nicht erkennen, welches Trennzeichen übereinstimmt).

Beachten Sie auch, dass mehrere aufeinanderfolgende Trennzeichen in der Quellzeichenfolge als

eins behandelt werden. In diesem Beispiel wird das zweite Komma ignoriert.

`strtok` ist weder threadsicher noch neu `strtok` da beim Parsing ein statischer Puffer verwendet wird. Das heißt, wenn eine Funktion `strtok` , kann keine Funktion, die sie während der Verwendung von `strtok` `strtok` , auch `strtok` , und sie kann nicht von einer Funktion aufgerufen werden, die selbst `strtok` .

Ein Beispiel, das die Probleme `strtok` die durch die Tatsache verursacht werden, dass `strtok` nicht erneut `strtok` lautet wie folgt:

```
char src[] = "1.2,3.5,4.2";
char *first = strtok(src, ",");

do
{
    char *part;
    /* Nested calls to strtok do not work as desired */
    printf("[%s]\n", first);
    part = strtok(first, ".");
    while (part != NULL)
    {
        printf(" [%s]\n", part);
        part = strtok(NULL, ".");
    }
} while ((first = strtok(NULL, ",")) != NULL);
```

Ausgabe:

```
[1.2]
 [1]
 [2]
```

Die erwartete Operation besteht darin, dass die äußere `do while` Schleife drei Token erstellen sollte, die aus jeder Dezimalzahl-Zeichenfolge (`"1.2"` , `"3.5"` , `"4.2"`) bestehen, für die jeder `strtok` die innere Schleife `strtok` Ziffernfolgen (`"1"` , `"2"` , `"3"` , `"5"` , `"4"` , `"2"`).

Da `strtok` jedoch nicht erneut `strtok` , tritt dies nicht auf. Stattdessen erstellt der erste `strtok` das Token "1.2 \ 0" richtig und die innere Schleife die Token `"1"` und `"2"` . Dann befindet sich der `strtok` in der äußeren Schleife am Ende der Zeichenfolge, die von der inneren Schleife verwendet wird, und gibt NULL sofort zurück. Der zweite und dritte Teilstring des `src` Arrays werden überhaupt nicht analysiert.

C11

Die Standard-C-Bibliotheken enthalten keine Thread-sichere oder wiedereintretende Version, andere jedoch, wie beispielsweise POSIX ' `strtok_r` . Beachten Sie, dass auf MSVC das `strtok` Äquivalent `strtok_s` Thread-sicher ist.

C11

C11 hat einen optionalen Teil, Annex K, der eine thread-sichere und wiedereintretende Version namens `strtok_s` . Sie können mit `__STDC_LIB_EXT1__` auf die Funktion `__STDC_LIB_EXT1__` . Dieser

optionale Teil wird nicht allgemein unterstützt.

Die Funktion `strtok_s` unterscheidet sich von der POSIX-Funktion `strtok_r` durch das `strtok_r` vor dem Speichern außerhalb der Zeichenkette, die mit Token versehen wird, und durch Überprüfung der Laufzeiteinschränkungen. Bei korrekt geschriebenen Programmen verhalten sich `strtok_s` und `strtok_r` gleich.

Wenn Sie `strtok_s` mit dem Beispiel verwenden, erhalten Sie jetzt die richtige Antwort:

```
/* you have to announce that you want to use Annex K */
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>

#ifndef __STDC_LIB_EXT1__
# error "we need strtok_s from Annex K"
#endif

char src[] = "1.2,3.5,4.2";
char *next = NULL;
char *first = strtok_s(src, ",", &next);

do
{
    char *part;
    char *posn;

    printf("[%s]\n", first);
    part = strtok_s(first, ".", &posn);
    while (part != NULL)
    {
        printf(" [%s]\n", part);
        part = strtok_s(NULL, ".", &posn);
    }
}
while ((first = strtok_s(NULL, ",", &next)) != NULL);
```

Und die Ausgabe wird sein:

```
[1.2]
 [1]
 [2]
[3.5]
 [3]
 [5]
[4.2]
 [4]
 [2]
```

Finden Sie das erste / letzte Vorkommen eines bestimmten Zeichens: strchr (), strrchr ()

Die Funktionen `strchr` und `strrchr` finden ein Zeichen in einer Zeichenfolge, das sich in einem NUL-terminierten Zeichenarray befindet. `strchr` einen Zeiger auf das erste Vorkommen zurück und `strrchr` auf das letzte Vorkommen.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char toSearchFor = 'A';

    /* Exit if no second argument is found. */
    if (argc != 2)
    {
        printf("Argument missing.\n");
        return EXIT_FAILURE;
    }

    {
        char *firstOcc = strchr(argv[1], toSearchFor);
        if (firstOcc != NULL)
        {
            printf("First position of %c in %s is %td.\n",
              toSearchFor, argv[1], firstOcc-argv[1]); /* A pointer difference's result
                                       is a signed integer and uses the length modifier 't'. */
        }
        else
        {
            printf("%c is not in %s.\n", toSearchFor, argv[1]);
        }
    }

    {
        char *lastOcc = strrchr(argv[1], toSearchFor);
        if (lastOcc != NULL)
        {
            printf("Last position of %c in %s is %td.\n",
              toSearchFor, argv[1], lastOcc-argv[1]);
        }
    }

    return EXIT_SUCCESS;
}
```

Ausgaben (nachdem eine ausführbare Datei mit dem Namen `pos` generiert wurde):

```
$ ./pos AAAAAAA
First position of A in AAAAAAA is 0.
Last position of A in AAAAAAA is 6.
$ ./pos BAbbbbbAccccAAAAzzz
First position of A in BAbbbbbAccccAAAAzzz is 1.
Last position of A in BAbbbbbAccccAAAAzzz is 15.
$  ./pos qwerty
A is not in qwerty.
```

`strrchr` wird häufig verwendet, um einen Dateinamen aus einem Pfad zu extrahieren. Zum Beispiel zum Extrahieren von `myfile.txt` aus `C:\Users\eak\myfile.txt` :

```
char *getFileName(const char *path)
{
    char *pend;
```

```
    if ((pend = strrchr(path, '\')) != NULL)
        return pend + 1;

    return NULL;
}
```

Iteration über die Zeichen in einer Zeichenfolge

Wenn wir die Länge der Zeichenfolge kennen, können wir eine for-Schleife verwenden, um ihre Zeichen zu durchlaufen:

```
char * string = "hello world"; /* This 11 chars long, excluding the 0-terminator. */
size_t i = 0;
for (; i < 11; i++) {
    printf("%c\n", string[i]);    /* Print each character of the string. */
}
```

Alternativ können wir die Standardfunktion `strlen()` , um die Länge einer Zeichenfolge zu ermitteln, wenn wir nicht wissen, was die Zeichenfolge ist:

```
size_t length = strlen(string);
size_t i = 0;
for (; i < length; i++) {
    printf("%c\n", string[i]);    /* Print each character of the string. */
}
```

Schließlich können wir die Tatsache ausnutzen, dass die Zeichenfolgen in C garantiert null-terminiert sind (was wir bereits bei der Übergabe an `strlen()` im vorherigen Beispiel `strlen()`). Wir können das Array unabhängig von seiner Größe durchlaufen und die Iteration abbrechen, sobald wir ein Null-Zeichen erreicht haben:

```
size_t i = 0;
while (string[i] != '\0') {       /* Stop looping when we reach the null-character. */
    printf("%c\n", string[i]);    /* Print each character of the string. */
    i++;
}
```

Grundlegende Einführung in Strings

In C ist eine **Zeichenfolge** eine Folge von Zeichen, die durch ein Nullzeichen ('\ 0') abgeschlossen wird.

Wir können Strings mit **String-Literalen** erstellen, bei denen es sich um Zeichenfolgen handelt, die von doppelten Anführungszeichen umgeben sind. Nehmen Sie zum Beispiel den String-Literal `"hello world"` . String-Literale werden automatisch mit Nullen abgeschlossen.

Wir können Strings mit verschiedenen Methoden erstellen. Zum Beispiel können wir ein `char *` deklarieren und es so initialisieren, dass es auf das erste Zeichen einer Zeichenfolge zeigt:

```
char * string = "hello world";
```

Beim Initialisieren eines `char *` mit einer String-Konstante wie oben wird der String selbst normalerweise in schreibgeschützten Daten zugewiesen. `string` ist ein Zeiger auf das erste Element des Arrays, das das Zeichen `'h'` .

Da das String-Literal im Nur-Lese-Speicher zugewiesen wird, kann es nicht geändert werden [1] . Jeder Versuch, es zu ändern, führt zu undefiniertem Verhalten. Es ist daher besser, `const` hinzuzufügen, um einen Kompilierungsfehler wie diesen zu erhalten

```
char const * string = "hello world";
```

Es hat einen ähnlichen Effekt [2] wie

```
char const string_arr[] = "hello world";
```

Um eine modifizierbare Zeichenfolge zu erstellen, können Sie ein Zeichenarray deklarieren und dessen Inhalt mit einem Zeichenfolgenliteral wie folgt initialisieren:

```
char modifiable_string[] = "hello world";
```

Dies entspricht dem Folgenden:

```
char modifiable_string[] = {'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\0'};
```

Da die zweite Version einen geschweiften Klammerinitialisierer verwendet, wird die Zeichenfolge nicht automatisch mit einem Nullwert abgeschlossen, es sei denn, ein `'\0'` Zeichen ist explizit im Zeichenarray enthalten, normalerweise als letztes Element.

---

1 Nicht modifizierbar bedeutet , dass die Zeichen im String können nicht geändert werden, aber denken Sie daran , dass der Zeiger `string` geändert werden kann (kann woanders Punkt oder kann erhöht oder verringert werden).

2 Beide Zeichenfolgen haben den gleichen Effekt, dass Zeichen beider Zeichenfolgen nicht geändert werden können. Es ist zu beachten, dass `string` ein Zeiger auf `char` und ein modifizierbarer l-Wert ist. Er kann also inkrementiert werden oder auf eine andere Position zeigen, während das Array `string_arr` ein nicht modifizierbarer l-Wert ist.

Erstellen von Arrays von Strings

Ein Array von Strings kann einige Dinge bedeuten:

1. Ein Array, dessen Elemente `char *` s sind
2. Ein Array, dessen Elemente Arrays von `char`

Wir können ein Array von Zeichenzeigern so erstellen:

```
char * string_array[] = {
    "foo",
    "bar",
    "baz"
};
```

Denken Sie daran: Wenn Sie `char *` Zeichenfolgenliterale zuweisen, werden die Zeichenfolgen selbst im Nur-Lese-Speicher zugeordnet. Das Array `string_array` wird jedoch im Lese- / Schreibspeicher zugewiesen. Dies bedeutet, dass wir die Zeiger im Array ändern können, nicht aber die Zeichenfolgen, auf die sie zeigen.

In C ist der Parameter für main `argv` (das Array von Befehlszeilenargumenten, das bei Ausführung des Programms übergeben wurde) ein Array von `char *` : `char * argv[]` .

Wir können auch Arrays von Zeichenarrays erstellen. Da Zeichenfolgen Arrays von Zeichen sind, handelt es sich bei einem Array von Zeichenfolgen einfach um ein Array, dessen Elemente Zeichenarrays sind:

```
char modifiable_string_array_literals[][4] = {
    "foo",
    "bar",
    "baz"
};
```

Das ist äquivalent zu:

```
char modifiable_string_array[][4] = {
    {'f', 'o', 'o', '\0'},
    {'b', 'a', 'r', '\0'},
    {'b', 'a', 'z', '\0'}
};
```

Beachten Sie, dass wir als Größe der zweiten Dimension des Arrays `4` angeben. Jede der Zeichenfolgen in unserem Array besteht aus 4 Bytes, da das nullterminierende Zeichen eingefügt werden muss.

strstr

```
/* finds the next instance of needle in haystack
   zbpos: the zero-based position to begin searching from
   haystack: the string to search in
   needle: the string that must be found
   returns the next match of `needle` in `haystack`, or -1 if not found
*/
int findnext(int zbpos, const char *haystack, const char *needle)
{
    char *p;

    if (((p = strstr(haystack + zbpos, needle)) != NULL)
        return p - haystack;

    return -1;
}
```

`strstr` sucht der `haystack` (ersten) Argument für den String , auf die `needle` . Wenn gefunden, gibt `strstr` die Adresse des Vorkommens zurück. Wenn die `needle` nicht gefunden werden konnte, wird NULL zurückgegeben. Wir verwenden `zbpos` damit wir nicht immer und immer wieder dieselbe Nadel finden. Um die erste Instanz zu überspringen, fügen wir einen Versatz von `zbpos` . Ein

Notepad-Klon könnte `findnext` so aufrufen, um den Dialog " `findnext` " zu implementieren:

```
/*
    Called when the user clicks "Find Next"
    doc: The text of the document to search
    findwhat: The string to find
*/
void onfindnext(const char *doc, const char *findwhat)
{
    static int i;

    if ((i = findnext(i, doc, findwhat)) != -1)
        /* select the text starting from i and ending at i + strlen(findwhat) */
    else
        /* display a message box saying "end of search" */
}
```

String-Literale

String-Literale stellen nullterminierte Arrays von `char` statischer Dauer dar . Da sie über eine statische Speicherdauer verfügen, kann ein String-Literal oder ein Zeiger auf dasselbe zugrunde liegende Array auf verschiedene Weise verwendet werden, die ein Zeiger auf ein automatisches Array nicht bietet. Die Rückgabe eines String-Literal aus einer Funktion hat beispielsweise ein genau definiertes Verhalten:

```
const char *get_hello() {
    return "Hello, World!";  /* safe */
}
```

Aus historischen Gründen sind die Elemente des Arrays, die einem String-Literal entsprechen, nicht formal `const` . Jeder Versuch, sie zu ändern, hat jedoch ein undefiniertes Verhalten . Normalerweise stürzt ein Programm ab, das versucht, das Array, das einem String-Literal entspricht, zu ändern, oder es kommt zu einer Fehlfunktion.

```
char *foo = "hello";
foo[0] = 'y';  /* Undefined behavior - BAD! */
```

Wenn ein Zeiger auf ein Zeichenfolgenliteral zeigt - oder wo dies manchmal möglich ist - empfiehlt es sich, den referent `const` dieses Zeigers zu deklarieren, um zu vermeiden, dass ein solches undefiniertes Verhalten versehentlich aktiviert wird.

```
const char *foo = "hello";
/* GOOD: can't modify the string pointed to by foo */
```

Andererseits ist ein Zeiger auf oder in das zugrunde liegende Array eines Zeichenfolgenlitals nicht inhärent speziell. Sein Wert kann frei geändert werden, um auf etwas anderes hinzuweisen:

```
char *foo = "hello";
foo = "World!"; /* OK - we're just changing what foo points to */
```

Obwohl Initialisierer für `char` Arrays dieselbe Form wie String-Literale haben können, werden bei

Verwendung eines solchen Initialisierers dem initialisierten Array nicht die Merkmale eines String-Literalwerts verliehen. Der Initialisierer legt einfach die Länge und den anfänglichen Inhalt des Arrays fest. Insbesondere sind die Elemente modifizierbar, wenn sie nicht explizit als `const` deklariert werden:

```
char foo[] = "hello";
foo[0] = 'y';  /* OK! */
```

Einen String auf Null setzen

Sie können `memset` , um einen String (oder einen anderen Speicherblock) auf Null zu setzen.

Dabei ist `str` die zu `str` Zeichenfolge und `n` die Anzahl der Bytes in der Zeichenfolge.

```
#include <stdlib.h> /* For EXIT_SUCCESS */
#include <stdio.h>
#include <string.h>


int main(void)
{
  char str[42] = "fortytwo";
  size_t n = sizeof str; /* Take the size not the length. */

  printf("'%s'\n", str);

  memset(str, '\0', n);

  printf("'%s'\n", str);

  return EXIT_SUCCESS;
}
```

Drucke:

```
'fortytwo'
''
```

Ein anderes Beispiel:

```
#include <stdlib.h> /* For EXIT_SUCCESS */
#include <stdio.h>
#include <string.h>


#define FORTY_STR "forty"
#define TWO_STR "two"

int main(void)
{
  char str[42] = FORTY_STR TWO_STR;
  size_t n = sizeof str; /* Take the size not the length. */
  char * point_to_two = strstr(str, TWO_STR);
```

```
  printf("'%s'\n", str);

  memset(point_to_two, '\0', n);

  printf("'%s'\n", str);

  memset(str, '\0', n);

  printf("'%s'\n", str);

  return EXIT_SUCCESS;
}
```

Drucke:

```
'fortytwo'
'forty'
''
```

strspn und strcspn

Bei `strspn` einer Zeichenfolge berechnet `strspn` die Länge der ursprünglichen Teilzeichenfolge (Spanne), die ausschließlich aus einer bestimmten Liste von Zeichen besteht. `strcspn` ist ähnlich, außer es berechnet die Länge der ursprünglichen Teilzeichenfolge, die aus beliebigen Zeichen besteht, mit Ausnahme der aufgelisteten Zeichen:

```
/*
  Provided a string of "tokens" delimited by "separators", print the tokens along
  with the token separators that get skipped.
*/
#include <stdio.h>
#include <string.h>

int main(void)
{
    const char sepchars[] = ",.;!?";
    char foo[] = ";ball call,.fall gall hall!?.,";
    char *s;
    int n;

    for (s = foo; *s != 0; /*empty*/) {
        /* Get the number of token separator characters. */
        n = (int)strspn(s, sepchars);

        if (n > 0)
            printf("skipping separators: << %.*s >> (length=%d)\n", n, s, n);

        /* Actually skip the separators now. */
        s += n;

        /* Get the number of token (non-separator) characters. */
        n = (int)strcspn(s, sepchars);

        if (n > 0)
            printf("token found: << %.*s >> (length=%d)\n", n, s, n);

        /* Skip the token now. */
```

```
        s += n;
    }

    printf("== token list exhausted ==\n");

    return 0;
}
```

Analoge Funktionen, die `wcsspn` verwenden, sind `wcsspn` und `wcscspn` . Sie werden auf die gleiche Weise verwendet.

Zeichenketten kopieren

Zeigerzuweisungen kopieren keine Zeichenfolgen

Sie können die Verwendung `=` Operator ganzen Zahlen zu kopieren, aber man kann nicht die Verwendung `=` Operator Strings in C Strings in C kopieren sind als Arrays von Zeichen mit einem abschließenden Null-Zeichen dargestellt, so die Verwendung `=` Operator nur die Adresse speichern (Zeiger) einer Zeichenfolge.

```
#include <stdio.h>

int main(void) {
    int a = 10, b;
    char c[] = "abc", *d;

    b = a; /* Integer is copied */
    a = 20; /* Modifying a leaves b unchanged - b is a 'deep copy' of a */
    printf("%d %d\n", a, b); /* "20 10" will be printed */

    d = c;
    /* Only copies the address of the string -
    there is still only one string stored in memory */

    c[1] = 'x';
    /* Modifies the original string - d[1] = 'x' will do exactly the same thing */

    printf("%s %s\n", c, d); /* "axc axc" will be printed */

    return 0;
}
```

Das obige Beispiel wurde kompiliert, weil wir `char *d` anstelle von `char d[3]` . Die Verwendung des letzteren würde einen Compiler-Fehler verursachen. Sie können Arrays in C nicht zuordnen.

```
#include <stdio.h>

int main(void) {
    char a[] = "abc";
    char b[8];
```

```
    b = a; /* compile error */
    printf("%s\n", b);

    return 0;
}
```

Zeichenketten mit Standardfunktionen kopieren

strcpy()

Um Strings tatsächlich zu kopieren, `strcpy()` Funktion `strcpy()` in `string.h` . Vor dem Kopieren muss genügend Speicherplatz für das Ziel zugewiesen werden.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char a[] = "abc";
    char b[8];

    strcpy(b, a); /* think "b special equals a" */
    printf("%s\n", b); /* "abc" will be printed */

    return 0;
}
```

C99

snprintf()

Um einen Pufferüberlauf zu vermeiden, kann `snprintf()` verwendet werden. Diese Lösung ist nicht die beste Lösung, da sie die Vorlagenzeichenfolge parsen muss, aber sie ist die einzige Pufferlimit-sichere Funktion zum Kopieren von in der Standardbibliothek verfügbaren Zeichenfolgen, die ohne zusätzliche Schritte verwendet werden kann.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char a[] = "012345678901234567890";
    char b[8];

#if 0
    strcpy(b, a); /* causes buffer overrun (undefined behavior), so do not execute this here!
*/
#endif

    snprintf(b, sizeof(b), "%s", a); /* does not cause buffer overrun */
    printf("%s\n", b); /* "0123456" will be printed */

    return 0;
```

```
}
```

strncat()

Eine zweite Option mit besserer Leistung ist die Verwendung von `strncat()` (einer Pufferüberlauf-Überprüfungsversion von `strcat()`). Es `strcat()` ein drittes Argument, das die maximale Anzahl der zu kopierenden Bytes `strcat()` :

```
char dest[32];

dest[0] = '\0';
strncat(dest, source, sizeof(dest) - 1);
    /* copies up to the first (sizeof(dest) - 1) elements of source into dest,
    then puts a \0 on the end of dest */
```

Beachten Sie, dass diese Formulierung `sizeof(dest) - 1` ; Dies ist wichtig, da `strncat()` immer ein Null-Byte (gut) hinzufügt, dies jedoch nicht in der Größe der Zeichenfolge zählt (eine Ursache für Verwirrung und Pufferüberschreibungen).

Beachten Sie auch, dass die Alternative - Verketten nach einer nicht leeren Zeichenfolge - noch schwieriger ist. Erwägen:

```
char dst[24] = "Clownfish: ";
char src[] = "Marvin and Nemo";
size_t len = strlen(dst);

strncat(dst, src, sizeof(dst) - len - 1);
printf("%zu: [%s]\n", strlen(dst), dst);
```

Die Ausgabe ist:

```
23: [Clownfish: Marvin and N]
```

Beachten Sie jedoch, dass die als Länge angegebene Größe *nicht* die Größe des Zielarrays, sondern die Menge des verbleibenden Speicherplatzes war und nicht das Nullbyte des Terminals zählt. Dies kann zu großen Überschreibungsproblemen führen. Es ist auch etwas verschwenderisch; Um das Längenargument korrekt anzugeben, kennen Sie die Länge der Daten im Ziel. Daher können Sie stattdessen die Adresse des Nullbytes am Ende des vorhandenen Inhalts angeben. Dadurch wird `strncat()` vor dem erneuten Scannen geschützt:

```
    strcpy(dst, "Clownfish: ");
    assert(len < sizeof(dst) - 1);
    strncat(dst + len, src, sizeof(dst) - len - 1);
    printf("%zu: [%s]\n", strlen(dst), dst);
```

Dies erzeugt dieselbe Ausgabe wie zuvor, aber `strncat()` muss den vorhandenen Inhalt von `dst` nicht scannen, bevor er mit dem Kopieren beginnt.

strncpy()

Die letzte Option ist die Funktion `strncpy()` . Obwohl Sie denken, dass es zuerst kommen sollte, handelt es sich um eine eher betrügerische Funktion, die zwei Hauptstricke aufweist:

1. Wenn das Kopieren über `strncpy()` das Pufferlimit erreicht, wird kein abschließendes Nullzeichen geschrieben.
2. `strncpy()` füllt das Ziel immer vollständig aus, falls erforderlich mit null Bytes.

(Eine solche skurrile Implementierung ist historisch und war ursprünglich für die Verarbeitung von UNIX-Dateinamen vorgesehen.)

Der einzig richtige Weg, um es zu benutzen, ist die manuelle Absperrung der Null-Terminierung:

```
strncpy(b, a, sizeof(b)); /* the third parameter is destination buffer size */
b[sizeof(b)/sizeof(*b) - 1] = '\0'; /* terminate the string */
printf("%s\n", b); /* "0123456" will be printed */
```

Selbst dann, wenn Sie einen großen Puffer haben, wird die Verwendung von `strncpy()` wegen des zusätzlichen `strncpy()` sehr ineffizient.

Konvertieren Sie Strings in Number: atoi (), atof () (gefährlich, verwenden Sie sie nicht)

Warnung: Die Funktionen `atoi` , `atol` , `atoll` und `atof` sind von Natur aus unsicher, weil: *Wenn der Wert des Ergebnisses nicht dargestellt werden kann, ist das Verhalten nicht definiert.* (7.20.1p1)

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
    int val;
    if (argc < 2)
    {
        printf("Usage: %s <integer>\n", argv[0]);
        return 0;
    }

    val = atoi(argv[1]);

    printf("String value = %s, Int value = %d\n", argv[1], val);

    return 0;
}
```

Wenn die zu konvertierende Zeichenfolge eine gültige Dezimalzahl ist, die im Bereich liegt, funktioniert die Funktion:

```
$ ./atoi 100
String value = 100, Int value = 100
$ ./atoi 200
String value = 200, Int value = 200
```

Für Zeichenfolgen, die mit einer Zahl beginnen, gefolgt von einer anderen, wird nur die erste Zahl analysiert:

```
$ ./atoi 0x200
0
$ ./atoi 0123x300
123
```

In allen anderen Fällen ist das Verhalten undefiniert:

```
$ ./atoi hello
Formatting the hard disk...
```

Aufgrund der oben genannten Unklarheiten und dieses undefinierten Verhaltens sollte die Funktionsfamilie `atoi` niemals verwendet werden.

- Um in `long int` zu konvertieren, verwenden Sie `strtol()` anstelle von `atol()` .
- Verwenden `strtod()` zum Konvertieren in `double strtod()` anstelle von `atof()` .

C99

- Um in `long long int` zu konvertieren, verwenden Sie `strtoll()` anstelle von `atoll()` .

String formatierte Daten lesen / schreiben

Schreibe formatierte Daten in einen String

```
int sprintf ( char * str, const char * format, ... );
```

Verwenden `sprintf` Funktion `sprintf` , um Float-Daten in eine Zeichenfolge zu schreiben.

```
#include <stdio.h>
int main ()
{
  char buffer [50];
  double PI = 3.1415926;
  sprintf (buffer, "PI = %.7f", PI);
  printf ("%s\n",buffer);
  return 0;
}
```

Formatierte Daten aus String lesen

```
int sscanf ( const char * s, const char * format, ...);
```

Verwenden `sscanf` Funktion `sscanf` , um formatierte Daten zu analysieren.

```
#include <stdio.h>
int main ()
{
  char sentence []="date : 06-06-2012";
```

```
  char str [50];
  int year;
  int month;
  int day;
  sscanf (sentence,"%s : %2d-%2d-%4d", str, &day, &month, &year);
  printf ("%s -> %02d-%02d-%4d\n",str, day, month, year);
  return 0;
}
```

Strings in Number: strtoX-Funktionen sicher konvertieren

C99

Seit C99 verfügt die C-Bibliothek über eine Reihe sicherer Konvertierungsfunktionen, die eine Zeichenfolge als Zahl interpretieren. Ihre Namen haben die Form `strtoX` , wobei `X` für `l` , `ul` , `d` usw. steht, um den `strtoX` der Konvertierung zu bestimmen

```
double strtod(char const* p, char** endptr);
long double strtold(char const* p, char** endptr);
```

Sie stellen sicher, dass eine Konvertierung einen Über- oder Unterlauf hatte:

```
double ret = strtod(argv[1], 0); /* attempt conversion */

/* check the conversion result. */
if ((ret == HUGE_VAL || ret == -HUGE_VAL) && errno == ERANGE)
    return;  /* numeric overflow in in string */
else if (ret == HUGE_VAL && errno == ERANGE)
    return; /* numeric underflow in in string */

/* At this point we know that everything went fine so ret may be used */
```

Wenn die Zeichenfolge tatsächlich keine Nummer enthält, gibt diese Verwendung von `strtod` `0.0` .

Wenn dies nicht zufriedenstellend ist, kann der zusätzliche Parameter `endptr` verwendet werden. Es ist ein Zeiger auf einen Zeiger, der auf das Ende der erkannten Zahl in der Zeichenfolge zeigt. Wenn es wie oben auf `0` ist oder `NULL` , wird es einfach ignoriert.

Dieser `endptr` Parameter gibt an, ob die Konvertierung erfolgreich war und wenn ja, wo die Nummer endete:

```
char *check = 0;
double ret = strtod(argv[1], &check); /* attempt conversion */

/* check the conversion result. */
if (argv[1] == check)
    return; /* No number was detected in string */
else if ((ret == HUGE_VAL || ret == -HUGE_VAL) && errno == ERANGE)
    return; /* numeric overflow in in string */
else if (ret == HUGE_VAL && errno == ERANGE)
    return; /* numeric underflow in in string */

/* At this point we know that everything went fine so ret may be used */
```

Es gibt analoge Funktionen, die in breitere Integer-Typen konvertiert werden können:

```
long strtol(char const* p, char** endptr, int nbase);
long long strtoll(char const* p, char** endptr, int nbase);
unsigned long strtoul(char const* p, char** endptr, int nbase);
unsigned long long strtoull(char const* p, char** endptr, int nbase);
```

Diese Funktionen haben einen dritten Parameter `nbase` , der die Nummernbasis enthält, in die die Nummer geschrieben wird.

```
long a = strtol("101",   0, 2 ); /* a = 5L */
long b = strtol("101",   0, 8 ); /* b = 65L */
long c = strtol("101",   0, 10); /* c = 101L */
long d = strtol("101",   0, 16); /* d = 257L */
long e = strtol("101",   0, 0 ); /* e = 101L */
long f = strtol("0101",  0, 0 ); /* f = 65L */
long g = strtol("0x101", 0, 0 ); /* g = 257L */
```

Der Sonderwert `0` für `nbase` bedeutet, dass die Zeichenfolge genauso interpretiert wird, wie `nbase` in einem C-Programm interpretiert werden: Ein Präfix von `0x` entspricht einer hexadezimalen Darstellung, ansonsten ist eine führende `0` eine oktale Zahl und alle anderen Zahlen werden als Dezimalzahlen betrachtet.

Die praktischste Art und Weise, ein Befehlszeilenargument als Zahl zu interpretieren, wäre daher der beste Weg

```
int main(int argc, char* argv[] {
    if (argc < 1)
        return EXIT_FAILURE; /* No number given. */

    /* use strtoull because size_t may be wide */
    size_t mySize = strtoull(argv[1], 0, 0);

    /* then check conversion results. */

     ...

    return EXIT_SUCCESS;
}
```

Das bedeutet, dass das Programm mit einem Parameter in Oktal, Dezimal oder Hexadezimal aufgerufen werden kann.

Zeichenketten online lesen: https://riptutorial.com/de/c/topic/1990/zeichenketten

Kapitel 60: Zeiger

Einführung

Ein Zeiger ist ein Variablentyp, der die Adresse eines anderen Objekts oder einer Funktion speichern kann.

Syntax

- <Datentyp> * <Variablenname>;
- int * ptrToInt;
- void * ptrToVoid; / * C89 + * /
- struct someStruct * ptrToStruct;
- int ** ptrToPtrToInt;
- int arr [Länge]; int * ptrToFirstElem = arr; / * Für <C99 muss 'length' eine Kompilierungszeitkonstante sein, für> = C11 muss es möglicherweise eine sein. * /
- int * arrayOfPtrsToInt [Länge]; / * Für <C99 muss 'length' eine Kompilierungszeitkonstante sein, für> = C11 muss es möglicherweise eine sein. * /

Bemerkungen

Die Position des Sterns hat keinen Einfluss auf die Bedeutung der Definition:

```
/* The * operator binds to right and therefore these are all equivalent. */
int *i;
int * i;
int* i;
```

Wenn Sie jedoch mehrere Zeiger gleichzeitig definieren, benötigt jeder einen eigenen Stern:

```
int *i, *j; /* i and j are both pointers */
int* i, j;  /* i is a pointer, but j is an int not a pointer variable */
```

Ein Array von Zeigern ist ebenfalls möglich, wobei ein Sternchen vor dem Namen der Array-Variablen steht:

```
int *foo[2]; /* foo is a array of pointers, can be accessed as *foo[0] and *foo[1] */
```

Examples

Häufige Fehler

Die unsachgemäße Verwendung von Zeigern ist häufig eine Fehlerquelle, die Sicherheitsfehler oder Programmabstürze beinhalten kann, meistens aufgrund von Segmentierungsfehlern.

Nicht auf Zuordnungsfehler prüfen

Es kann nicht garantiert werden, dass die Speicherzuweisung erfolgreich ist, und kann stattdessen einen `NULL` Zeiger zurückgeben. Wenn Sie den zurückgegebenen Wert verwenden, ohne zu überprüfen, ob die Zuordnung erfolgreich ist, wird ein undefiniertes Verhalten aufgerufen. Dies führt in der Regel zu einem Absturz, es gibt jedoch keine Garantie dafür, dass es zu einem Absturz kommt. Wenn Sie sich darauf verlassen, kann dies ebenfalls zu Problemen führen.

Zum Beispiel unsichere Weise:

```
struct SomeStruct *s = malloc(sizeof *s);
s->someValue = 0; /* UNSAFE, because s might be a null pointer */
```

Sicherer Weg:

```
struct SomeStruct *s = malloc(sizeof *s);
if (s)
{
    s->someValue = 0; /* This is safe, we have checked that s is valid */
}
```

Verwenden Sie beim Anfordern von Speicher Literalzahlen anstelle von sizeof

Für eine bestimmte Compiler- / Maschinenkonfiguration haben Typen eine bekannte Größe. Es gibt jedoch keinen Standard, der definiert, dass die Größe eines bestimmten Typs (mit Ausnahme von `char`) für alle Compiler- / Maschinenkonfigurationen gleich ist. Wenn der Code 4 anstelle von `sizeof(int)` für die Speicherzuweisung verwendet, funktioniert er möglicherweise auf dem ursprünglichen Computer, der Code muss jedoch nicht unbedingt auf andere Computer oder Compiler übertragen werden. Feste Größen für Typen sollten durch `sizeof(that_type)` oder `sizeof(*var_ptr_to_that_type)` .

Nicht portierbare Zuteilung:

```
 int *intPtr = malloc(4*1000);    /* allocating storage for 1000 int */
 long *longPtr = malloc(8*1000);  /* allocating storage for 1000 long */
```

Portable Zuordnung:

```
 int *intPtr = malloc(sizeof(int)*1000);     /* allocating storage for 1000 int */
 long *longPtr = malloc(sizeof(long)*1000);  /* allocating storage for 1000 long */
```

Oder noch besser:

```
 int *intPtr = malloc(sizeof(*intPtr)*1000);     /* allocating storage for 1000 int */
 long *longPtr = malloc(sizeof(*longPtr)*1000);  /* allocating storage for 1000 long */
```

Speicherlecks

Wenn die Zuordnung des Speichers über `free` nicht mehr erfolgt, führt dies zu einem nicht wiederverwendbaren Speicher, der vom Programm nicht mehr verwendet wird. Dies wird als Speicherleck bezeichnet . Speicherverluste verschwenden Speicherressourcen und können zu Zuordnungsfehlern führen.

Logische Fehler

Alle Zuordnungen müssen demselben Muster folgen:

1. Zuweisung unter Verwendung `malloc` (oder `calloc`)
2. Verwendung zum Speichern von Daten
3. Aufteilung mit `free`

Wird auf diese Muster zu halten, wie Speicher nach einem Aufruf mit `free` (baumelt Zeiger) oder vor einem Aufruf von `malloc` (Wild Zeigern), rufe `free` zweimal („Double Free“) etc., in der Regel verursacht einen Segmentierungsfehler und führt zum Absturz des Programms.

Diese Fehler können vorübergehend und schwer zu debuggen sein. Zum Beispiel wird freigegebener Speicher normalerweise nicht sofort vom Betriebssystem zurückgefordert. Daher können baumelnde Zeiger für eine Weile bestehen bleiben und scheinen zu funktionieren.

In Systemen, in denen es funktioniert, ist Valgrind ein unschätzbares Werkzeug, um zu ermitteln, welcher Speicher verloren geht und wo er ursprünglich zugewiesen wurde.

Erstellen von Zeigern zum Stapeln von Variablen

Durch das Erstellen eines Zeigers wird die Lebensdauer der Variablen, auf die gezeigt wird, nicht verlängert. Zum Beispiel:

```
int* myFunction()
{
    int x = 10;
    return &x;
}
```

Hier hat `x` eine *automatische Speicherdauer* (allgemein als *Stapelzuordnung bezeichnet*). Da es auf dem Stack zugewiesen wird, ist seine Lebensdauer nur so lange, wie `myFunction` ausgeführt wird. Nachdem `myFunction` wurde, wird die Variable `x` zerstört. Diese Funktion ruft die Adresse von `x` (mit `&x`) ab und gibt sie an den Aufrufer zurück. Der Aufrufer erhält einen Zeiger auf eine nicht vorhandene Variable. Wenn Sie versuchen, auf diese Variable zuzugreifen, wird ein undefiniertes Verhalten ausgelöst.

Die meisten Compiler löschen einen Stack-Frame nach dem Beenden der Funktion nicht wirklich. Daher liefert dereferenzierte Zeiger oft die erwarteten Daten. Wenn jedoch eine andere Funktion aufgerufen wird, kann der Speicher, auf den gezeigt wird, überschrieben werden, und es scheint,

dass die Daten, auf die gezeigt wird, beschädigt wurden.

Um dies zu beheben, `malloc` den Speicher für die zurückzugebende Variable `malloc` und einen Zeiger auf den neu erstellten Speicher zurückgeben oder die Übergabe eines gültigen Zeigers an die Funktion, anstatt einen zu übergeben, z. B .:

```
#include <stdlib.h>
#include <stdio.h>

int *solution1(void)
{
    int *x = malloc(sizeof *x);
    if (x == NULL)
    {
        /* Something went wrong */
        return NULL;
    }

    *x = 10;

    return x;
}

void solution2(int *x)
{
    /* NB: calling this function with an invalid or null pointer
       causes undefined behaviour. */

    *x = 10;
}

int main(void)
{
    {
        /* Use solution1() */

        int *foo = solution1();
        if (foo == NULL)
        {
            /* Something went wrong */
            return 1;
        }

        printf("The value set by solution1() is %i\n", *foo);
        /* Will output: "The value set by solution1() is 10" */

        free(foo);    /* Tidy up */
    }

    {
        /* Use solution2() */

        int bar;
        solution2(&bar);

        printf("The value set by solution2() is %i\n", bar);
        /* Will output: "The value set by solution2() is 10" */
    }

    return 0;
```

```
}
```

Inkrementieren / Dekrementieren und Dereferenzieren

Wenn Sie `*p++` schreiben, um zu erhöhen, was von `p` angezeigt wird, sind Sie falsch.

Nach dem Inkrementieren / Dekrementieren wird vor der Dereferenzierung ausgeführt. Daher wird dieser Ausdruck den Zeiger `p` selbst inkrementieren und den Wert von `p` bevor er inkrementiert wird, ohne ihn zu ändern.

Sie sollten `(*p)++` schreiben, um zu erhöhen, was von `p` angezeigt wird.

Diese Regel gilt auch für das `*p--` : `*p--` dekrementiert den Zeiger `p` selbst, nicht das, was von `p` `*p--` wird.

Dereferenzieren eines Zeigers

```
int a = 1;
int *a_pointer = &a;
```

Um `a_pointer` zu dereferenzieren und den Wert von a zu ändern, verwenden wir die folgende Operation

```
*a_pointer = 2;
```

Dies kann mit den folgenden Druckanweisungen überprüft werden.

```
printf("%d\n", a); /* Prints 2 */
printf("%d\n", *a_pointer); /* Also prints 2 */
```

Es wäre jedoch falsch, einen `NULL` Zeiger oder einen anderen ungültigen Zeiger dereferenzieren zu wollen. Diese

```
int *p1, *p2;

p1 = (int *) 0xbad;
p2 = NULL;

*p1 = 42;
*p2 = *p1 + 1;
```

ist normalerweise undefiniertes Verhalten . `p1` darf nicht dereferenziert werden, da er auf eine Adresse `0xbad` verweist, die möglicherweise keine gültige Adresse ist. Wer weiß was da ist Es kann sich um den Speicher des Betriebssystems oder um den Speicher eines anderen Programms handeln. Der einzige Zeitcode wie dieser wird in der Embedded-Entwicklung verwendet, in der bestimmte Informationen an hart codierten Adressen gespeichert werden. `p2` kann nicht dereferenziert werden, da es `NULL` , was ungültig ist.

Dereferenzieren eines Zeigers auf eine Struktur

Nehmen wir an, wir haben folgende Struktur:

```
struct MY_STRUCT
{
    int my_int;
    float my_float;
};
```

Wir können `MY_STRUCT` so definieren, dass das Schlüsselwort `struct` sodass wir nicht jedes Mal die `struct MY_STRUCT` müssen. Dies ist jedoch optional.

```
typedef struct MY_STRUCT MY_STRUCT;
```

Wenn wir dann einen Zeiger auf eine Instanz dieser Struktur haben

```
MY_STRUCT *instance;
```

Wenn diese Anweisung im Dateibereich angezeigt wird, wird die `instance` beim Programmstart mit einem Nullzeiger initialisiert. Wenn diese Anweisung in einer Funktion erscheint, ist ihr Wert nicht definiert. Die Variable muss so initialisiert werden, dass sie auf eine gültige `MY_STRUCT` Variable oder auf einen dynamisch zugewiesenen Speicherplatz `MY_STRUCT` , bevor sie dereferenziert werden kann. Zum Beispiel:

```
MY_STRUCT info = { 1, 3.141593F };
MY_STRUCT *instance = &info;
```

Wenn der Zeiger gültig ist, können Sie ihn mit einer von zwei verschiedenen Schreibweisen auf seine Mitglieder zugreifen:

```
int a = (*instance).my_int;
float b = instance->my_float;
```

Obwohl beide Methoden funktionieren, ist es besser, den Operator arrow `->` anstelle der Kombination von Klammern, des dereference `*` Operators und des Punktes zu verwenden `.` Operator, weil es einfacher zu lesen und zu verstehen ist, insbesondere bei verschachtelten Anwendungen.

Ein weiterer wichtiger Unterschied wird unten gezeigt:

```
MY_STRUCT copy = *instance;
copy.my_int    = 2;
```

In diesem Fall enthält die `copy` eine Kopie des Inhalts der `instance` . Das `my_int` von `my_int` einer `copy` ändert dies nicht in der `instance` .

```
MY_STRUCT *ref = instance;
```

```
ref->my_int    = 2;
```

In diesem Fall verweist `ref` eine `instance` . `my_int` mithilfe der Referenz ändern, wird dies in der `instance` geändert.

Es ist allgemein üblich, Zeiger auf Strukturen als Parameter in Funktionen und nicht auf die Struktur selbst zu verwenden. Die Verwendung der Struktur als Funktionsparameter kann dazu führen, dass der Stapel überläuft, wenn die Struktur groß ist. Die Verwendung eines Zeigers auf eine Struktur beansprucht nur genügend Stapelspeicherplatz für den Zeiger, kann jedoch Nebenwirkungen verursachen, wenn die Funktion die Struktur ändert, die an die Funktion übergeben wird.

Funktionszeiger

Zeiger können auch zum Zeigen auf Funktionen verwendet werden.

Nehmen wir eine grundlegende Funktion:

```
int my_function(int a, int b)
{
    return 2 * a + 3 * b;
}
```

Nun definieren wir einen Zeiger des Typs dieser Funktion:

```
int (*my_pointer)(int, int);
```

Um eine zu erstellen, verwenden Sie einfach diese Vorlage:

```
return_type_of_func (*my_func_pointer)(type_arg1, type_arg2, ...)
```

Wir müssen diesen Zeiger dann der Funktion zuordnen:

```
my_pointer = &my_function;
```

Mit diesem Zeiger kann jetzt die Funktion aufgerufen werden:

```
/* Calling the pointed function */
int result = (*my_pointer)(4, 2);

...

/* Using the function pointer as an argument to another function */
void another_function(int (*another_pointer)(int, int))
{
    int a = 4;
    int b = 2;
    int result = (*another_pointer)(a, b);

    printf("%d\n", result);
}
```

Obwohl diese Syntax mit Basistypen natürlicher und kohärenter erscheint, erfordern das Zuordnen und Dereferenzieren von Funktionszeigern keine Verwendung der Operatoren `&` und `*` . Der folgende Ausschnitt ist also gleichermaßen gültig:

```
/* Attribution without the & operator */
my_pointer = my_function;

/* Dereferencing without the * operator */
int result = my_pointer(4, 2);
```

Um die Lesbarkeit von Funktionszeigern zu erhöhen, können Typedefs verwendet werden.

```
typedef void (*Callback)(int a);

void some_function(Callback callback)
{
    int a = 4;
    callback(a);
}
```

Ein weiterer Lesbarkeitstrick besteht darin, dass der C-Standard einen Funktionszeiger in Argumenten wie oben (aber nicht in der Variablendeklaration) auf etwas vereinfacht, das wie ein Funktionsprototyp aussieht. Daher kann Folgendes für Funktionsdefinitionen und Deklarationen gleichwertig verwendet werden:

```
void some_function(void callback(int))
{
    int a = 4;
    callback(a);
}
```

Siehe auch

Funktionszeiger

Zeiger initialisieren

Die Zeigerinitialisierung ist eine gute Möglichkeit, wilde Zeiger zu vermeiden. Die Initialisierung ist einfach und unterscheidet sich nicht von der Initialisierung einer Variablen.

```
#include <stddef.h>

int main()
{
    int *p1 = NULL;
    char *p2 = NULL;
    float *p3 = NULL;

         /* NULL is a macro defined in stddef.h, stdio.h, stdlib.h, and string.h */

    ...
```

```
}
```

Bei den meisten Betriebssystemen führt das versehentliche Verwenden eines Zeigers, der auf `NULL` initialisiert wurde, häufig zum sofortigen Absturz des Programms, wodurch die Ursache des Problems leicht ermittelt werden kann. Die Verwendung eines nicht initialisierten Zeigers kann häufig zu schwer diagnostizierbaren Fehlern führen.

Vorsicht:

Das Ergebnis der Dereferenzierung eines `NULL` Zeigers ist undefiniert, sodass es *nicht unbedingt zu einem Absturz führt,* selbst wenn dies das natürliche Verhalten des Betriebssystems ist, auf dem das Programm ausgeführt wird. Compiler-Optimierungen können den Absturz maskieren, dazu führen, dass der Absturz vor oder nach dem Punkt im Quellcode auftritt, an dem die Nullzeiger-Dereferenzierung aufgetreten ist, oder Teile des Codes, der die Nullzeiger-Dereferenzierung enthält, unerwartet aus dem Programm entfernt werden. Debug-Builds weisen diese Verhaltensweisen normalerweise nicht auf, dies wird jedoch durch den Sprachstandard nicht garantiert. Andere unerwartete und / oder unerwünschte Verhaltensweisen sind ebenfalls zulässig.

Da `NULL` niemals auf eine Variable, auf zugewiesenen Speicher oder auf eine Funktion verweist, ist die Verwendung als Guard-Wert sicher.

Vorsicht:

Normalerweise ist `NULL` als `(void *)0` . Dies bedeutet jedoch nicht, dass die zugewiesene Speicheradresse `0x0` . Weitere Informationen finden Sie unter C-faq für NULL-Zeiger

Beachten Sie, dass Sie Zeiger auch so initialisieren können, dass sie andere Werte als NULL enthalten.

```
int i1;

int main()
{
   int *p1 = &i1;
   const char *p2 = "A constant string to point to";
   float *p3 = malloc(10 * sizeof(float));
}
```

Adresse des Betreibers (&)

Für jedes Objekt (dh Variable, Array, Union, Struktur, Zeiger oder Funktion) kann der unäre Adressoperator verwendet werden, um auf die Adresse dieses Objekts zuzugreifen.

Nehme an, dass

```
int i = 1;
int *p = NULL;
```

Also dann eine Aussage `p = &i;` kopiert die Adresse der Variablen `i` in den Zeiger `p` .

Es wird als `p` ***Punkte auf*** `i` ausgedrückt.

`printf("%d\n", *p);` gibt 1 aus, der Wert von `i` .

Zeigerarithmetik

Bitte sehen Sie hier: Pointer Arithmetic

void * -Punkte als Argumente und geben Werte an Standardfunktionen zurück

K & R

`void*` ist ein catch-All-Typ für Zeiger auf Objekttypen. Ein Beispiel dafür ist die `malloc` Funktion, die als deklariert ist

```
void* malloc(size_t);
```

Der Rückgabetyp Zeiger auf ungültig bedeutet, dass der Rückgabewert von `malloc` einem Zeiger auf einen anderen Objekttyp zugewiesen werden kann:

```
int* vector = malloc(10 * sizeof *vector);
```

Es wird allgemein als gute Praxis angesehen, die Werte *nicht* explizit in leere Zeiger zu verschieben und aus ihnen herauszugeben. In bestimmten Fällen von `malloc()` dies darauf zurückzuführen, dass der Compiler bei einer expliziten Umwandlung einen falschen Rückgabetyp für `malloc()` annehmen kann, aber nicht warnt, wenn Sie vergessen, `stdlib.h` . Es ist auch der Fall, dass das richtige Verhalten von Leerzeiger verwendet wird, um sich besser dem DRY-Prinzip (nicht sich selbst wiederholen) anzupassen; Vergleichen Sie das obige mit dem folgenden, wobei der folgende Code mehrere unnötige zusätzliche Stellen enthält, an denen ein Tippfehler Probleme verursachen kann:

```
int* vector = (int*)malloc(10 * sizeof int*);
```

Ebenso Funktionen wie

```
void* memcpy(void *restrict target, void const *restrict source, size_t size);
```

haben ihre Argumente als `void *` da die Adresse eines Objekts unabhängig vom Typ übergeben werden kann. Auch hier sollte ein Aufruf keine Umwandlung verwenden

```
unsigned char buffer[sizeof(int)];
int b = 67;
memcpy(buffer, &b, sizeof buffer);
```

Const-Zeiger

Einzelzeiger

- Zeiger auf ein `int`

  Der Zeiger kann auf verschiedene Ganzzahlen zeigen und die `int` können durch den Zeiger geändert werden. Dieses Codebeispiel weist b zu, um auf `int b` b zu zeigen, und ändert dann den Wert von `b` in `100` .

  ```
  int b;
  int* p;
  p = &b;    /* OK */
  *p = 100;  /* OK */
  ```

- Zeiger auf eine `const int`

  Der Zeiger kann auf verschiedene ganze Zahlen zeigen, der Wert des `int` kann jedoch nicht über den Zeiger geändert werden.

  ```
  int b;
  const int* p;
  p = &b;    /* OK */
  *p = 100;  /* Compiler Error */
  ```

- `const` Zeiger auf `int`

  Der Zeiger kann nur auf ein `int` aber der Wert des `int` kann durch den Zeiger geändert werden.

  ```
  int a, b;
  int* const p = &b; /* OK as initialisation, no assignment */
  *p = 100;  /* OK */
  p = &a;    /* Compiler Error */
  ```

- `const` Zeiger auf `const int`

  Der Zeiger kann nur auf ein `int` und das `int` kann nicht durch den Zeiger geändert werden.

  ```
  int a, b;
  const int* const p = &b; /* OK as initialisation, no assignment */
  p = &a;   /* Compiler Error */
  *p = 100; /* Compiler Error */
  ```

Zeiger auf Zeiger

- Zeiger auf einen Zeiger auf ein `int`

  Dieser Code weist dem Doppelzeiger `p` die Adresse von `p1` zu (der dann auf `int* p1` (was auf `int`)).

Dann ändert `p1` auf `int a` . Ändert dann den Wert von a auf 100.

```
void f1(void)
{
  int a, b;
  int *p1;
  int **p;
  p1 = &b; /* OK */
  p = &p1; /* OK */
  *p = &a; /* OK */
  **p = 100; /* OK */
}
```

- Zeiger auf Zeiger auf ein `const int`

```
 void f2(void)
{
  int b;
  const int *p1;
  const int **p;
  p = &p1; /* OK */
  *p = &b; /* OK */
  **p = 100; /* error: assignment of read-only location '**p' */
}
```

- Zeiger auf `const` Zeiger auf ein `int`

```
void f3(void)
{
  int b;
  int *p1;
  int * const *p;
  p = &p1; /* OK */
  *p = &b; /* error: assignment of read-only location '*p' */
  **p = 100; /* OK */
}
```

- `const` Zeiger auf Zeiger auf `int`

```
void f4(void)
{
  int b;
  int *p1;
  int ** const p = &p1; /* OK as initialisation, not assignment */
  p = &p1; /* error: assignment of read-only variable 'p' */
  *p = &b; /* OK */
  **p = 100; /* OK */
}
```

- Zeiger auf `const` Zeiger auf `const int`

```
void f5(void)
{
  int b;
  const int *p1;
```

```
  const int * const *p;
  p = &p1; /* OK */
  *p = &b; /* error: assignment of read-only location '*p' */
  **p = 100; /* error: assignment of read-only location '**p' */
}
```

- `const` Zeiger auf `const int`

```
void f6(void)
{
  int b;
  const int *p1;
  const int ** const p = &p1; /* OK as initialisation, not assignment */
  p = &p1; /* error: assignment of read-only variable 'p' */
  *p = &b; /* OK */
  **p = 100; /* error: assignment of read-only location '**p' */
}
```

- `const` Zeiger auf `const` Zeiger auf `int`

```
void f7(void)
{
  int b;
  int *p1;
  int * const * const p = &p1; /* OK as initialisation, not assignment */
  p = &p1; /* error: assignment of read-only variable 'p'  */
  *p = &b; /* error: assignment of read-only location '*p' */
  **p = 100; /* OK */
}
```

Gleicher Stern, unterschiedliche Bedeutungen

Prämisse

Die verwirrendste Sache, die die Zeigersyntax in C und C ++ umgibt, besteht darin, dass es tatsächlich zwei verschiedene Bedeutungen gibt, wenn das Zeigersymbol, das Sternchen (`*`), mit einer Variablen verwendet wird.

Beispiel

Erstens verwenden Sie `*` **, um** eine Zeigervariable zu **deklarieren** .

```
int i = 5;
/* 'p' is a pointer to an integer, initialized as NULL */
int *p = NULL;
/* '&i' evaluates into address of 'i', which then assigned to 'p' */
p = &i;
/* 'p' is now holding the address of 'i' */
```

Wenn Sie nicht deklarieren (oder multiplizieren), wird mit `*` eine Zeigervariable **dereferenziert** :

```
*p = 123;
/* 'p' was pointing to 'i', so this changes value of 'i' to 123 */
```

Wenn Sie eine vorhandene Zeigervariable wollen Adresse von anderen Variablen zu halten, verwenden Sie **nicht** `*` , aber es wie folgt tun:

```
p = &another_variable;
```

Eine häufige Verwirrung unter den C-Programmierern entsteht, wenn sie gleichzeitig eine Zeigervariable deklarieren und initialisieren.

```
int *p = &i;
```

Da `int i = 5;` und `int i; i = 5;` geben Sie das gleiche Ergebnis, einige von ihnen denken vielleicht `int *p = &i;` und `int *p; *p = &i;` gib auch das gleiche ergebnis. Tatsache ist, nein, `int *p; *p = &i;` versucht, einen **nicht initialisierten** Zeiger **vorzugeben,** was zu UB führt. Verwenden Sie `*` niemals, wenn Sie keinen Zeiger deklarieren oder dereferenzieren.

Fazit

Das Sternchen (`*`) hat zwei verschiedene Bedeutungen in Bezug auf Zeiger, abhängig davon, wo es verwendet wird. Bei Verwendung in einer **Variablendeklaration** sollte der Wert auf der rechten Seite der Gleichen Seite ein ***Zeigerwert*** auf eine ***Adresse*** im Speicher sein. Wenn der Stern mit einer bereits **deklarierten Variablen verwendet wird** , wird der Zeigerwert ***dereferenziert,*** der Zeigerwert wird an die angegebene Stelle im Speicher weitergeleitet, und der dort gespeicherte Wert kann zugewiesen oder abgerufen werden.

Wegbringen

Wenn Sie mit Zeigern arbeiten, müssen Sie sozusagen auf Ihre Ps und Qs achten. Denken Sie daran, wenn Sie das Sternchen verwenden und was es bedeutet, wenn Sie es dort verwenden. Wenn Sie dieses winzige Detail übersehen, kann dies zu fehlerhaftem und / oder undefiniertem Verhalten führen, mit dem Sie sich wirklich nicht auseinandersetzen müssen.

Zeiger auf Zeiger

In C kann ein Zeiger auf einen anderen Zeiger verweisen.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int A = 42;
  int* pA = &A;
  int** ppA = &pA;
  int*** pppA = &ppA;

  printf("%d", ***pppA); /* prints 42 */
```

```
  return EXIT_SUCCESS;
}
```

Verweise und Verweise direkt sind jedoch nicht zulässig.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int A = 42;
  int* pA = &A;
  int** ppA = &&A; /* Compilation error here! */
  int*** pppA = &&&A;  /* Compilation error here! */

  ...
```

Einführung

Ein Zeiger wird wie jede andere Variable deklariert, mit der Ausnahme, dass ein Sternchen (`*`) zwischen dem Typ und dem Namen der Variablen steht, um anzuzeigen, dass es sich um einen Zeiger handelt.

```
int *pointer; /* inside a function, pointer is uninitialized and doesn't point to any valid
object yet */
```

Um zwei Zeigervariablen desselben Typs zu deklarieren, verwenden Sie in derselben Deklaration das Sternchen vor jedem Bezeichner. Zum Beispiel,

```
int *iptr1, *iptr2;
int *iptr3,  iptr4;  /* iptr3 is a pointer variable, whereas iptr4 is misnamed and is an int
*/
```

Der durch ein kaufmännisches Und (`&`) gekennzeichnete Adressen- oder Referenzoperator gibt die Adresse einer gegebenen Variablen an, die in einem Zeiger eines geeigneten Typs platziert werden kann.

```
int value = 1;
pointer = &value;
```

Der Indirektions- oder Dereferenzierungsoperator, der durch ein Sternchen (`*`) gekennzeichnet ist, ruft den Inhalt eines Objekts ab, auf das ein Zeiger zeigt.

```
printf("Value of pointed to integer: %d\n", *pointer);
/* Value of pointed to integer: 1 */
```

Wenn der Zeiger auf eine Struktur oder einen Unionstyp zeigt, können Sie ihn dereferenzieren und direkt auf seine Mitglieder mit dem Operator `->` zugreifen:

```
SomeStruct *s = &someObject;
```

```
s->someMember = 5; /* Equivalent to (*s).someMember = 5 */
```

In C ist ein Zeiger ein bestimmter Werttyp, der erneut zugewiesen werden kann und ansonsten als eigenständige Variable behandelt wird. Im folgenden Beispiel wird beispielsweise der Wert des Zeigers (der Variablen) selbst gedruckt.

```
printf("Value of the pointer itself: %p\n", (void *)pointer);
/* Value of the pointer itself: 0x7ffcd41b06e4 */
/* This address will be different each time the program is executed */
```

Da ein Zeiger eine veränderliche Variable ist, kann er nicht auf ein gültiges Objekt zeigen, indem er entweder auf null gesetzt wird

```
pointer = 0;     /* or alternatively */
pointer = NULL;
```

oder einfach durch das Enthalten eines beliebigen Bitmusters, das keine gültige Adresse ist. Letzteres ist eine sehr schlechte Situation, da es nicht getestet werden kann, bevor der Zeiger dereferenziert wird. Es gibt nur einen Test für den Fall, dass ein Zeiger null ist:

```
if (!pointer) exit(EXIT_FAILURE);
```

Ein Zeiger darf nur dereferenziert werden, wenn er auf ein *gültiges* Objekt zeigt. Andernfalls ist das Verhalten undefiniert. Viele moderne Implementierungen können Ihnen helfen, indem Sie Fehler wie Segmentierungsfehler auslösen und die Ausführung abbrechen. Andere können jedoch dazu führen, dass Ihr Programm in einem ungültigen Zustand bleibt.

Der vom Dereferenzierungsoperator zurückgegebene Wert ist ein veränderlicher Alias für die ursprüngliche Variable. Er kann also geändert werden, indem die ursprüngliche Variable geändert wird.

```
*pointer += 1;
printf("Value of pointed to variable after change: %d\n", *pointer);
/* Value of pointed to variable after change: 2 */
```

Zeiger sind auch neu zuweisbar. Dies bedeutet, dass ein Zeiger, der auf ein Objekt zeigt, später verwendet werden kann, um auf ein anderes Objekt desselben Typs zu zeigen.

```
int value2 = 10;
pointer = &value2;
printf("Value from pointer: %d\n", *pointer);
/* Value from pointer: 10 */
```

Zeiger haben wie jede andere Variable einen bestimmten Typ. Sie können die Adresse eines `short int` einem `long int` zuordnen. Ein solches Verhalten wird als Typ Punning bezeichnet und ist in C verboten, es gibt jedoch einige Ausnahmen.

Obwohl der Zeiger von einem bestimmten Typ sein muss, entspricht der für jeden Zeigertyp zugewiesene Speicher dem Speicher, der von der Umgebung zum Speichern von Adressen

verwendet wird, und nicht der Größe des Typs, auf den der Zeiger verweist.

```
#include <stdio.h>

int main(void) {
    printf("Size of int pointer: %zu\n", sizeof (int*));      /* size 4 bytes */
    printf("Size of int variable: %zu\n", sizeof (int));      /* size 4 bytes */
    printf("Size of char pointer: %zu\n", sizeof (char*));    /* size 4 bytes */
    printf("Size of char variable: %zu\n", sizeof (char));    /* size 1 bytes */
    printf("Size of short pointer: %zu\n", sizeof (short*));  /* size 4 bytes */
    printf("Size of short variable: %zu\n", sizeof (short));  /* size 2 bytes */
    return 0;
}
```

(Hinweis: Wenn Sie Microsoft Visual Studio verwenden, das die Standards C99 oder C11 nicht unterstützt, müssen Sie im obigen Beispiel `%Iu` [1] anstelle von `%zu` .)

Beachten Sie, dass die obigen Ergebnisse in Zahlen von Umgebung zu Umgebung variieren können, aber alle Umgebungen würden für unterschiedliche Zeigertypen die gleiche Größe aufweisen.

Auszug basierend auf Informationen der Cardiff University C Pointers Introduction

Zeiger und Arrays

Zeiger und Arrays sind in C eng miteinander verbunden. Arrays in C werden immer an zusammenhängenden Stellen im Speicher gehalten. Die Zeigerarithmetik wird immer durch die Größe des Elements skaliert, auf das gezeigt wird. Wenn wir also ein Array von drei Doubles und einen Zeiger auf die Basis haben, bezieht sich `*ptr` auf das erste Double, `*(ptr + 1)` auf das zweite, `*(ptr + 2)` auf das dritte. Eine bequemere Notation ist die Verwendung der Array-Notation `[]` .

```
double point[3] = {0.0, 1.0, 2.0};
double *ptr = point;

/* prints x 0.0, y 1.0 z 2.0 */
printf("x %f y %f z %f\n", ptr[0], ptr[1], ptr[2]);
```

Daher sind im Wesentlichen ptr und der Arrayname austauschbar. Diese Regel bedeutet auch, dass ein Array bei der Übergabe an ein Unterprogramm in einen Zeiger zerfällt.

```
double point[3] = {0.0, 1.0, 2.0};

printf("length of point is %s\n", length(point));

/* get the distance of a 3D point from the origin */
double length(double *pt)
{
   return sqrt(pt[0] * pt[0] + pt[1] * pt[1] + pt[2] * pt[2])
}
```

Ein Zeiger kann auf ein beliebiges Element in einem Array oder auf das Element hinter dem letzten Element zeigen. Es ist jedoch ein Fehler, einen Zeiger auf einen anderen Wert einschließlich des Elements vor dem Array zu setzen. (Der Grund ist, dass bei segmentierten Architekturen die Adresse vor dem ersten Element eine Segmentgrenze überschreitet. Der Compiler stellt sicher, dass dies nicht für das letzte Element plus eins geschieht.)

Fußnote 1: Informationen zum Microsoft-Format finden Sie über die Syntax `printf()` und Formatspezifikation .

Polymorphes Verhalten mit leeren Zeigern

Die Standardbibliotheksfunktion `qsort()` ist ein gutes Beispiel dafür, wie man mithilfe von Leerzeigern eine einzelne Funktion auf eine Vielzahl unterschiedlicher Typen anwenden kann.

```
void qsort (
    void *base,                                 /* Array to be sorted */
    size_t num,                                 /* Number of elements in array */
    size_t size,                                /* Size in bytes of each element */
    int (*compar)(const void *, const void *)); /* Comparison function for two elements */
```

Das zu sortierende Array wird als ungültiger Zeiger übergeben, sodass ein Array mit einem beliebigen Elementtyp bearbeitet werden kann. Die nächsten beiden Argumente geben `qsort()` wie viele Elemente im Array erwartet werden sollen und wie groß jedes Element in Byte ist.

Das letzte Argument ist ein Funktionszeiger auf eine Vergleichsfunktion, die selbst zwei leere Zeiger verwendet. Indem der Aufrufer diese Funktion zur Verfügung stellt, kann `qsort()` Elemente eines beliebigen Typs effektiv sortieren.

Hier ist ein Beispiel für eine solche Vergleichsfunktion zum Vergleichen von Schwimmern. Beachten Sie, dass alle Vergleichsfunktionen, die an `qsort()` werden, diese Art Signatur haben müssen. Die Art und Weise, wie sie polymorph gemacht wird, besteht darin, die leeren Zeigerargumente auf Zeiger des zu vergleichenden Elementtyps umzuwandeln.

```
int compare_floats(const void *a, const void *b)
{
    float fa = *((float *)a);
    float fb = *((float *)b);
    if (fa < fb)
        return -1;
    if (fa > fb)
        return 1;
    return 0;
}
```

Da wir wissen, dass qsort diese Funktion zum Vergleichen von Floats verwendet, wandeln wir die Argumente des leeren Zeigers wieder in Float-Pointer um, bevor sie dereferenzieren.

Nun ist die Verwendung der polymorphen Funktion qsort auf einem Array "Array" mit der Länge "len" sehr einfach:

```
qsort(array, len, sizeof(array[0]), compare_floats);
```

Zeiger online lesen: https://riptutorial.com/de/c/topic/1108/zeiger

Kapitel 61: Zufallszahlengenerierung

Bemerkungen

Aufgrund der Mängel von `rand()` sind im Laufe der Jahre viele andere Standardimplementierungen entstanden. Darunter sind:

- `arc4random()` (verfügbar für OS X und BSD)
- `random()` (verfügbar unter Linux)
- `drand48()` (verfügbar für POSIX)

Examples

Generelle Zufallszahlengenerierung

Die Funktion `rand()` kann verwendet werden, um einen Pseudo-Zufalls-Integer-Wert zwischen `0` und `RAND_MAX` (`0` und `RAND_MAX` enthalten) zu generieren.

`srand(int)` wird zum `srand(int)` des Pseudo-Zufallszahlengenerators verwendet. Jedes Mal, wenn `rand()` demselben Samen ausgesät wird, muss dieselbe Folge von Werten erzeugt werden. Es sollte nur einmal ausgesät werden, bevor `rand()` . Es sollte nicht jedes Mal wiederholt werden, wenn Sie einen neuen Stapel von Pseudo-Zufallszahlen generieren möchten.

Standardmäßig wird das Ergebnis der `time(NULL)` als Startwert verwendet. Wenn für Ihren Zufallszahlengenerator eine deterministische Sequenz erforderlich ist, können Sie den Generator bei jedem Programmstart mit dem gleichen Wert starten. Dies ist im Allgemeinen nicht für den Versionscode erforderlich, ist jedoch bei Debug-Läufen hilfreich, um Fehler reproduzierbar zu machen.

Es wird empfohlen, den Generator immer zu säen, wenn er nicht ausgesät wird, verhält er sich so, als ob er mit `srand(1)` ausgesät worden `srand(1)` .

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void) {
    int i;
    srand(time(NULL));
    i = rand();

    printf("Random value between [0, %d]: %d\n", RAND_MAX, i);
    return 0;
}
```

Mögliche Ausgabe:

```
Random value between [0, 2147483647]: 823321433
```

Anmerkungen:

Der C-Standard garantiert nicht die Qualität der erzeugten Zufallssequenz. In der Vergangenheit hatten einige Implementierungen von `rand()` ernsthafte Probleme bei der Verteilung und Zufälligkeit der generierten Zahlen. **Die Verwendung von `rand()` wird nicht für ernsthafte Zufallszahlengenerierungsanforderungen wie Kryptographie empfohlen.**

Permutierter kongruentieller Generator

Hier ist ein Standalone-Zufallszahlengenerator, der nicht auf `rand()` oder ähnlichen Bibliotheksfunktionen angewiesen ist.

Warum willst du so etwas? Möglicherweise vertrauen Sie nicht dem in Ihre Plattform integrierten Zufallszahlengenerator, oder Sie möchten eine reproduzierbare Zufallsquelle unabhängig von einer bestimmten Bibliotheksimplementierung.

Dieser Code ist PCG32 von pcg-random.org , ein modernes, schnelles Allzweck-RNG mit hervorragenden statistischen Eigenschaften. Es ist nicht kryptografisch sicher, verwenden Sie es also nicht für die Kryptographie.

```
#include <stdint.h>

/* *Really* minimal PCG32 code / (c) 2014 M.E. O'Neill / pcg-random.org
 * Licensed under Apache License 2.0 (NO WARRANTY, etc. see website) */

typedef struct { uint64_t state;  uint64_t inc; } pcg32_random_t;

uint32_t pcg32_random_r(pcg32_random_t* rng) {
    uint64_t oldstate = rng->state;
    /* Advance internal state */
    rng->state = oldstate * 6364136223846793005ULL + (rng->inc | 1);
    /* Calculate output function (XSH RR), uses old state for max ILP */
    uint32_t xorshifted = ((oldstate >> 18u) ^ oldstate) >> 27u;
    uint32_t rot = oldstate >> 59u;
    return (xorshifted >> rot) | (xorshifted << ((-rot) & 31));
}

void pcg32_srandom_r(pcg32_random_t* rng, uint64_t initstate, uint64_t initseq) {
    rng->state = 0U;
    rng->inc = (initseq << 1u) | 1u;
    pcg32_random_r(rng);
    rng->state += initstate;
    pcg32_random_r(rng);
}
```

Und so nennt man es:

```
#include <stdio.h>
int main(void) {
    pcg32_random_t rng; /* RNG state */
    int i;

    /* Seed the RNG */
    pcg32_srandom_r(&rng, 42u, 54u);
```

```
    /* Print some random 32-bit integers */
    for (i = 0; i < 6; i++)
        printf("0x%08x\n", pcg32_random_r(&rng));

    return 0;
}
```

Beschränken Sie die Erzeugung auf einen bestimmten Bereich

Normalerweise ist es bei der Erzeugung von Zufallszahlen sinnvoll, Ganzzahlen innerhalb eines Bereichs oder einen p-Wert zwischen 0,0 und 1,0 zu erzeugen. Während der Modulusbetrieb verwendet werden kann, um den Startwert auf eine niedrige ganze Zahl zu reduzieren, werden die niedrigen Bits verwendet, die häufig einen kurzen Zyklus durchlaufen, was zu einer geringfügigen Verteilung der Verteilung führt, wenn N im Verhältnis zu RAND_MAX groß ist.

Das Makro

```
#define uniform() (rand() / (RAND_MAX + 1.0))
```

produziert einen p-Wert von 0,0 bis 1,0 - epsilon, also

```
i = (int)(uniform() * N)
```

setzt `i` auf eine einheitliche Zufallszahl im Bereich von 0 bis N - 1.

Leider gibt es einen technischen Fehler, da RAND_MAX größer sein darf, als eine Variable vom Typ `double` genau darstellen kann. Dies bedeutet, dass `RAND_MAX + 1.0` zu RAND_MAX ausgewertet wird und die Funktion gelegentlich Eins zurückgibt. Dies ist jedoch unwahrscheinlich.

Xorshift Generation

Eine gute und einfache Alternative zu den fehlerhaften `rand()` Prozeduren ist *Xorshift* , eine Klasse von Pseudozufallszahlengeneratoren, die von George Marsaglia entdeckt wurde. Der Xorshift-Generator gehört zu den schnellsten nicht-kryptographisch sicheren Zufallszahlengeneratoren. Weitere Informationen und weitere Beispiele für Implementierungen finden Sie auf der Wikipedia-Seite zu Xorshift

Beispielimplementierung

```
#include <stdint.h>

/* These state variables must be initialised so that they are not all zero. */
uint32_t w, x, y, z;

uint32_t xorshift128(void)
{
    uint32_t t = x;
    t ^= t << 11U;
    t ^= t >> 8U;
    x = y; y = z; z = w;
```

```
    w ^= w >> 19U;
    w ^= t;
    return w;
}
```

Zufallszahlengenerierung online lesen:
https://riptutorial.com/de/c/topic/365/zufallszahlengenerierung

Kapitel 62: Zusammengesetzte Literale

Syntax

- (Typ) {Initialisierungsliste}

Bemerkungen

C-Standard sagt in C11-§6.5.2.5 / 3:

> Ein Postfix-Ausdruck, der aus einem eingeklammerten Typnamen gefolgt von einer in Klammern eingeschlossenen Liste von Initialisierern besteht, ist ein *zusammengesetztes Literal* . Es stellt ein unbenanntes Objekt bereit, dessen Wert von der Initialisierungsliste angegeben wird. [99)]

und Fußnote 99 sagt:

> Beachten Sie, dass sich dies von einem Besetzungsausdruck unterscheidet. Eine Umwandlung gibt beispielsweise eine Konvertierung in Skalartypen oder nur eine **Leere an** , und das Ergebnis eines Umwandlungsausdrucks ist kein Wert.

Beachten Sie, dass:

> String-Literale und zusammengesetzte Literale mit const-qualifizierten Typen müssen keine unterschiedlichen Objekte definieren. [101)]
>
> 101) Dies ermöglicht Implementierungen, um den Speicher für Stringliterale und konstante zusammengesetzte Literale mit der gleichen oder überlappenden Darstellung zu teilen.

Beispiel ist in Standard angegeben:
C11-§6.5.2.5 / 13:

> Wie String-Literale können const-qualifizierte Verbundliterale in den Nur-Lese-Speicher gestellt und sogar geteilt werden. Zum Beispiel,
>
> ```
> (const char []){"abc"} == "abc"
> ```
>
> kann 1 ergeben, wenn der Speicher der Literale gemeinsam genutzt wird.

Examples

Definition / Initialisierung von zusammengesetzten Literalen

Ein zusammengesetztes Literal ist ein unbenanntes Objekt, das in dem definierten Bereich erstellt wird. Das Konzept wurde erstmals im C99-Standard eingeführt. Ein Beispiel für zusammengesetzte Wörter ist

Beispiele aus dem C-Standard C11-§6.5.2.5 / 9:

```
int *p = (int [2]){ 2, 4 };
```

`p` wird auf die Adresse des ersten Elements eines unbenannten Arrays von zwei Ints initialisiert.

Das zusammengesetzte Literal ist ein Wert. Die Speicherdauer des unbenannten Objekts ist entweder statisch (wenn das Literal im Dateibereich angezeigt wird) oder automatisch (wenn das Literal im Blockbereich angezeigt wird). In letzterem Fall endet die Lebensdauer des Objekts, wenn die Steuerung den umgebenden Block verlässt.

```
void f(void)
{
    int *p;
    /*...*/
    p = (int [2]){ *p };
    /*...*/
}
```

> `p` wird der Adresse des ersten Elements eines Arrays von zwei Ints zugewiesen, wobei das erste den Wert hat, auf den zuvor `p` und das zweite, Null.

Hier bleibt `p` bis zum Ende des Blocks gültig.

Zusammengesetztes Literal mit Bezeichnern

(auch ab C11)

```
  struct point {
    unsigned x;
    unsigned y;
  };

  extern void drawline(struct point, struct point);

 // used somewhere like this
 drawline((struct point){.x=1, .y=1}, (struct point){.x=3, .y=4});
```

Eine fiktive Funktion `drawline` empfängt zwei Argumente vom Typ `struct point` . Der erste hat Koordinatenwerte `x == 1` und `y == 1` , während der zweite `x == 3` und `y == 4`

Zusammengesetztes Literal ohne Angabe der Arraylänge

```
int *p = (int []){ 1, 2, 3};
```

In diesem Fall wird die Größe des Arrays nicht angegeben und dann von der Länge des Initialisierers bestimmt.

Zusammengesetztes Literal, dessen Initialisierungslänge weniger als die angegebene Arraygröße ist

```
int *p = (int [10]){1, 2, 3};
```

Die restlichen Elemente des zusammengesetzten Literal werden implizit auf `0` .

Schreibgeschütztes zusammengesetztes Literal

Beachten Sie, dass ein zusammengesetztes Literal ein Wert ist und seine Elemente daher modifiziert werden können. Ein *schreibgeschütztes* zusammengesetztes Literal kann mit `const` Qualifier als `(const int[]){1,2}` .

Zusammengesetztes Literal mit beliebigen Ausdrücken

Innerhalb einer Funktion kann ein zusammengesetztes Literal wie für jede Initialisierung seit C99 beliebige Ausdrücke haben.

```
void foo()
{
    int *p;
    int i = 2; j = 5;
    /*...*/
    p = (int [2]){ i+j, i*j };
    /*...*/
}
```

Zusammengesetzte Literale online lesen: https://riptutorial.com/de/c/topic/4135/zusammengesetzte-literale

Kapitel 63: Zusammenstellung

Einführung

Die C-Sprache ist traditionell eine kompilierte Sprache (im Gegensatz zu interpretiert). Der C-Standard definiert ***Übersetzungsphasen*** , und das Produkt ihrer Anwendung ist ein Programmbild (oder ein kompiliertes Programm). In c11 sind die Phasen in §5.1.1.2 aufgeführt.

Bemerkungen

Dateinamenerweiterung	Beschreibung
`.c`	Quelldatei Enthält normalerweise Definitionen und Code.
`.h`	Header-Datei. Enthält normalerweise Deklarationen.
`.o`	Objektdatei Kompilierter Code in Maschinensprache.
`.obj`	Alternative Erweiterung für Objektdateien.
`.a`	Bibliotheksdatei. Paket von Objektdateien.
`.dll`	Dynamic Link Library unter Windows.
`.so`	Shared Object (Bibliothek) auf vielen Unix-ähnlichen Systemen.
`.dylib`	Dynamic-Link Library unter OSX (Unix-Variante).
`.exe` , `.com`	Ausführbare Windows-Datei. Wird durch Verknüpfen von Objektdateien und Bibliotheksdateien gebildet. In Unix-ähnlichen Systemen gibt es keine spezielle Dateinamenerweiterung für ausführbare Dateien.

POSIX c99-Compiler-Flags	Beschreibung
`-o filename`	Name der Ausgabedatei, z. (`bin/program.exe` , `program`)
`-I directory`	Suche nach Headern im `direrctory` .
`-D name`	definieren `name`
`-L directory`	Suche nach Bibliotheken im `directory` .
`-l name`	`libname` .

Compiler auf POSIX-Plattformen (Linux, Mainframes, Mac) akzeptieren diese Optionen normalerweise, auch wenn sie nicht als `c99` .

- Siehe auch c99 - Standard-C-Programme übersetzen

GCC-Flags (GNU Compiler Collection)	Beschreibung
`-Wall`	Aktiviert alle Warnmeldungen, die allgemein akzeptiert werden, als nützlich.
`-Wextra`	Aktiviert weitere Warnmeldungen, kann zu laut sein.
`-pedantic`	Erzwingen Sie Warnungen, wenn der Code den gewählten Standard verletzt.
`-Wconversion`	Aktivieren Sie Warnungen bei der impliziten Konvertierung.
`-c`	Kompiliert Quelldateien ohne Verknüpfung.
`-v`	Gibt Informationen zur Zusammenstellung aus.

- `gcc` akzeptiert die POSIX-Flags und viele andere.
- Viele andere Compiler auf POSIX-Plattformen (`clang` , herstellerspezifische Compiler) verwenden ebenfalls die oben aufgeführten Flags.
- Siehe auch GCC aufrufen für viele weitere Optionen.

TCC (Tiny C Compiler) -Flaggen	Beschreibung
`-Wimplicit-function-declaration`	Warnung vor impliziter Funktionsdeklaration
`-Wunsupported`	Warnung vor nicht unterstützten GCC-Funktionen, die von TCC ignoriert werden.
`-Wwrite-strings`	Machen Sie String-Konstanten vom Typ const char * anstelle von char *.
`-Werror`	Kompilierung abbrechen, wenn Warnungen ausgegeben werden.
`-Wall`	Aktivieren Sie alle Warnungen mit Ausnahme der `-Werror` , " `-Wunusupported` und " `-Wwrite strings` .

Examples

Der Linker

Die Aufgabe des Linkers besteht darin, eine Reihe von Objektdateien (`.o` Dateien) zu einer binären ausführbaren Datei zu verknüpfen. Bei der *Verknüpfung werden* hauptsächlich *symbolische Adressen in numerische Adressen aufgelöst* . Das Ergebnis des

Verknüpfungsprozesses ist normalerweise ein ausführbares Programm.

Während des Verbindungsvorgangs nimmt der Linker alle auf der Befehlszeile angegebenen Objektmodule auf, fügt systemspezifischen *Startcode hinzu* und versucht, alle *externen* Verweise im Objektmodul mit *externen Definitionen* in anderen Objektdateien (Objektdateien) aufzulösen kann direkt in der Befehlszeile angegeben werden oder implizit durch Bibliotheken hinzugefügt werden). Es wird dann *Last* - *Adressen* für die Objektdateien zugeordnet werden , das heißt, es gibt an, wo der Code und die Daten werden in dem Adressraum des fertigen Programms enden. Sobald sie die Ladeadressen erhalten hat, kann sie alle symbolischen Adressen im Objektcode durch "echte" numerische Adressen im Adressraum des Ziels ersetzen. Das Programm kann jetzt ausgeführt werden.

Dies umfasst sowohl die Objektdateien, die der Compiler aus Ihren Quellcodedateien erstellt hat, als auch Objektdateien, die für Sie vorkompiliert und in Bibliotheksdateien gesammelt wurden. Diese Dateien haben Namen, die auf `.a` oder `.so` `.a` `.so` müssen Sie sie nicht kennen, da der Linker weiß, wo sich die meisten befinden, und er wird sie bei Bedarf automatisch einbinden.

Implizites Aufrufen des Linkers

Wie der Vorprozessor ist der Linker ein separates Programm, das häufig als `ld` (Linux verwendet `collect2` zum Beispiel `collect2`). Ebenso wie der Vorprozessor wird der Linker automatisch aufgerufen, wenn Sie den Compiler verwenden. Die normale Art, den Linker zu verwenden, lautet daher wie folgt:

```
% gcc foo.o bar.o baz.o -o myprog
```

Diese Zeile weist den Compiler an, drei Objektdateien (`foo.o` , `bar.o` und `baz.o`) zu einer binären ausführbaren Datei namens `myprog` zu `myprog` . Jetzt haben Sie eine Datei namens `myprog` , die Sie ausführen können und die hoffentlich etwas `myprog` und / oder Nützliches tun wird.

Expliziter Aufruf des Linkers

Es ist möglich, den Linker direkt aufzurufen, dies ist jedoch selten ratsam und in der Regel sehr plattformspezifisch. Das heißt, Optionen, die unter Linux funktionieren, funktionieren unter Solaris, AIX, MacOS, Windows und anderen Plattformen nicht unbedingt. Wenn Sie mit GCC arbeiten, können Sie mit `gcc -v` sehen, was in Ihrem Namen ausgeführt wird.

Optionen für den Linker

Der Linker benötigt auch einige Argumente, um sein Verhalten zu ändern. Der folgende Befehl würde gcc `bar.o` , `foo.o` und `bar.o` zu verknüpfen, aber auch die Bibliothek `ncurses` .

```
% gcc foo.o bar.o -o foo -lncurses
```

Dies ist eigentlich (mehr oder weniger) äquivalent zu

```
% gcc foo.o bar.o /usr/lib/libncurses.so -o foo
```

(obwohl `libncurses.so` sein `libncurses.a` , was nur ein Archiv ist, das mit `ar`). Beachten Sie, dass Sie die Bibliotheken (entweder über Pfadnamen oder über `-lname` Optionen `-lname`) nach den Objektdateien `-lname` . Bei statischen Bibliotheken spielt die Reihenfolge, in der sie angegeben werden, eine Rolle. Bei gemeinsam genutzten Bibliotheken spielt die Reihenfolge oft keine Rolle.

Beachten Sie, dass auf vielen Systemen, wenn Sie mathematische Funktionen verwenden (von `<math.h>`), müssen Sie angeben , `-lm` die Mathematik - Bibliothek zu laden - aber Mac OS X und Mac OS Sierra dies nicht erforderlich ist . Es gibt andere Bibliotheken, die separate Bibliotheken auf Linux und anderen Unix-Systemen sind, jedoch nicht auf macOS - POSIX-Threads und POSIX-Echtzeit und Netzwerkbibliotheken. Folglich variiert der Verknüpfungsprozess zwischen den Plattformen.

Andere Zusammenstellungsoptionen

Dies ist alles, was Sie wissen müssen, um Ihre eigenen C-Programme zu erstellen. Im Allgemeinen empfehlen wir auch die Verwendung der `-Wall` :

```
% gcc -Wall -c foo.cc
```

Die Option `-Wall` bewirkt, dass der Compiler Sie vor legalen, aber zweifelhaften Code-Konstrukten warnt, und hilft Ihnen, sehr früh Fehler zu finden.

Wenn Sie möchten, dass der Compiler weitere Warnungen ausgibt (einschließlich deklarierter, aber nicht verwendeter Variablen, vergessen Sie die Rückgabe eines Werts usw.), können Sie diese Optionen verwenden, da sich `-Wall` trotz des Namens nicht dreht *alle möglichen Warnungen* zu:

```
% gcc -Wall -Wextra -Wfloat-equal -Wundef -Wcast-align -Wwrite-strings -Wlogical-op \
>      -Wmissing-declarations -Wredundant-decls -Wshadow …
```

Beachten Sie, dass `clang` eine Option hat - `-Weverything` was wirklich alle Warnungen in `clang` `-Weverything` .

Datentypen

Zum Kompilieren von C-Programmen müssen Sie mit fünf Arten von Dateien arbeiten:

1. **Quelldateien** : Diese Dateien enthalten Funktionsdefinitionen und haben Namen, die nach Konvention auf `.c` enden. Hinweis: `.cc` und `.cpp` sind C ++ - Dateien. *keine* C-Dateien.
   zB `foo.c`

2. **Header-Dateien** : Diese Dateien enthalten Funktionsprototypen und verschiedene Präprozessoranweisungen (siehe unten). Sie werden verwendet, um Quellcodedateien den Zugriff auf extern definierte Funktionen zu ermöglichen. Header-Dateien enden nach Konvention in `.h` .
   zB `foo.h`

3. **Objektdateien** : Diese Dateien werden als Ausgabe des Compilers erzeugt. Sie bestehen

aus Funktionsdefinitionen in binärer Form, sind jedoch nicht selbst ausführbar. Objektdateien enden nach der Konvention `.o` , obwohl sie bei einigen Betriebssystemen (z. B. Windows, MS-DOS) häufig auf `.obj` .
zB `foo.o` `foo.obj`

4. **Binäre ausführbare Dateien** : Diese werden als Ausgabe eines Programms erzeugt, das als "Linker" bezeichnet wird. Der Linker verknüpft mehrere Objektdateien miteinander, um eine Binärdatei zu erzeugen, die direkt ausgeführt werden kann. Binäre ausführbare Dateien haben unter Unix-Betriebssystemen kein spezielles Suffix, obwohl sie unter Windows `.exe` auf `.exe` enden.
zB `foo` `foo.exe`

5. **Bibliotheken** : Eine Bibliothek ist eine kompilierte Binärdatei, ist jedoch keine ausführbare Datei (dh es gibt keine `main()` Funktion in einer Bibliothek). Eine Bibliothek enthält Funktionen, die von mehr als einem Programm verwendet werden können. Eine Bibliothek sollte mit Header-Dateien geliefert werden, die Prototypen für alle Funktionen in der Bibliothek enthalten. Diese Header-Dateien sollten in jeder Quelldatei, die die Bibliothek verwendet, referenziert werden (z. B. `#include <library.h>`). Der Linker muss dann auf die Bibliothek verwiesen werden, damit das Programm erfolgreich kompiliert werden kann. Es gibt zwei Arten von Bibliotheken: statisch und dynamisch.

   - **Statische Bibliothek** : Eine statische Bibliothek (`.a` Dateien für POSIX-Systeme und `.lib` Dateien für Windows - nicht zu verwechseln mit DLL-Import-Bibliotheksdateien , die ebenfalls die Erweiterung `.lib`) ist statisch in das Programm integriert. Statische Bibliotheken haben den Vorteil, dass das Programm genau weiß, welche Version einer Bibliothek verwendet wird. Andererseits sind die ausführbaren Dateien größer, da alle verwendeten Bibliotheksfunktionen enthalten sind.
   zB `libfoo.a` `foo.lib`
   - **Dynamische Bibliothek** : Eine dynamische Bibliothek (`.so` Dateien für die meisten POSIX-Systeme, `.dylib` für `.dylib` und `.dll` Dateien für Windows) wird vom Programm zur Laufzeit dynamisch verknüpft. Diese werden manchmal auch als gemeinsam genutzte Bibliotheken bezeichnet, da ein Bibliotheksbild von vielen Programmen gemeinsam genutzt werden kann. Dynamische Bibliotheken haben den Vorteil, dass weniger Speicherplatz beansprucht wird, wenn die Bibliothek von mehr als einer Anwendung verwendet wird. Sie ermöglichen auch Bibliotheksaktualisierungen (Fehlerbehebungen), ohne dass ausführbare Dateien neu erstellt werden müssen.
   zB `foo.so` `foo.dylib` `foo.dll`

Der Präprozessor

Bevor der C-Compiler mit dem Kompilieren einer Quellcodedatei beginnt, wird die Datei in einer Vorverarbeitungsphase verarbeitet. Diese Phase kann durch ein separates Programm erfolgen oder vollständig in eine ausführbare Datei integriert werden. In jedem Fall wird es automatisch vom Compiler aufgerufen, bevor die eigentliche Kompilierung beginnt. Die Vorverarbeitungsphase konvertiert Ihren Quellcode in einen anderen Quellcode oder eine andere Übersetzungseinheit, indem Sie Text ersetzen. Sie können es sich als "modifizierten" oder "erweiterten" Quellcode vorstellen. Diese erweiterte Quelle kann als echte Datei im Dateisystem vorhanden sein oder nur

für kurze Zeit im Speicher gespeichert werden, bevor sie weiter verarbeitet wird.

Präprozessor-Befehle beginnen mit dem Nummernzeichen ("#"). Es gibt mehrere Präprozessor-Befehle. zwei der wichtigsten sind:

1. **Definiert** :

   `#define` wird hauptsächlich zur Definition von Konstanten verwendet. Zum Beispiel,

   ```
   #define BIGNUM 1000000
   int a = BIGNUM;
   ```

   wird

   ```
   int a = 1000000;
   ```

   `#define` wird auf diese Weise verwendet, um nicht explizit einen konstanten Wert an vielen verschiedenen Stellen in einer Quellcodedatei schreiben zu müssen. Dies ist wichtig, wenn Sie den konstanten Wert später ändern müssen. Es ist viel weniger fehleranfällig, es einmal in `#define` zu ändern, als es an mehreren Stellen im gesamten Code ändern zu müssen.

   Da `#define` nur erweiterte Such- und Ersetzungsvorgänge durchführt, können Sie auch Makros deklarieren. Zum Beispiel:

   ```
   #define ISTRUE(stm) do{stm = stm ? 1 : 0;}while(0)
   // in the function:
   a = x;
   ISTRUE(a);
   ```

   wird:

   ```
   // in the function:
   a = x;
   do {
       a = a ? 1 : 0;
   } while(0);
   ```

   Auf den ersten Näherungswert ist dieser Effekt ungefähr derselbe wie bei Inline-Funktionen, aber der Präprozessor bietet keine Typüberprüfung für `#define` Makros. Dies ist bekanntermaßen fehleranfällig, und ihre Verwendung erfordert große Vorsicht.

   Beachten Sie auch hier, dass der Präprozessor auch Kommentare durch Leerzeichen ersetzt, wie unten erläutert.

2. **Beinhaltet** :

   `#include` wird verwendet, um auf Funktionsdefinitionen zuzugreifen, die außerhalb einer Quellcodedatei definiert sind. Zum Beispiel:

   ```
    #include <stdio.h>
   ```

bewirkt, dass der Präprozessor den Inhalt von `<stdio.h>` in die Quellcodedatei an der Position der `#include` Anweisung `#include` , bevor er kompiliert wird. `#include` wird fast immer verwendet, um Header-Dateien `#define` . `#define` sind Dateien, die hauptsächlich Funktionsdeklarationen und `#include` `#define` Anweisungen enthalten. In diesem Fall verwenden wir `#include` , um Funktionen wie `printf` und `scanf` , deren Deklarationen sich in der Datei `stdio.h` . C-Compiler erlauben keine Verwendung einer Funktion, es sei denn, sie wurde zuvor in dieser Datei deklariert oder definiert. `#include` Anweisungen sind daher die Möglichkeit, zuvor geschriebenen Code in Ihren C-Programmen wiederzuverwenden.

3. **Logikoperationen** :

```
#if defined A || defined B
variable = another_variable + 1;
#else
variable = another_variable * 2;
#endif
```

   wird geändert in:

```
variable = another_variable + 1;
```

   wenn A oder B zuvor im Projekt definiert wurden. Wenn dies nicht der Fall ist, führt der Präprozessor dies natürlich aus:

```
variable = another_variable * 2;
```

   Dies wird häufig für Code verwendet, der auf verschiedenen Systemen läuft oder auf verschiedenen Compilern kompiliert wird. Da es globale Definitionen gibt, die compiler- / systemspezifisch sind, können Sie diese Definitionen testen und den Compiler immer nur den Code verwenden lassen, den er sicher kompiliert.

4. **Bemerkungen**

   Der Präprozessor ersetzt alle Kommentare in der Quelldatei durch einzelne Leerzeichen. Kommentare werden durch `//` bis zum Ende der Zeile oder durch eine Kombination aus öffnendem `/*` und schließendem `*/` -Kommentar angegeben.

Der Compiler

Nachdem der C-Vorprozessor alle Header-Dateien enthalten und alle Makros erweitert hat, kann der Compiler das Programm kompilieren. Dazu wird der C-Quellcode in eine Objektcodedatei umgewandelt, die auf `.o` endet und die binäre Version des Quellcodes enthält. Der Objektcode ist jedoch nicht direkt ausführbar. Um eine ausführbare Datei zu machen, müssen Sie auch Code hinzufügen , für alle Bibliotheksfunktionen , die waren `#include` d in die Datei (dies ist nicht das gleiche wie die Erklärungen , einschließlich, das ist , was `#include` tut). Dies ist die Aufgabe des Linkers .

Im Allgemeinen hängt die genaue Reihenfolge, in der ein C-Compiler aufgerufen wird, stark von

dem System ab, das Sie verwenden. Hier verwenden wir den GCC-Compiler, wobei zu beachten ist, dass viele weitere Compiler existieren:

```
% gcc -Wall -c foo.c
```

`%` ist die Eingabeaufforderung des Betriebssystems. Dies weist den Compiler an, den Vorprozessor für die Datei `foo.c` und ihn dann in die Objektcodedatei `foo.o` . Die Option `-c` bedeutet, die Quellcodedatei in eine Objektdatei zu kompilieren, den Linker jedoch nicht aufzurufen. Diese Option `-c` ist auf POSIX-Systemen wie Linux oder macOS verfügbar. andere Systeme verwenden möglicherweise eine andere Syntax.

Wenn sich Ihr gesamtes Programm in einer Quellcode-Datei befindet, können Sie stattdessen Folgendes tun:

```
% gcc -Wall foo.c -o foo
```

Dadurch wird der Compiler `foo.c` , den Vorprozessor auf `foo.c` , zu kompilieren und dann zu verknüpfen, um eine ausführbare Datei namens `foo` zu erstellen. Die Option `-o` gibt an, dass das nächste Wort in der Zeile der Name der ausführbaren binären Datei (Programm) ist. Wenn Sie nicht `-o` angeben (wenn Sie nur `gcc foo.c`), wird die ausführbare Datei aus historischen Gründen als `a.out` bezeichnet.

Im Allgemeinen führt der Compiler beim Konvertieren einer `.c` Datei in eine ausführbare Datei vier Schritte aus:

1. **Vorverarbeitung** - erweitert textlich die Anweisungen `#include` und `#include #define` Makros in Ihrer `.c` Datei
2. **Kompilierung** - wandelt das Programm in eine Assembly um (Sie können den Compiler bei diesem Schritt durch Hinzufügen der Option `-S` stoppen)
3. **Assembly** - wandelt die Assembly in Maschinencode um
4. **Verknüpfung** - Verknüpft den Objektcode mit externen Bibliotheken, um eine ausführbare Datei zu erstellen

Beachten Sie auch, dass der Name des Compilers, den wir verwenden, GCC ist, der je nach Kontext sowohl für "GNU C-Compiler" als auch für "GNU-Compiler-Collection" steht. Andere C-Compiler existieren. Bei Unix-ähnlichen Betriebssystemen haben viele von ihnen den Namen `cc` für "C-Compiler", der häufig eine symbolische Verbindung zu einem anderen Compiler darstellt. Auf Linux-Systemen ist `cc` häufig ein Alias für GCC. Unter Mac OS oder OS-X deutet es auf Lärm.

Der POSIX-Standard `c99` derzeit `c99` als Namen eines C-Compilers - er unterstützt standardmäßig den C99-Standard. In früheren Versionen von POSIX war `c89` als Compiler vorgeschrieben. POSIX verlangt auch, dass dieser Compiler die oben genannten Optionen `-c` und `-o` versteht.

Hinweis: Die Option `-Wall` in beiden `gcc` Beispielen weist den Compiler an, Warnungen zu fragwürdigen Konstruktionen zu drucken. `-Wall` wird dringend empfohlen. Es ist eine gute Idee , auch andere hinzufügen Warnoptionen , zB `-Wextra` .

Die Übersetzungsphasen

Seit dem C 2011-Standard, aufgeführt in *§5.1.1.2 Übersetzungsphasen* , wird die Übersetzung des Quellcodes in ein Programmabbild (z. B. die ausführbare Datei) so aufgelistet, dass sie in 8 geordneten Schritten erfolgt.

1. Die Eingabe der Quelldatei wird (falls erforderlich) dem Quellzeichensatz zugeordnet. Trigraphs werden in diesem Schritt ersetzt.
2. Fortsetzungszeilen (Zeilen, die mit `\` enden) werden mit der nächsten Zeile verbunden.
3. Der Quellcode wird in Whitespace- und Vorverarbeitungstoken geparst.
4. Der Präprozessor wird angewendet, der Anweisungen ausführt, Makros erweitert und Pragmas anwendet. Jede durch `#include` eingezogene Quelldatei durchläuft die Übersetzungsphasen 1 bis 4 (wenn nötig rekursiv). Alle Anweisungen für den Präprozessor werden dann gelöscht.
5. Quellzeichensatzwerte in Zeichenkonstanten und String-Literalen werden dem Ausführungszeichensatz zugeordnet.
6. Nebeneinander liegende String-Literale werden verkettet.
7. Der Quellcode wird in Token analysiert, aus denen die Übersetzungseinheit besteht.
8. Externe Referenzen werden aufgelöst und das Programmbild wird gebildet.

Eine Implementierung eines C-Compilers kann mehrere Schritte kombinieren, das resultierende Image muss sich jedoch so verhalten, als ob die oben genannten Schritte in der oben aufgelisteten Reihenfolge separat aufgetreten wären.

Zusammenstellung online lesen: https://riptutorial.com/de/c/topic/1337/zusammenstellung

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit C Language	4444, Abhineet, Alejandro Caro, alk, Ankush, ArturFH, Bahm, bevenson, bfd, Blacksilver, blatinox, bta, chqrlie, Community, Dair, Dan Fairaizl, Daniel Jour, Daniel Margosian, David G., David Grayson, Donald Duck, Dov Benyomin Sohacheski, Ed Cottrell, employee of the month, EOF, EsmaeelE, Frosty The DopeMan, Iskar Jarak, Jens Gustedt, John Slegers, JonasCz, Jonathan Leffler, Juan T, juleslasne, Kusalananda, Leandros, LiHRaM, Lundin, Malick, Mark Yisri, MC93, MoultoB, msohng, Myst, Narox Nox, Neal, Nemanja Boric, Nicolas Verlet, OiciTrap, P.P., PSN, Rakitić, RamenChef, Roland Illig, Ryan Hilbert, Shoe, Shog9, skrtbhtngr, sohnryang, stackptr, syb0rg, techydesigner, tlhIngan, Toby, vasili111 , Vin, Vraj Pandya
2	- Klassifizierung und Konvertierung von Zeichen	Alejandro Caro, Jonathan Leffler, Roland Illig, Toby
3	Aliasing und effektiver Typ	2501, 4386427, Jens Gustedt
4	Allgemeine C-Programmiersprachen und Entwicklerpraktiken	Chandrahas Aroori, Jonathan Leffler, Nityesh Agarwal, Shubham Agrawal
5	Arrays	2501, alk, AnArrayOfFunctions, AShelly, cdrini, cSmout, Dariusz, Elazar, Eli Sadoff, Firas Moalla, Guy, Iskar Jarak, Jasmin Solanki, Jens Gustedt, John Bollinger, Jonathan Leffler, L.V.Rao, Leandros, Liju Thomas, lordjohncena, Magisch, mhk, OznOg, Ray, Ryan Haining, Ryan Hilbert, stackptr, Toby, Waqas Bukhary
6	Atomik	Jens Gustedt
7	Aufzählungen	Alejandro Caro, alk, jasoninnn, Jens Gustedt, Jonathan Leffler, OznOg, Toby
8	Auswahlanweisungen	alk, bevenson, Blagovest Buyukliev, Faisal Mudhir, GoodDeeds, gsamaras, jxh, L.V.Rao, lordjohncena, MikeCAT, NeoR, noamgot, OznOg, P.P., Toby, tofro
9	Behauptung	2501, AShelly, Blagovest Buyukliev, bta, eush77, greatwolf, J Wu, Jens Gustedt, Jonathan Leffler, Jossi,

		jxh, Leandros, Malcolm McLean, Ryan Haining, stackptr, syb0rg, Tim Post, Toby
10	Bemerkungen	Ankush, Chandrahas Aroori, Jonathan Leffler, Toby
11	Bitfelder	alk, EvilTeach, Fantastic Mr Fox, haccks, Ishay Peled, Jens Gustedt, John Odom, Jonathan Leffler, Lundin, madD7, Paul Hutchinson, RamenChef, Rishikesh Raje, Toby, vkgade
12	Boolean	alk, Bob__, Braden Best, Chrono Kitsune, dhein, Insane, Jens Gustedt, Magisch, Mateusz Piotrowski, Peter, Toby
13	Dateien und E / A-Streams	alk, bevenson, EWoodward, haccks, iRove, Jean Vitor, Jens Gustedt, Jonathan Leffler, Jossi, Leandros, Malcolm McLean, Pedro Henrique A. Oliveira, RamenChef, reshad, Snaipe, stackptr, syb0rg, tkk, Toby, tversteeg, William Pursell
14	Datentypen	2501, alk, Blagovest Buyukliev, Firas Moalla, Jens Gustedt, Keith Thompson, Ken Y-N, Leandros, P.P., Peter, WMios
15	Deklaration vs. Definition	Ashish Ahuja, foxtrot9, Kerrek SB, Toby
16	Einschränkungen	Armali, Toby, Vality
17	Erklärungen	alk, AnArrayOfFunctions, Blacksilver, Firas Moalla, J Wu, Jens Gustedt, Jonathan Leffler, Jonathon Reinhart
18	Erstellen Sie Header-Dateien und fügen Sie sie ein	4444, Jonathan Leffler, patrick96, Sirsireesh Kodali
19	Fehlerbehandlung	Jens Gustedt, stackptr
20	Folgepunkte	2501, Armali, bta, Community, haccks, Jens Gustedt, John Bode, Toby
21	Formatierte Eingabe / Ausgabe	alk, fluter, Jonathan Leffler, Jossi, lardenn, MikeCAT, polarysekt, StardustGogeta
22	Frameworks testen	Community, EsmaeelE, Jonathan Leffler, lordjohncena, Toby, user2314737, vuko_zrno
23	Funktionsparameter	2501, Alejandro Caro, alk, Chrono Kitsune, ganesh kumar, George Stocker, Jens Gustedt, Jonathan Leffler, Leandros, MikeCAT, Minar Ashiq Tishan, P.P., RamenChef, Richard Chambers, someoneigna, syb0rg, Toby

24	Funktionszeiger	Alejandro Caro, alk, David Refaeli, Filip Allberg, hlovdal, John Burger, Leandros, Malcolm McLean, P.P., Srikar, stackptr, Toby
25	Generische Auswahl	2501, Jens Gustedt, Sun Qingyao
26	Gewerkschaften	Jossi, RamenChef, Toby, Vality
27	Häufige Fehler	abacles, Accepted Answer, alk, bevenson, Bjorn A., Chrono Kitsune, clearlight, Community, Dmitry Grigoryev, Dreamer, Dunno, FedeWar, Fred Barclay, Gavin Higham , Giorgi Moniava, hlovdal, Ishay Peled, Jeremy, John Hascall, Jonathan Leffler, Ken Y-N, Leandros, Lord Farquaad, MikeCAT, P.P., Roland Illig, rxantos, Sourav Ghosh, stackptr, Tamarous, techEmbedded, Toby, Waqas Bukhary
28	Identifier-Bereich	embedded_guy, Firas Moalla, Jean-Baptiste Yunès, Jens Gustedt, Jonathan Leffler
29	Implementierungsdefiniertes Verhalten	Jens Gustedt, John Bollinger, P.P.
30	Implizite und explizite Konvertierungen	alk, Firas Moalla, Jens Gustedt, Jeremy Thien, kdopen, Lundin, Toby
31	Initialisierung	Jonathan Leffler, Liju Thomas, P.P.
32	Inline-Montage	bevenson, EsmaeelE, Jonathan Leffler
33	Inlining	Alex, EsmaeelE, Jens Gustedt, Jonathan Leffler, Toby
34	Iterationsanweisungen / -schleifen: für, während, währenddessen	alk, GoodDeeds, Jens Gustedt, jxh, L.V.Rao, Malcolm McLean, Nagaraj, RamenChef, reshad, Toby
35	Kommandozeilenargumente	4386427, A B, alk, drov, dvhh, Jonathan Leffler, Malcolm McLean, Shog9, syb0rg, Toby, Woodrow Barlow, Yotam Salmon
36	Literale für Zahlen, Zeichen und Zeichenfolgen	Jens Gustedt, Jonathan Leffler, Klas Lindbäck, Neui, Paul92, Toby
37	Multithreading	Parham Alvani, Toby
38	Nebenwirkungen	EsmaeelE, Jonathan Leffler, L.V.Rao, madD7, RamenChef, Sirsireesh Kodali, Toby
39	Operatoren	202_accepted, 3442, alk, Amani Kilumanga, Andrea Corbelli, Bakhtiar Hasan, BenG, blatinox, cpplearner,

		Damien, Dariusz, EsmaeelE, Faisal Mudhir, Fantastic Mr Fox, Firas Moalla, gsamaras, hrs, Iwillnotexist Idonotexist , Jens Gustedt, Jonathan Leffler, kdopen, Ken Y-N, L.V.Rao, Leandros, LostAvatar, Magisch, MikeCAT, noamgot, P.P., Paul92, Peter, stackptr, Toby, Will, Wolf, Yu Hao
40	Preprozessor und Makros	Alex Garcia, alk, bevenson, bwoebi, Dariusz, DrPrItay, Erlend Graff, EsmaeelE, EvilTeach, fastlearner, Firas Moalla, gman, hashdefine, hlovdal, javac, Jens Gustedt, Jonathan Leffler, Justin, Leandros, luser droog, Madhusoodan P, Maniero, mnoronha, Nitinkumar Ambekar, P.P., Paul J. Lucas, Peter, Richard Chambers, Robert Baldyga, stackptr, Toby, v7d8dpo4
41	Prozessübergreifende Kommunikation (IPC)	CLDSEED, EsmaeelE, Jonathan Leffler, Toby
42	Signalverarbeitung	3442, alk, Dariusz, Jens Gustedt, Leandros, mirabilos
43	Speicherklassen	alk, Blagovest Buyukliev, Chrono Kitsune, greatwolf, Jean-Baptiste Yunès, Jens Gustedt, Jonathan Leffler, L.V.Rao, madD7, Neui, Nitinkumar Ambekar, P.P., Toby, tversteeg, vuko_zrno
44	Speicherverwaltung	4386427, alk, Anderson Giacomolli, Andrey Markeev, Ankush, Antti Haapala, Cullub, Daksh Gupta, dhein, dkrmr, doppelheathen, dvhh, elslooo, EOF, EsmaeelE, Firas Moalla, fluter, gdc, greatwolf, honk, Jens Gustedt, Jonathan Leffler, juleslasne, Luiz Berti, madD7, Malcolm McLean, Mark Yisri, Matthieu, Neui, P.P., Paul Campbell, Paul V, reflective_mind, Seth, Srikar, stackptr, syb0rg, Tamarous, tbodt, the sudhakar, Toby, tofro, Vivek S, vuko_zrno, Wyzard
45	Sprunganweisungen	alk, Jens Gustedt, Jonathan Leffler, lordjohncena, Malcolm McLean, Sourav Ghosh, syb0rg, Toby
46	Standard Math	Alejandro Caro, alk, Blagovest Buyukliev, immerhart, Jonathan Leffler, manav m-n, Toby
47	Structs	alk, Chrono Kitsune, Damien, Elazar, EsmaeelE, Faisal Mudhir, Firas Moalla, gmug, jasoninnn, Jens Gustedt, Jonathan Leffler, Jossi, kamoroso94, Madhusoodan P, OznOg, Paul Kramme, PhotometricStereo, RamenChef, Toby, Vality
48	Strukturpolsterung und Verpackung	EsmaeelE, Jarrod Dixon, Jedi, Jesferman, Jonathan Leffler, Liju Thomas, MayeulC, tilz0R

49	Themen (einheimisch)	alk, Jens Gustedt, P.P.
50	Typ Qualifiers	alk, Blagovest Buyukliev, Jens Gustedt, Jesferman, madD7, tversteeg
51	Typedef	Buser, Chandrahas Aroori, GoodDeeds, Jonathan Leffler , mame98, PhotometricStereo, Stephen Leppik, Toby
52	Übergeben Sie 2D-Arrays an Funktionen	deamentiaemundi, Malcolm McLean, Shrinivas Patgar, Toby
53	Undefiniertes Verhalten	2501, Abhineet, Aleksi Torhamo, alk, Antti Haapala, Armali, Ben Steffan, blatinox, bta, BurnsBA, caf, Christoph, Cody Gray, Community, cshu, DaBler, Daniel Jour, DarkDust, FedeWar, Firas Moalla, Giorgi Moniava, gsamaras, haccks, hmijail, honk, Jacob H, Jean-Baptiste Yunès, Jens Gustedt, John, John Bollinger, Jonathan Leffler, Kamiccolo, Leandros, Lundin, Magisch, Mark Yisri, Martin, MikeCAT, Nemanja Boric, P.P., Peter, Roland Illig, TimF, Toby, tversteeg, user45891, Vasfed, void
54	Valgrind	abacles, alk, Ankush, Chandrahas Aroori, Devansh Tandon, drov, Firas Moalla, J F, Jonathan Leffler, vasili111
55	Variable Argumente	2501, Blacksilver, eush77, Jean-Baptiste Yunès, Jonathan Leffler, Leandros, mirabilos, syb0rg, Toby
56	Verknüpfte Listen	4386427, alk, Andrea Biondo, bevenson, iRove, Jonathan Leffler, Jossi, Leandros, Mateusz Piotrowski, Ryan, Toby
57	X-Makros	Cimbali, Jens Gustedt, John Bollinger, Leandros, MD XF, mpromonet, poolie, RamenChef, technosaurus, templatetypedef, Toby
58	Zeichenfolge mit mehreren Zeichen	Jonathan Leffler, PassionInfinite, Toby
59	Zeichenketten	4386427, alk, Amani Kilumanga, Andrey Markeev, bevenson, catalogue_number, Chris Sprague, Chrono Kitsune, Cody Gray, Damien, Daniel, depperm, dylanweber, FedeWar, Firas Moalla, haccks, Ishay Peled , jasoninnn, Jean-Baptiste Yunès, Jens Gustedt, John Bollinger, Jonathan Leffler, Leandros, Malcolm McLean, mantal, MikeCAT, P.P., Purag, Roland Illig, stackptr, still_learning, syb0rg, Toby, vasili111, Waqas Bukhary, Wolf, Wyzard, Алексей Неудачин

60	Zeiger	0xEDD1E, alk, Altece, Amani Kilumanga, Andrey Markeev, Ankush, Antti Haapala, Ashish Ahuja, Bjorn A., bruno, bta, chqrlie, Courtney Pattison, Dair, Daniel Porteous, David G., dhein, dkrmr, Don't You Worry Child, e.jahandar, elslooo, EOF, erebos, Faisal Mudhir, Fantastic Mr Fox, FedeWar, Firas Moalla, fluter, foxtrot9, Gavin Higham, gdc, Giorgi Moniava, gsamaras, haccks, haltode, Harry Johnston, Hemant Kumar, honk, Jens Gustedt, Jonathan Leffler, Jonnathan Soares, Josh de Kock, jpX, L.V.Rao, LaneL, Leandros, Luiz Berti, Malcolm McLean, Matthieu, Michael Fitzpatrick, MikeCAT, Neui, Nitinkumar Ambekar, OiciTrap, P.P., Pbd, Peter, RamenChef, raymai97, Rohan, Sergey, Shahbaz, signal, slugonamission, solomonope, someoneigna, Spidey, Srikar, stackptr, syb0rg, tbodt, the sudhakar, thndrwrks, Toby, Vality, vijay kant sharma, Vivek S, Wyzard, xhienne, Алексей Неудачин
61	Zufallszahlengenerierung	dylanweber, ganchito55, haccks, hexwab, Jonathan Leffler, Leandros, Malcolm McLean, MikeCAT, Toby
62	Zusammengesetzte Literale	alk, haccks, Jens Gustedt, Kerrek SB
63	Zusammenstellung	alk, Amani Kilumanga, bevenson, Blacksilver, Firas Moalla, haccks, Ishay Peled, Jean-Baptiste Yunès, Jens Gustedt, Jonathan Leffler, Jossi, jxh, MC93, MikeCAT, nathanielng, P.P., Qrchack, R. Joiny, syb0rg, Toby, tofro, Turtle, Vraj Pandya, Алексей Неудачин