EBook Gratuito

APPRENDIMENTO

C Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#c

Sommario

Di

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.

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

Capitolo 1: Iniziare con C Language

Osservazioni

C è un linguaggio di programmazione per computer di uso generale, imperativo, che supporta programmazione strutturata, ambito variabile lessicale e ricorsione, mentre un sistema di tipo statico impedisce molte operazioni indesiderate. In base alla progettazione, C fornisce costrutti che si adattano in modo efficiente alle istruzioni tipiche della macchina e pertanto ha trovato un uso duraturo in applicazioni che erano state precedentemente codificate in linguaggio assembly, inclusi i sistemi operativi, nonché vari software applicativi per computer che vanno dai supercomputer ai sistemi embedded .

Nonostante le sue capacità di basso livello, il linguaggio è stato progettato per incoraggiare la programmazione multipiattaforma. Un programma C conforme agli standard e portabile può essere compilato per una vasta gamma di piattaforme e sistemi operativi con poche modifiche al codice sorgente. Il linguaggio è diventato disponibile su una vasta gamma di piattaforme, dai microcontrollori embedded ai supercomputer.

C è stato originariamente sviluppato da Dennis Ritchie tra il 1969 e il 1973 presso i Bell Labs e utilizzato per ri-implementare i sistemi operativi Unix . Da allora è diventato uno dei linguaggi di programmazione più utilizzati di tutti i tempi, con compilatori C di vari fornitori disponibili per la maggior parte delle architetture e dei sistemi operativi esistenti.

Compilatori comuni

Il processo per compilare un programma C differisce tra compilatori e sistemi operativi. La maggior parte dei sistemi operativi viene spedita senza un compilatore, quindi dovrai installarne uno. Alcune scelte comuni dei compilatori sono:

- GCC, la raccolta del compilatore GNU
- clang: un front-end di famiglia in linguaggio C per LLVM
- MSVC, strumenti di compilazione di Microsoft Visual C / C ++

I seguenti documenti dovrebbero darti una buona panoramica su come iniziare ad usare alcuni dei più comuni compilatori:

- Iniziare con Microsoft Visual C
- Iniziare con GCC

Supporto per la versione C del compilatore

Si noti che i compilatori hanno vari livelli di supporto per lo standard C con molti C99 ancora non completamente supportati. Ad esempio, a partire dalla versione 2015, MSVC supporta gran parte del C99 ma ha ancora alcune importanti eccezioni per il supporto del linguaggio stesso (ad esempio la pre-elaborazione sembra non conforme) e per la libreria C (ad esempio `<tgmath.h>`),

né documentano necessariamente le loro "scelte dipendenti dall'implementazione". Wikipedia ha una tabella che mostra il supporto offerto da alcuni compilatori popolari.

Alcuni compilatori (in particolare GCC) hanno offerto o continuano a offrire *estensioni del compilatore* che implementano funzionalità aggiuntive che i produttori di compilatori ritengono necessarie, utili o che potrebbero diventare parte di una versione C futura, ma che attualmente non fanno parte di alcun standard C. Poiché queste estensioni sono specifiche del compilatore, possono essere considerate non compatibili per la compatibilità e gli sviluppatori di compilatori possono rimuoverle o modificarle nelle versioni successive del compilatore. L'uso di tali estensioni può generalmente essere controllato dai flag del compilatore.

Inoltre, molti sviluppatori hanno compilatori che supportano solo versioni specifiche di C imposte dall'ambiente o dalla piattaforma a cui sono indirizzate.

Se si seleziona un compilatore, si consiglia di scegliere un compilatore che abbia il miglior supporto per l'ultima versione di C consentita per l'ambiente di destinazione.

Stile del codice (off-topic qui):

Poiché lo spazio bianco è insignificante in C (cioè, non influenza il funzionamento del codice), i programmatori utilizzano spesso uno spazio bianco per rendere il codice più facile da leggere e comprendere, questo è chiamato lo *stile* del *codice* . È un insieme di regole e linee guida utilizzate durante la scrittura del codice sorgente. Riguarda argomenti come il modo in cui le linee dovrebbero essere rientrate, se devono essere usati spazi o tabulazioni, come posizionare le parentesi, come gli spazi dovrebbero essere usati attorno agli operatori e le parentesi, come le variabili dovrebbero essere nominate e così via.

Lo stile del codice non è coperto dallo standard ed è principalmente basato sull'opinione pubblica (diverse persone trovano stili diversi più facili da leggere), in quanto tale è generalmente considerato off-topic su SO. Il consiglio imperativo sullo stile nel proprio codice è che la coerenza è fondamentale: scegliere o creare uno stile e attenersi ad esso. Basti pensare che ci sono vari stili di nome nell'uso comune che vengono spesso scelti dai programmatori piuttosto che creare il proprio stile.

Alcuni stili di rientro comuni sono: stile K & R, stile Allman, stile GNU e così via. Alcuni di questi stili hanno varianti diverse. Allman, ad esempio, è usato sia come Allman normale che come variante popolare, Allman-8. Informazioni su alcuni degli stili popolari possono essere trovate su Wikipedia . Tali nomi di stile sono presi dagli standard che gli autori o le organizzazioni pubblicano spesso per l'uso da parte di molte persone che contribuiscono al loro codice, in modo che tutti possano facilmente leggere il codice quando conoscono lo stile, come la guida alla formattazione GNU che fa parte di il documento degli standard di codifica GNU .

Alcune convenzioni di denominazione comuni sono: UpperCamelCase, lowerCamelCase, lower_case_with_underscore, ALL_CAPS, ecc. Questi stili sono combinati in vari modi per l'uso con oggetti e tipi diversi (ad esempio, le macro spesso usano lo stile ALL_CAPS)

Lo stile K & R è generalmente raccomandato per l'uso all'interno della documentazione SO, mentre gli stili più esoterici, come Pico, sono scoraggiati.

Librerie e API non coperte dallo standard C (e quindi non trattate qui):

- API POSIX (che copre ad esempio PThread , socket , segnali)

Versioni

Versione	Standard	Data di pubblicazione
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

Ciao mondo

Per creare un semplice programma C che stampi *"Hello, World"* sullo schermo, usa un editor di testo per creare un nuovo file (ad es. `hello.c` - l'estensione del file deve essere `.c`) contenente il seguente codice sorgente:

Ciao C

```
#include <stdio.h>

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

Demo dal vivo su Coliru

Diamo un'occhiata a questo semplice programma riga per riga

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

Questa riga indica al compilatore di includere il contenuto del file di intestazione della libreria standard `stdio.h` nel programma. Le intestazioni sono in genere file contenenti dichiarazioni di funzioni, macro e tipi di dati e devi includere il file di intestazione prima di utilizzarli. Questa riga include `stdio.h` quindi può chiamare la funzione `puts()` .

Vedi di più sulle intestazioni.

```
int main(void)
```

Questa riga avvia la definizione di una funzione. Indica il nome della funzione (`main`), il tipo e il numero di argomenti che si aspetta (`void` , che significa nessuno) e il tipo di valore restituito da questa funzione (`int`). L'esecuzione del programma inizia nella funzione `main()` .

```
{
    …
}
```

Le parentesi graffe vengono utilizzate in coppia per indicare dove inizia e termina un blocco di codice. Possono essere utilizzati in molti modi, ma in questo caso indicano dove inizia e finisce la funzione.

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

Questa linea chiama la funzione `puts()` per l'output del testo sullo standard output (lo schermo, per impostazione predefinita), seguito da una nuova riga. La stringa da produrre è inclusa tra parentesi.

`"Hello, World"` è la stringa che verrà scritta sullo schermo. In C, ogni valore letterale di stringa deve essere compreso tra virgolette `"…"` .

Vedi di più sulle stringhe.

Nei programmi C, ogni istruzione deve essere terminata da un punto e virgola (cioè `;`).

```
    return 0;
```

Quando abbiamo definito `main()` , lo abbiamo dichiarato come funzione che restituisce un `int` , il che significa che deve restituire un intero. In questo esempio, restituiamo il valore intero 0, che viene utilizzato per indicare che il programma è stato chiuso correttamente. Dopo il `return 0;` dichiarazione, il processo di esecuzione terminerà.

Modifica del programma

Gli editor di testo semplici includono `vim` o `gedit` su Linux o `Notepad` su Windows. Gli editor multipiattaforma includono anche `Visual Studio Code` o `Sublime Text` .

L'editor deve creare file di solo testo, non RTF o altri formati.

Compilare ed eseguire il programma

Per eseguire il programma, questo file sorgente (`hello.c`) deve prima essere compilato in un file eseguibile (ad es. `hello` su sistema Unix / Linux o `hello.exe` su Windows). Questo viene fatto usando un compilatore per il linguaggio C.

Vedi di più sulla compilazione

Compilare usando GCC

GCC (GNU Compiler Collection) è un compilatore C ampiamente utilizzato. Per usarlo, apri un terminale, usa la riga di comando per navigare fino alla posizione del file sorgente e quindi eseguire:

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

Se non si trovano errori nel codice sorgente (`hello.c`), il compilatore creerà un **file binario** , il cui nome è dato dall'argomento all'opzione `-o` riga di comando (`hello`). Questo è il file eseguibile finale.

Possiamo anche utilizzare le opzioni di avviso `-Wall -Wextra -Werror` , che aiutano a identificare i problemi che possono causare il fallimento del programma o produrre risultati imprevisti. Non sono necessari per questo semplice programma, ma questo è il modo di aggiungerli:

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

Usando il compilatore clang

Per compilare il programma usando `clang` puoi usare:

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

In base alla progettazione, le opzioni della riga di comando `clang` sono simili a quelle di GCC.

Utilizzo del compilatore Microsoft C dalla riga di comando

Se si utilizza il compilatore Microsoft `cl.exe` su un sistema Windows che supporta Visual Studio e se tutte le variabili di ambiente sono impostate, questo esempio C può essere compilato utilizzando il seguente comando che produrrà un file eseguibile `hello.exe` all'interno della directory in cui viene eseguito il comando in (Ci sono opzioni di avviso come `/W3` per `cl` , grosso modo analogo a `-Wall` ecc. Per GCC o clang).

```
cl hello.c
```

Esecuzione del programma

Una volta compilato, il file binario può quindi essere eseguito digitando `./hello` nel terminale. Al momento dell'esecuzione, il programma compilato stamperà `Hello, World` , seguito da una nuova riga, al prompt dei comandi.

Originale "Hello, World!" in K & R C

Quello che segue è l'originale "Hello, World!" programma tratto dal libro The C Programming Language di Brian Kernighan e Dennis Ritchie (Ritchie è stato lo sviluppatore originale del linguaggio di programmazione C presso i Bell Labs), denominato "K & R":

K & R

```
#include <stdio.h>

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

Si noti che il linguaggio di programmazione C non era standardizzato al momento della stesura della prima edizione di questo libro (1978) e che questo programma probabilmente non verrà compilato sulla maggior parte dei compilatori moderni a meno che non venga loro richiesto di accettare il codice C90.

Questo primissimo esempio nel libro K & R è ora considerato di scarsa qualità, in parte perché manca un tipo di ritorno esplicito per `main()` e in parte perché manca un'istruzione `return` . La seconda edizione del libro è stata scritta per il vecchio standard C89. In C89, il tipo di `main` predefinito su `int` , ma l'esempio K & R non restituisce un valore definito per l'ambiente. Negli standard C99 e successivi, è richiesto il tipo restituito, ma è sicuro tralasciare l'istruzione `return` di `main` (e solo `main`), a causa di un caso speciale introdotto con C99 5.1.2.2.3 - equivale a restituire 0 , che indica il successo.

La forma raccomandata e più portabile di `main` per i sistemi ospitati è `int main (void)` quando il programma non usa alcun argomento della riga di comando, o `int main(int argc, char **argv)` quando il programma usa gli argomenti della riga di comando.

C90 §5.1.2.2.3 **Terminazione del programma**

> Un ritorno dalla chiamata iniziale alla funzione `main` equivale a chiamare la funzione di `exit` con il valore restituito dalla funzione `main` come argomento. Se la funzione `main` esegue un ritorno che non specifica alcun valore, lo stato di terminazione restituito all'ambiente host non è definito.

C90 §6.6.6.4 **Il `return` dichiarazione**

> Se viene eseguita un'istruzione `return` senza un'espressione e il valore della chiamata della funzione viene utilizzato dal chiamante, il comportamento non è definito. Raggiungere il `}` che termina una funzione equivale all'esecuzione di un'istruzione `return` senza un'espressione.

C99 §5.1.2.2.3 **Terminazione del programma**

> Se il tipo restituito della funzione `main` è un tipo compatibile con `int` , un ritorno dalla chiamata iniziale alla funzione `main` equivale a chiamare la funzione di `exit` con il valore restituito dalla funzione `main` come argomento; raggiungendo il `}` che termina la funzione `main` restituisce un valore di 0. Se il tipo restituito non è compatibile con `int` , lo stato di terminazione restituito all'ambiente host non è specificato.

Leggi Iniziare con C Language online: https://riptutorial.com/it/c/topic/213/iniziare-con-c-language

Capitolo 2: - classificazione e conversione dei caratteri

Examples

Classificazione dei caratteri letti da un flusso

```
#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;
}
```

La funzione `classify` legge i caratteri da uno stream e conta il numero di spazi, caratteri alfanumerici e di punteggiatura. Evita molte insidie.

- Durante la lettura di un carattere da uno stream, il risultato viene salvato come `int` , poiché altrimenti ci sarebbe un'ambiguità tra la lettura di `EOF` (il marker di fine del file) e un carattere con lo stesso pattern di bit.
- Le funzioni di classificazione dei caratteri (ad esempio `isspace`) si aspettano che il loro argomento *sia rappresentabile come un* `unsigned char` *o il valore della macro* `EOF` . Poiché questo è esattamente ciò che restituisce il `fgetc` , non c'è bisogno di conversione qui.
- Il valore di ritorno delle funzioni di classificazione dei caratteri distingue solo tra zero (significato `false`) e non zero (che significa `true`). Per il conteggio del numero di occorrenze, questo valore deve essere convertito in 1 o 0, operazione eseguita dalla doppia negazione, `!!` .

Classificazione dei caratteri da una stringa

```
#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;
}
```

La `classify` la funzione esamina tutti i caratteri di una stringa e conta il numero di spazi, caratteri alfanumerici e di punteggiatura. Evita molte insidie.

- Le funzioni di classificazione dei caratteri (ad esempio `isspace`) si aspettano che il loro argomento *sia rappresentabile come un* `unsigned char` *o il valore della macro* `EOF` .
- L'espressione `*p` è di tipo `char` e deve quindi essere convertita per corrispondere alla dicitura precedente.
- Il tipo di `char` è definito come equivalente al `signed char` o al `unsigned char` .
- Quando `char` è equivalente a `unsigned char` , non ci sono problemi, dal momento che ogni possibile valore del tipo `char` è rappresentabile come `unsigned char` .
- Quando `char` è equivalente al `signed char` , deve essere convertito in `unsigned char` prima di essere passato alle funzioni di classificazione dei caratteri. E sebbene il valore del personaggio possa cambiare a causa di questa conversione, questo è esattamente ciò che si aspettano queste funzioni.
- Il valore di ritorno delle funzioni di classificazione dei caratteri distingue solo tra zero (significato `false`) e non zero (che significa `true`). Per il conteggio del numero di occorrenze, questo valore deve essere convertito in 1 o 0, operazione eseguita dalla doppia negazione, `!!` .

introduzione

L'intestazione `ctype.h` è una parte della libreria C standard. Fornisce funzioni per la classificazione e la conversione di caratteri.

Tutte queste funzioni accettano un parametro, un `int` che deve essere *EOF* o rappresentabile come un carattere senza segno.

I nomi delle funzioni di classificazione sono preceduti da "è". Ognuno restituisce un valore intero diverso da zero (VERO) se il carattere passato ad esso soddisfa la condizione correlata. Se la condizione non è soddisfatta, la funzione restituisce un valore zero (FALSE).

Queste funzioni di classificazione funzionano come mostrato, assumendo la locale predefinita C:

```
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. */
```

Ci sono due funzioni di conversione. Questi sono nominati usando il prefisso 'a'. Queste funzioni riprendono lo stesso argomento di quelle sopra. Tuttavia, il valore restituito non è uno zero semplice o diverso da zero ma l'argomento passato è cambiato in qualche modo.

Queste funzioni di conversione funzionano come mostrato, assumendo la locale predefinita C:

```
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);
```

Le informazioni sottostanti sono citate da cplusplus.com che mappano come l'insieme ASCII di 127 caratteri originale è considerato da ciascuna delle funzioni di tipo di classificazione (a • indica che la funzione restituisce un valore diverso da zero per quel carattere)

Valori ASCII	personaggi	iscntrl	è vuoto	isspace	isupper	è più basso	isalpha	isdi
0x00 .. 0x08	NUL, (altri codici di controllo)	•						
0x09	scheda ('\ t')	•	•	•				

Valori ASCII	personaggi	iscntrl	è vuoto	isspace	isupper	è più basso	isalpha	isdi
0x0A .. 0x0D	(Codici di controllo dello spazio bianco: '\ f', '\ v', '\ n', '\ r')	•		•				
0x0E .. 0x1F	(altri codici di controllo)	•						
0x20	spazio (' ')		•	•				
0x21 .. 0x2F	! "# $% & '() * +, -. /							
0x30 .. 0x39	0123456789							•
0x3a .. 0x40	:;? <=> @							
0x41 .. 0x46	A B C D E F				•		•	
0x47 .. 0x5A	GHIJKLMNOPQRSTUVWXYZ				•		•	
0x5B .. 0x60	[] ^ _ `							
0x61 .. 0x66	a B c D e F					•	•	
0x67 .. 0x7A	ghijklmnopqrstuvwxyz					•	•	
0x7B .. 0x7E	{} ~ Bar							
0x7F	(DEL)	•						

Leggi - classificazione e conversione dei caratteri online: https://riptutorial.com/it/c/topic/6846/-ctype-h----classificazione-e-conversione-dei-caratteri

Capitolo 3: Aliasing e tipo effettivo

Osservazioni

Le violazioni delle regole di aliasing e la violazione del tipo effettivo di un oggetto sono due cose diverse e non devono essere confuse.

- *L'aliasing* è la proprietà di due puntatori `a` e `b` che si riferiscono allo stesso oggetto, cioè `a == b` .
- Il *tipo effettivo* di un oggetto dati viene utilizzato da C per determinare quali operazioni possono essere eseguite su quell'oggetto. In particolare, il tipo effettivo viene utilizzato per determinare se due puntatori possono alias l'un l'altro.

Aliasing può essere un problema di ottimizzazione, perché cambiando l'oggetto i puntatore, `a` esempio, può cambiare l'oggetto che è visibile attraverso l'altro puntatore, `b` . Se il compilatore C dovesse assumere che i puntatori potrebbero sempre essere alias l'un l'altro, indipendentemente dal loro tipo e provenienza, molte opportunità di ottimizzazione andrebbero perse e molti programmi sarebbero più lenti.

Le rigide regole di aliasing di C si riferiscono ai casi in cui il compilatore *può assumere* che gli oggetti (o non) si alias l'un l'altro. Ci sono due regole pratiche che dovresti sempre avere in mente per i puntatori di dati.

> Salvo diversa indicazione, due puntatori con lo stesso tipo di base possono essere alias.
>
> Due puntatori con un tipo di base diverso non possono essere pseudonimi, a meno che almeno uno dei due tipi sia un tipo di carattere.

Qui il *tipo di base* significa che mettiamo da parte le qualifiche di tipo come `const` , ad esempio se `a` è `double*` e `b` è `const double*` , il compilatore *deve* generalmente assumere che un cambiamento di `*a` possa cambiare `*b` .

La violazione della seconda regola può avere risultati catastrofici. Qui violare la rigida regola di aliasing significa che si presentano due puntatori `a` e `b` di tipo diverso al compilatore che in realtà puntano allo stesso oggetto. Il compilatore quindi può sempre presumere che i due punti a oggetti diversi, e non aggiornerà la sua idea di `*b` se hai cambiato l'oggetto tramite `*a` .

Se lo fai, il comportamento del tuo programma diventa indefinito. Pertanto, C pone restrizioni piuttosto severe alle conversioni del puntatore al fine di evitare che tale situazione si verifichi accidentalmente.

> A meno che il tipo di origine o di destinazione sia `void` , tutte le conversioni di puntatore tra i puntatori con diverso tipo di base devono essere *esplicite* .

O in altre parole, hanno bisogno di un *cast* , a meno che non si faccia una conversione che

aggiunge solo un qualificatore come `const` al tipo di destinazione.

Evitare le conversioni puntatore in generale e le cas in particolare ti protegge dai problemi di aliasing. A meno che tu non abbia davvero bisogno di loro, e questi casi sono molto speciali, dovresti evitarli il più possibile.

Examples

Non è possibile accedere ai tipi di caratteri tramite tipi non di caratteri.

Se un oggetto è definito con statico, thread o durata della memorizzazione automatica e ha un tipo di carattere: `char` , `unsigned char` o `signed char` , potrebbe non essere accessibile da un tipo non di caratteri. Nell'esempio seguente un array di `char` viene reinterpretato come il tipo `int` e il comportamento non è definito su ogni dereferenziazione del puntatore `int b` .

```
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;
}
```

Non è definito perché viola la regola del "tipo effettivo", non è possibile accedere a nessun oggetto dati di tipo efficace tramite un altro tipo che non è un tipo di carattere. Poiché l'altro tipo qui è `int` , questo non è permesso.

Anche se l'allineamento e le dimensioni del puntatore sarebbero note per adattarsi, questo non sarebbe esente da questa regola, il comportamento sarebbe ancora indefinito.

Ciò significa in particolare che non esiste alcun modo in standard C di riservare un oggetto buffer di tipo carattere che può essere utilizzato attraverso puntatori con tipi diversi, poiché si utilizzerà un buffer ricevuto da `malloc` o funzione simile.

Un modo corretto per raggiungere lo stesso obiettivo dell'esempio precedente sarebbe utilizzare un `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;
}
```

Qui, l' `union` assicura che il compilatore sappia dall'inizio che il buffer può essere consultato attraverso diverse viste. Questo ha anche il vantaggio che ora il buffer ha una "vista" `ai` che è già di tipo `int` e non è necessaria alcuna conversione del puntatore.

Tipo efficace

Il *tipo effettivo* di un oggetto dati è l'ultimo tipo di informazione ad esso associato, se presente.

```
// 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);
```

Osserva che per quest'ultimo non era necessario avere un puntatore `uint32_t*` su quell'oggetto. Il fatto che abbiamo copiato un altro oggetto `uint32_t` è sufficiente.

Violare le rigide regole di aliasing

Nel seguente codice supponiamo per semplicità che `float` e `uint32_t` abbiano la stessa dimensione.

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

`u` e `f` hanno un diverso tipo di base, e quindi il compilatore può assumere che punti a oggetti diversi. Non c'è alcuna possibilità che `*f` avrebbe potuto cambiare tra i due inizializzazioni di `a` e `b` , e quindi il compilatore può ottimizzare il codice per qualcosa di equivalente a

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

Cioè, la seconda operazione di caricamento di `*f` può essere completamente ottimizzata.

Se chiamiamo questa funzione "normalmente"

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

tutto va bene e qualcosa di simile

    4 dovrebbe essere uguale a 4

è stampato Ma se imbrogliamo e passiamo lo stesso puntatore, dopo averlo convertito,

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

violiamo la rigida regola di aliasing. Quindi il comportamento diventa indefinito. L'output potrebbe essere come sopra, se il compilatore ha ottimizzato il secondo accesso, o qualcosa di completamente diverso, e così il tuo programma finisce in uno stato completamente inaffidabile.

limitare la qualifica

Se abbiamo due argomenti puntatori dello stesso tipo, il compilatore non può assumere alcuna ipotesi e dovrà sempre presumere che la modifica a `*e` potrebbe cambiare `*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);
```

tutto va bene e qualcosa di simile

> è 4 uguale a 4?

è stampato Se passiamo lo stesso puntatore, il programma farà ancora la cosa giusta e stamperà

> è 4 uguale a 22?

Questo può rivelarsi inefficiente, se *sappiamo* da alcune informazioni esterne che `e` e `f` non puntano mai allo stesso oggetto dati. Possiamo riflettere questa conoscenza aggiungendo qualificatori `restrict` ai parametri del puntatore:

```
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);
}
```

Quindi il compilatore può sempre supporre che `e` e `f` puntino a oggetti diversi.

Modifica dei byte

Una volta che un oggetto ha un tipo efficace, non si deve tentare di modificarlo tramite un puntatore di un altro tipo, a meno che quell'altro tipo sia un tipo di carattere, `char` , `signed char` o `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);
}
```

Questo è un programma valido che stampa

> un ora ha il valore 707406378

Funziona perché:

- L'accesso viene effettuato ai singoli byte visualizzati con `unsigned char` pertanto ogni modifica è ben definita.
- Le due viste dell'oggetto, tramite `a` e attraverso `*ap` , alias, ma poiché `ap` è un puntatore a un tipo di carattere, la regola di aliasing rigorosa non si applica. Quindi il compilatore deve assumere che il valore di `a` possa essere stato modificato nel ciclo `for` . Il valore modificato di `a` deve essere costruito dai byte che sono stati modificati.
- Il tipo di `a` , `uint32_t` non ha bit di riempimento. Tutti i suoi bit della rappresentazione contano per il valore, qui `707406378` , e non può esserci alcuna rappresentazione di trap.

Leggi Aliasing e tipo effettivo online: https://riptutorial.com/it/c/topic/1301/aliasing-e-tipo-effettivo

Capitolo 4: Ambito dell'identificatore

Examples

Block Scope

Un identificatore ha scope di blocco se la relativa dichiarazione appare all'interno di un blocco (si applica la dichiarazione dei parametri nella definizione della funzione). L'oscilloscopio termina alla fine del blocco corrispondente.

Nessuna entità diversa con lo stesso identificatore può avere lo stesso ambito, ma gli ambiti possono sovrapporsi. In caso di ambiti sovrapposti, l'unico visibile è quello dichiarato nello scope più interno.

```
#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
```

Funzione Prototipo Ambito

```
#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;
}
```

Si noti che si ottengono messaggi di errore enigmatici se si introduce un nome di tipo in un prototipo:

```
int function(struct whatever *arg);

struct whatever
{
    int a;
    // ...
};

int function(struct whatever *arg)
{
    return arg->a;
}
```

Con GCC 6.3.0, questo codice (file di origine `dc11.c`) produce:

```
$ 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
$
```

Posiziona la definizione della struttura prima della dichiarazione della funzione o aggiungi `struct whatever;` come una riga prima della dichiarazione della funzione, e non ci sono problemi. Non dovresti introdurre nuovi nomi di tipi in un prototipo di funzione perché non c'è modo di usare quel tipo, e quindi non c'è modo di definire o usare quella funzione.

Scopo del file

```
#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;
}
```

Ambito di funzione

L'ambito della funzione è lo scopo speciale per le **etichette** . Ciò è dovuto alla loro proprietà insolita. **Un'etichetta** è visibile attraverso l'intera funzione che è definita e si può saltare (usando l' `label goto` dell'istruzione) ad essa da qualsiasi punto nella stessa funzione. Anche se non utile, il seguente esempio illustra il punto:

```
#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` può sembrare definito *all'interno* del blocco `if` , come nel caso di `i` che scope è il blocco, ma non lo è. È visibile nell'intera funzione mentre l'istruzione `goto INSIDE;` illustra. Quindi non ci possono essere due etichette con lo stesso identificatore in una singola funzione.

Un possibile utilizzo è il seguente schema per realizzare corrette puliture complesse delle risorse allocate:

```
#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);
}
```

Etichette come `CLEANUP1` e `CLEANUP2` sono identificatori speciali che si comportano in modo diverso da tutti gli altri identificativi. Sono visibili da qualsiasi parte all'interno della funzione, anche in posizioni che sono eseguite prima dell'istruzione etichettata, o anche in luoghi che non potrebbero mai essere raggiunti se nessuno dei `goto` viene eseguito. Le etichette sono spesso scritte in lettere minuscole e non maiuscole.

Leggi Ambito dell'identificatore online: https://riptutorial.com/it/c/topic/1804/ambito-dell-identificatore

Capitolo 5: Argomenti della riga di comando

Sintassi

- int main (int argc, char * argv [])

Parametri

Parametro	Dettagli
argc	argomento count - inizializzato sul numero di argomenti separati dallo spazio dati al programma dalla riga di comando e sul nome del programma stesso.
argv	vettore di argomento: inizializzato su una matrice di puntatori di `char` (stringhe) contenenti gli argomenti (e il nome del programma) forniti sulla riga di comando.

Osservazioni

Il programma AC in esecuzione in un "ambiente ospitato" (il tipo normale - al contrario di un "ambiente indipendente") deve avere una funzione `main` . È tradizionalmente definito come:

```
int main(int argc, char *argv[])
```

Si noti che `argv` può anche essere, e molto spesso, definito come `char **argv` ; il comportamento è lo stesso. Inoltre, i nomi dei parametri possono essere modificati perché sono solo variabili locali all'interno della funzione, ma `argc` e `argv` sono convenzionali e dovresti usare questi nomi.

Per le funzioni `main` cui il codice non utilizza argomenti, utilizzare `int main(void)` .

Entrambi i parametri sono inizializzati all'avvio del programma:

- `argc` è inizializzato sul numero di argomenti separati dallo spazio dati al programma dalla riga di comando e sul nome del programma stesso.
- `argv` è un array di `char` -pointers (stringhe) contenenti gli argomenti (e il nome del programma) che è stato fornito sulla riga di comando.
- alcuni sistemi espandono gli argomenti della riga di comando "nella shell", altri no. Su Unix se l'utente digita `myprogram *.txt` il programma riceverà un elenco di file di testo; su Windows riceverà la stringa " `*.txt` ".

Nota: prima di utilizzare `argv` , potrebbe essere necessario controllare il valore di `argc` . In teoria, `argc` potrebbe essere `0` , e se `argc` è zero, allora non ci sono argomenti e `argv[0]` (equivalente a `argv[argc]`) è un puntatore nullo. Sarebbe un sistema insolito con un ambiente ospitato se si è verificato questo problema. Allo stesso modo, è possibile, sebbene molto insolito, che non ci siano informazioni sul nome del programma. In tal caso, `argv[0][0] == '\0'` - il nome del programma

potrebbe essere vuoto.

Supponiamo di iniziare il programma in questo modo:

```
./some_program abba banana mamajam
```

Quindi `argc` è uguale a `4` e gli argomenti della riga di comando:

- `argv[0]` punta a `"./some_program"` (il nome del programma) se il nome del programma è disponibile dall'ambiente host. Altrimenti una stringa vuota `""` .
- `argv[1]` indica `"abba"` ,
- `argv[2]` indica `"banana"` ,
- `argv[3]` indica `"mamajam"` ,
- `argv[4]` contiene il valore `NULL` .

Vedi anche What should `main()` return in C e C ++ per le virgolette complete dallo standard.

Examples

Stampa degli argomenti della riga di comando

Dopo aver ricevuto gli argomenti, puoi stamparli come segue:

```
int main(int argc, char **argv)
{
    for (int i = 1; i < argc; i++)
    {
        printf("Argument %d: [%s]\n", i, argv[i]);
    }
}
```

Gli appunti

1. Il parametro `argv` può anche essere definito come `char *argv[]` .
2. `argv[0]` *può* contenere il nome del programma stesso (a seconda di come è stato eseguito il programma). Il primo argomento di riga di comando "reale" è in `argv[1]` , e questo è il motivo per cui la variabile di loop `i` è inizializzata su 1.
3. Nell'istruzione print, puoi usare `*(argv + i)` invece di `argv[i]` - valuta la stessa cosa, ma è più dettagliato.
4. Le parentesi quadre attorno al valore dell'argomento aiutano a identificare l'inizio e la fine. Questo può essere inestimabile se ci sono spazi vuoti finali, newline, ritorni a capo o altri caratteri oddball nell'argomento. Alcune varianti di questo programma sono uno strumento utile per il debug degli script di shell in cui è necessario capire cosa contiene effettivamente l'elenco degli argomenti (sebbene esistano alternative di shell semplici che sono quasi equivalenti).

Stampa gli argomenti in un programma e converti in valori interi

Il seguente codice stamperà gli argomenti per il programma e il codice tenterà di convertire ogni argomento in un numero (a `long`):

```
#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;
}
```

RIFERIMENTI:

- strtol () restituisce un valore errato
- Uso corretto di strtol

Usare gli strumenti GNU getopt

Le opzioni della riga di comando per le applicazioni non vengono trattate in modo diverso dagli argomenti della riga di comando dal linguaggio C. Sono solo argomenti che, in un ambiente Linux o Unix, iniziano tradizionalmente con un trattino (`-`).

Con glibc in ambiente Linux o Unix è possibile utilizzare gli strumenti getopt per definire, validare e analizzare facilmente le opzioni da riga di comando dal resto degli argomenti.

Questi strumenti si aspettano che le opzioni siano formattate secondo gli standard di codifica GNU , che è un'estensione di ciò che specifica POSIX per il formato delle opzioni della riga di comando.

L'esempio seguente mostra la gestione delle opzioni da riga di comando con gli strumenti GNU getopt.

```
#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;
}
```

Può essere compilato con `gcc` :

```
gcc example.c -o example
```

Supporta tre opzioni della riga di comando (`--help` , `--file` e `--msg`). Tutti hanno anche una "forma breve" (`-h` , `-f` , e `-m`). Le opzioni "file" e "msg" accettano entrambi gli argomenti. Se si specifica l'opzione "msg", è richiesto l'argomento.

Gli argomenti per le opzioni sono formattati come:

- `--option=value` (per le opzioni di forma lunga)
- `-ovalue` o `-o"value"` (per opzioni in forma breve)

Leggi Argomenti della riga di comando online: https://riptutorial.com/it/c/topic/1285/argomenti-della-riga-di-comando

Capitolo 6: Argomenti variabili

introduzione

Gli argomenti variabili sono usati dalle funzioni della famiglia printf (`printf` , `fprintf` , ecc.) E altri per consentire a una funzione di essere chiamata ogni volta con un numero diverso di argomenti, da cui il nome *varargs* .

Per implementare le funzioni utilizzando la funzione di argomenti variabili, utilizzare `#include <stdarg.h>` .

Per chiamare le funzioni che accettano un numero variabile di argomenti, assicurarsi che esista un prototipo completo con i puntini di sospensione finali: `void err_exit(const char *format, ...);` per esempio.

Sintassi

- void va_start (va_list ap, *last*); / * Inizia l'elaborazione degli argomenti variadici; *l'ultimo* è l'ultimo parametro di funzione prima dei puntini di sospensione ("...") * /
- *digita* va_arg (va_list ap, *type*); / * Trova il prossimo argomento variadico nella lista; assicurati di passare il tipo corretto *promosso* * /
- void va_end (va_list ap); / * Elaborazione argomento finale * /
- void va_copy (va_list dst, va_list src); / * C99 o successivo: elenco degli argomenti della copia, ovvero la posizione corrente nell'elaborazione degli argomenti, in un'altra lista (ad esempio per passare più volte gli argomenti) * /

Parametri

Parametro	Dettagli
`va_list` ap	puntatore argomento, posizione corrente nell'elenco di argomenti variadici
scorso	nome dell'ultimo argomento di funzione non variabile, quindi il compilatore trova il punto corretto per iniziare l'elaborazione degli argomenti variadici; potrebbe non essere dichiarato come variabile di `register` , funzione o tipo di matrice
genere	tipo **promosso** dell'argomento variadic da leggere (es. `int` per un `short int` argomento `short int`)
`va_list` src	puntatore argomento corrente da copiare
`va_list` dst	nuovo elenco di argomenti da compilare

Osservazioni

Le `va_start` , `va_arg` , `va_end` e `va_copy` sono in realtà macro.

Assicurati di chiamare *sempre* `va_start` prima, e solo una volta, e di chiamare `va_end` ultimo, e solo una volta, e su ogni punto di uscita della funzione. Non farlo *potrebbe* funzionare sul *tuo* sistema ma sicuramente **non** è portabile e quindi invita bug.

Fare attenzione per dichiarare la funzione in modo corretto, vale a dire con un prototipo, e la mente le restrizioni *l'ultimo* argomento non variadic (non `register` , non è un tipo di funzione o array). Non è possibile dichiarare una funzione che accetta solo argomenti variadici, poiché è necessario almeno un argomento non-variadico per avviare l'elaborazione degli argomenti.

Quando si chiama `va_arg` , è necessario richiedere il tipo di argomento **promosso** , ovvero:

- `short` è promosso a `int` (e `unsigned short` è anche promosso a `int` meno che `sizeof(unsigned short) == sizeof(int)` , nel qual caso viene promosso a `unsigned int`).
- `float` è promosso a `double` .
- `signed char` è promosso a `int` ; `unsigned char` viene promosso a `int` meno che `sizeof(unsigned char) == sizeof(int)` , che raramente è il caso.
- `char` è solitamente promosso a `int` .
- I tipi C99 come `uint8_t` o `int16_t` sono promossi in modo simile.

L'elaborazione di argomenti variadici storici (cioè K & R) è dichiarata in `<varargs.h>` ma non dovrebbe essere utilizzata in quanto obsoleta. L'elaborazione standard degli argomenti variadici (quella descritta qui e dichiarata in `<stdarg.h>`) è stata introdotta in C89; la macro `va_copy` stata introdotta in C99 ma fornita da molti compilatori precedenti.

Examples

Utilizzo di un argomento di conteggio esplicito per determinare la lunghezza della va_list

Con qualsiasi funzione variadica, la funzione deve sapere come interpretare la lista degli argomenti variabili. Con le funzioni `printf()` o `scanf()` , la stringa di formato indica alla funzione cosa aspettarsi.

La tecnica più semplice consiste nel passare un conteggio esplicito degli altri argomenti (che sono normalmente tutti dello stesso tipo). Ciò è dimostrato nella funzione variadica nel codice sottostante che calcola la somma di una serie di numeri interi, dove può esserci un numero qualsiasi di numeri interi ma quel conteggio è specificato come argomento prima dell'elenco di argomenti variabili.

```
#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;
}
```

Utilizzo dei valori del terminatore per determinare la fine di va_list

Con qualsiasi funzione variadica, la funzione deve sapere come interpretare la lista degli argomenti variabili. L'approccio "tradizionale" (esemplificato da `printf`) è quello di specificare il numero di argomenti in anticipo. Tuttavia, questa non è sempre una buona idea:

```
/* 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 */
```

A volte è più robusto aggiungere un terminatore esplicito, esemplificato dalla funzione `execlp()` POSIX. Ecco un'altra funzione per calcolare la somma di una serie di numeri `double` :

```
#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));
}
```

Buoni valori terminatori:

- intero (che dovrebbe essere tutto positivo o non negativo) - `0` o `-1`
- tipi a virgola mobile - `NAN`
- tipi di puntatore - `NULL`
- tipi di enumeratore - qualche valore speciale

Implementazione di funzioni con un'interfaccia `` printf () `

Un uso comune di elenchi di argomenti a lunghezza variabile consiste nell'implementare funzioni che sono un sottile involucro attorno alla famiglia di funzioni `printf()` . Uno di questi esempi è un insieme di funzioni di segnalazione degli errori.

`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
```

Questo è un esempio di nudo; tali pacchetti possono essere molto elaborati. Normalmente, i programmatori useranno `errmsg()` o `warnmsg()` , che usano `verrmsg()` . Se qualcuno esce con la necessità di fare di più, però, la funzione esposta `verrmsg()` sarà utile. Si potrebbe evitare di esporre fino a quando si ha la necessità di esso (YAGNI - non sono gonna bisogno), ma la necessità sarà sorgere alla fine (si *sono* gonna bisogno - YAGNI).

`errmsg.c`

Questo codice ha solo bisogno di inoltrare gli argomenti variadici alla funzione `vfprintf()` per l'output all'errore standard. Segnala anche il messaggio di errore di sistema corrispondente al numero di errore di sistema (`errno`) passato alle funzioni.

```
#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);
}
```

Utilizzando `errmsg.h`

Ora puoi usare quelle funzioni come segue:

```
#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;
}
```

Se le chiamate di sistema `open()` o `read()` falliscono, l'errore viene scritto sull'errore standard e il programma viene `close()` con il codice di uscita 1. Se la chiamata di sistema `close()` fallisce, l'errore viene semplicemente stampato come messaggio di avviso e il programma continua.

Verifica dell'uso corretto dei formati `printf()`

Se si utilizza GCC (il compilatore GNU C, che fa parte della raccolta del compilatore GNU) o si utilizza Clang, è possibile fare in modo che il compilatore verifichi che gli argomenti passati alle funzioni del messaggio di errore corrispondano a quanto `printf()` . Poiché non tutti i compilatori supportano l'estensione, devono essere compilati in modo condizionale, il che è un po 'complicato. Tuttavia, la protezione che dà vale la pena.

Per prima cosa, dobbiamo sapere come rilevare che il compilatore è GCC o Clang che emula GCC. La risposta è che GCC definisce `__GNUC__` per indicare ciò.

Vedi gli attributi delle funzioni comuni per informazioni sugli attributi, in particolare l'attributo `format` .

Errmsg.h `errmsg.h`

```
#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
```

Ora, se commetti un errore come:

```
errmsg(EXIT_FAILURE, errno, "Failed to open file '%d' for reading", filename);
```

(dove `%d` dovrebbe essere `%s`), quindi il compilatore si lamenterà:

```
$ 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
$
```

Utilizzando una stringa di formato

L'uso di una stringa di formato fornisce informazioni sul numero e sul tipo attesi degli argomenti variadici successivi in modo tale da evitare la necessità di un argomento di conteggio esplicito o di un valore di terminatore.

L'esempio seguente mostra una funzione che avvolge la `printf()` standard `printf()` , consentendo solo l'uso di argomenti variadici del tipo `char` , `int` e `double` (nel formato decimale in virgola mobile). Qui, come con `printf()` , il primo argomento della funzione wrapping è la stringa di formato. Quando viene analizzata la stringa di formato, la funzione è in grado di determinare se esiste un altro argomento variadico previsto e quale dovrebbe essere il tipo.

```
#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);
}
```

Leggi Argomenti variabili online: https://riptutorial.com/it/c/topic/455/argomenti-variabili

Capitolo 7: Array

introduzione

Le matrici sono tipi di dati derivati, che rappresentano una raccolta ordinata di valori ("elementi") di un altro tipo. La maggior parte degli array in C ha un numero fisso di elementi di qualsiasi tipo, e la sua rappresentazione memorizza gli elementi in modo contiguo nella memoria senza spazi vuoti o padding. C consente array multidimensionali i cui elementi sono altri array e anche matrici di puntatori.

C supporta matrici allocate dinamicamente le cui dimensioni sono determinate in fase di esecuzione. C99 e versioni successive supporta matrici di lunghezza variabile o VLA.

Sintassi

- nome del tipo [lunghezza]; / * Definisce la matrice di 'tipo' con nome 'nome' e lunghezza 'lunghezza'. * /
- int arr [10] = {0}; / * Definire una matrice e inizializzare TUTTI gli elementi a 0. * /
- int arr [10] = {42}; / * Definire una matrice e inizializzare i primi elementi a 42 e il resto a 0. * /
- int arr [] = {4, 2, 3, 1}; / * Definire e inizializzare un array di lunghezza 4. * /
- arr [n] = valore; / * Imposta il valore all'indice n. * /
- value = arr [n]; / * Ottieni valore all'indice n. * /

Osservazioni

Perché abbiamo bisogno di array?

Gli array forniscono un modo per organizzare gli oggetti in un aggregato con il proprio significato. Ad esempio, le stringhe C sono matrici di caratteri (`char` s) e una stringa come "Hello, World!" ha un significato come un aggregato che non è inerente ai personaggi individualmente. Allo stesso modo, gli array sono comunemente usati per rappresentare vettori e matrici matematiche, così come liste di molti tipi. Inoltre, senza un modo per raggruppare gli elementi, bisognerebbe affrontarli individualmente, ad esempio tramite variabili separate. Non solo è ingombrante, non ospita facilmente raccolte di lunghezze diverse.

Gli array sono convertiti implicitamente in puntatori nella maggior parte dei contesti .

Tranne quando appare come l'operando dell'operatore `sizeof` , l'operatore `_Alignof` (C2011), o l'operatore unario `&` (indirizzo-di), o come letterale stringa usato per inizializzare un (altro) array, una matrice viene convertita implicitamente in ("decade") un puntatore al suo primo elemento. Questa conversione implicita è strettamente accoppiata alla definizione dell'operatore di sottoscrizione dell'array (`[]`): l'espressione `arr[idx]` è definita come equivalente a `*(arr + idx)` . Inoltre, poiché l'aritmetica del puntatore è commutativa, `*(arr + idx)` è anche equivalente a `*(idx + arr)` , che a sua volta è equivalente a `idx[arr]` . Tutte queste espressioni sono valide e valutano allo stesso valore, a condizione che `idx` o `arr` sia un puntatore (o un array, che decompone a un

puntatore), l'altro è un numero intero e il numero intero è un indice valido nell'array a cui punta il puntatore.

Come caso speciale, osserva che `&(arr[0])` è equivalente a `&*(arr + 0)` , che semplifica l' `arr` . Tutte queste espressioni sono intercambiabili ovunque l'ultimo decadimento di un puntatore. Questo semplicemente esprime ancora una volta che un array decade in un puntatore al suo primo elemento.

Al contrario, se l'operatore address-of viene applicato a una matrice di tipo `T[N]` (*ie* `&arr`), allora il risultato ha tipo `T (*)[N]` e punta all'intero array. Questo è distinto da un puntatore al primo elemento dell'array, almeno rispetto all'aritmetica del puntatore, che è definita in termini della dimensione del tipo puntato.

I parametri di funzione non sono matrici .

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

Sebbene la prima dichiarazione di `foo` utilizzi la sintassi di tipo array per il parametro `a` , tale sintassi viene utilizzata per dichiarare un parametro di funzione che dichiara tale parametro come *puntatore* al tipo di elemento dell'array. Pertanto, la seconda firma per `foo()` è semanticamente identica alla prima. Ciò corrisponde al decadimento dei valori di matrice nei puntatori in cui vengono visualizzati come argomenti di una *chiamata di* funzione, in modo tale che se una variabile e un parametro di funzione vengono dichiarati con lo stesso tipo di matrice, il valore di tale variabile è adatto per l'uso in una chiamata di funzione come argomento associato al parametro.

Examples

Dichiarazione e inizializzazione di un array

La sintassi generale per dichiarare un array monodimensionale è

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

dove `type` può essere un tipo built-in o tipi definiti dall'utente come le strutture, `arrName` è un identificatore definito dall'utente e la `size` è una costante intera.

La dichiarazione di una matrice (una matrice di 10 variabili int in questo caso) è fatta in questo modo:

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

ora contiene valori indeterminati. Per assicurarti che mantenga valori zero durante la dichiarazione, puoi farlo:

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

Le matrici possono anche avere inizializzatori, questo esempio dichiara un array di 10 `int` 's, dove i primi 3 `int` conterranno i valori `1` , `2` , `3` , tutti gli altri valori saranno zero:

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

Nel suddetto metodo di inizializzazione, il primo valore nell'elenco sarà assegnato al primo membro dell'array, il secondo valore sarà assegnato al secondo membro dell'array e così via. Se la dimensione dell'elenco è inferiore alla dimensione dell'array, come nell'esempio precedente, i membri rimanenti dell'array verranno inizializzati su zeri. Con l'inizializzazione della lista designata (ISO C99), è possibile l'inizializzazione esplicita dei membri dell'array. Per esempio,

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

Nella maggior parte dei casi, il compilatore può dedurre la lunghezza dell'array per te, questo può essere ottenuto lasciando vuote le parentesi quadre:

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

La dichiarazione di una matrice di lunghezza zero non è consentita.

C99 C11

Array a lunghezza variabile (VLA in breve) sono stati aggiunti in C99 e resi facoltativi in C11. Sono uguali agli array normali, con una differenza importante: la lunghezza non deve essere nota al momento della compilazione. Gli VLA hanno una durata di archiviazione automatica. Solo i riferimenti agli VLA possono avere una durata di archiviazione statica.

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

Importante:

I VLA sono potenzialmente pericolosi. Se l'array `vla` nell'esempio sopra richiede più spazio nello stack rispetto a quello disponibile, lo stack andrà in overflow. L'utilizzo di VLA è quindi spesso scoraggiato da guide di stile e da libri ed esercizi.

Cancellazione dei contenuti dell'array (azzeramento)

A volte è necessario impostare un array su zero, dopo che l'inizializzazione è stata eseguita.

```
#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;
}
```

Una scorciatoia comune al ciclo precedente consiste nell'utilizzare `memset()` da `<string.h>` . Il passaggio `array` come mostrato di seguito lo fa decadere da un puntatore al suo 1 ° elemento.

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

o

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

Come in questo esempio, l' `array` *è* un array e non solo un puntatore al primo elemento di un array (vedi la lunghezza dell'array sul perché questo è importante) è possibile una terza opzione per 0-out dell'array:

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

Lunghezza della matrice

Gli array hanno lunghezze fisse che sono note nell'ambito delle loro dichiarazioni. Tuttavia, è possibile ea volte conveniente calcolare le lunghezze degli array. In particolare, questo può rendere il codice più flessibile quando la lunghezza dell'array viene determinata automaticamente da un inizializzatore:

```
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]);
```

Tuttavia, nella maggior parte dei contesti in cui un array appare in un'espressione, viene automaticamente convertito in ("decays to") un puntatore al suo primo elemento. Il caso in cui un array è l'operando dell'operatore `sizeof` è uno di un piccolo numero di eccezioni. Il puntatore risultante non è di per sé un array e non contiene alcuna informazione sulla lunghezza dell'array da cui è stato derivato. Pertanto, se tale lunghezza è necessaria insieme al puntatore, ad esempio quando il puntatore viene passato a una funzione, deve essere convogliato separatamente.

Ad esempio, supponiamo di voler scrivere una funzione per restituire l'ultimo elemento di una matrice di `int` . Continuando da quanto sopra, potremmo chiamarlo così:

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

La funzione potrebbe essere implementata in questo modo:

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

Si noti in particolare che sebbene la dichiarazione `input` dei parametri sia simile a quella di un array, **in realtà dichiara l' `input` come un puntatore** (su `int`). È esattamente equivalente a dichiarare `input` come `input int *input` . Lo stesso sarebbe vero anche se venisse data una dimensione. Ciò è possibile perché gli array non possono mai essere argomenti reali delle funzioni (decadono in base ai puntatori quando compaiono nelle espressioni di chiamata di funzione) e possono essere visualizzati come mnemonici.

È un errore molto comune tentare di determinare la dimensione dell'array da un puntatore, che non può funzionare. NON FARLO:

```
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 */
}
```

In effetti, quel particolare errore è così comune che alcuni compilatori lo riconoscono e lo mettono in guardia. `clang` , ad esempio, emetterà il seguente avviso:

```
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[])
                     ^
```

Impostazione dei valori negli array

L'accesso ai valori dell'array avviene generalmente tramite parentesi quadre:

```
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];
```

Come effetto collaterale degli operandi verso l'operatore `+` che è scambiabile (-> legge commutativa) il seguente è equivalente:

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

così come le prossime affermazioni sono equivalenti:

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

e anche loro due:

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

C non esegue alcuna verifica dei limiti, l'accesso ai contenuti al di fuori dell'array dichiarato non è definito (Accesso alla memoria oltre il blocco assegnato):

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

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

Definire l'array e accedere all'elemento dell'array

```
#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;
}
```

Allocare e azzerare l'inizializzazione di un array con dimensioni definite dall'utente

```
#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;
}
```

Questo programma tenta di eseguire la scansione di un valore intero senza segno dall'input standard, allocare un blocco di memoria per un array di `n` elementi di tipo `int` chiamando la funzione `calloc()` . La memoria è inizializzata a tutti gli zeri da quest'ultimo.

In caso di successo, la memoria viene rilasciata dalla chiamata a `free()` .

Iterazione attraverso una matrice in modo efficiente e ordine di riga maggiore

Le matrici in C possono essere viste come un blocco contiguo di memoria. Più precisamente, l'ultima dimensione dell'array è la parte contigua. Lo chiamiamo l' *ordine di riga-maggiore* . La comprensione di questo e il fatto che un guasto di cache carica una linea di cache completa nella cache quando l'accesso ai dati memorizzati nella cache per evitare successive difetti di cache, possiamo vedere perché l'accesso una matrice di dimensione 10000x10000 con `array[0][0]` **potenzialmente** caricare `array[0][1]` nella cache, ma l'accesso `array[1][0]` subito dopo genererebbe un secondo errore di cache, dato che è `sizeof(type)*10000` byte di distanza `array[0][0]` , e quindi sicuramente non sulla stessa riga della cache. Ecco perché iterare in questo

modo è inefficiente:

```
#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;
    }
}
```

E l'iterazione di questo è più efficiente:

```
#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;
    }
}
```

Allo stesso modo, questo è il motivo per cui quando si ha a che fare con un array con una dimensione e più indici (diciamo 2 dimensioni qui per semplicità con gli indici i e j) è importante scorrere attraverso l'array in questo modo:

```
#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;
    }
}
```

O con 3 dimensioni e indici i, j e 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;
        }
    }
}
```

O in un modo più generico, quando abbiamo un array con **N1 x N2 x ... x Nd** elementi, **d** dimensioni e indici annotati come **n1, n2, ..., e** l'offset è calcolato come questo

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

Immagine / formula presa da: https://en.wikipedia.org/wiki/Row-major_order

Matrici multidimensionali

Il linguaggio di programmazione C consente array multidimensionali . Ecco la forma generale di una dichiarazione di array multidimensionale -

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

Ad esempio, la seguente dichiarazione crea un array intero tridimensionale (5 x 10 x 4):

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

Array bidimensionali

La forma più semplice di array multidimensionale è l'array bidimensionale. Un array bidimensionale è, in sostanza, un elenco di matrici monodimensionali. Per dichiarare un array intero bidimensionale di dimensioni mxn, possiamo scrivere come segue:

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

Dove `type` può essere qualsiasi tipo di dati C valido (`int` , `float` , ecc.) E `arrayName` può essere qualsiasi identificatore C valido. Una matrice bidimensionale può essere visualizzata come una tabella con `m` righe e `n` colonne. **Nota** : l'ordine *ha* importanza in C. L'array `int a[4][3]` non è lo stesso dell'array `int a[3][4]` . Il numero di righe viene prima dato che C è una *riga, la* lingua principale.

Un array bidimensionale `a` , che contiene tre righe e quattro colonne, può essere mostrato come segue:

	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]

Quindi, ogni elemento dell'array `a` è identificato da un nome di elemento del modulo `a[i][j]` , dove `a` è il nome dell'array, `i` rappresenta quale riga e `j` rappresenta quale colonna. Ricorda che le righe e le colonne sono indicizzate a zero. Questo è molto simile alla notazione matematica per le matrici 2-D subscript.

Inizializzazione di matrici bidimensionali

Gli array multidimensionali possono essere inizializzati specificando i valori tra parentesi per ogni riga. Di seguito viene definito un array con 3 righe in cui ogni riga ha 4 colonne.

```
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 */
};
```

Le parentesi graffe nidificate, che indicano la riga desiderata, sono facoltative. La seguente inizializzazione è equivalente all'esempio precedente:

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

Mentre il metodo di creazione di array con parentesi graffe nidificate è facoltativo, è fortemente incoraggiato in quanto è più leggibile e più chiaro.

Accesso a elementi di array bidimensionali

È possibile accedere a un elemento di una matrice bidimensionale utilizzando gli indici, ovvero l'indice di riga e l'indice di colonna dell'array. Ad esempio -

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

L'affermazione precedente prenderà il 4o elemento dalla 3a riga dell'array. Controlliamo il seguente programma in cui abbiamo usato un ciclo annidato per gestire un array bidimensionale:

```
#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++ ) {
```

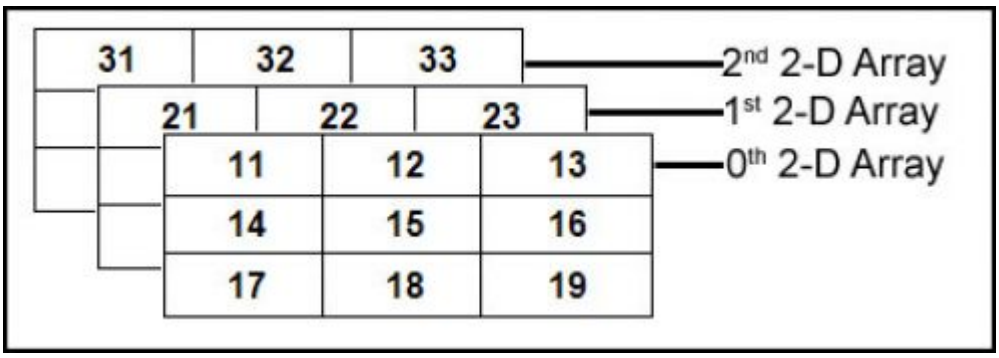
```
      for ( j = 0; j < 2; j++ ) {
         printf("a[%d][%d] = %d\n", i,j, a[i][j] );
      }
   }

   return 0;
}
```

Quando il codice sopra è compilato ed eseguito, produce il seguente risultato:

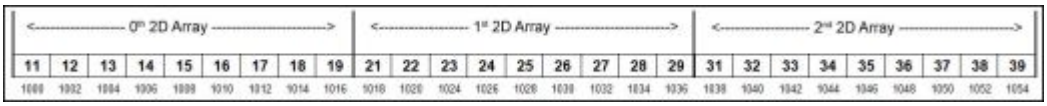
```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

Array tridimensionale:

Una matrice 3D è essenzialmente una matrice di matrici di array: è una matrice o una raccolta di array 2D e una matrice 2D è una matrice di matrici 1D.



Mappa memoria array 3D:



Inizializzazione di una matrice 3D:

```
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}}
};
```

Possiamo avere array con qualsiasi numero di dimensioni, sebbene sia probabile che la maggior parte degli array che verranno creati sarà di una o due dimensioni.

Iterare attraverso un array usando i puntatori

```
#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;
}
```

Qui, nella inizializzazione `p` nel primo `for` condizione del ciclo, l'array `a` *decade* ad un puntatore al suo primo elemento, come sarebbe in quasi tutti i luoghi in cui è utilizzato un tale variabile array.

Quindi, `++p` esegue l'aritmetica del puntatore sul puntatore `p` e cammina uno alla volta attraverso gli elementi dell'array, e fa riferimento a essi delegandoli con `*p` .

Passaggio di array multidimensionali a una funzione

Gli array multidimensionali seguono le stesse regole degli array monodimensionali quando li passano a una funzione. Tuttavia, la combinazione di decay-to-pointer, la precedenza degli operatori e i due diversi modi di dichiarare una matrice multidimensionale (array di matrici contro array di puntatori) possono rendere la dichiarazione di tali funzioni non intuitiva. L'esempio seguente mostra i modi corretti per passare gli array multidimensionali.

```
#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);
}
```

Guarda anche

Passare da array a funzioni

Leggi Array online: https://riptutorial.com/it/c/topic/322/array

Capitolo 8: Assemblaggio in linea

Osservazioni

L'assemblaggio in linea è la pratica di aggiungere istruzioni di assemblaggio nel mezzo del codice sorgente C. Nessuno standard ISO C richiede il supporto del montaggio in linea. Poiché non è richiesto, la sintassi per l'assembly in linea varia dal compilatore al compilatore. Anche se in genere è supportato, esistono pochissimi motivi per utilizzare l'assembly inline e molte ragioni per non farlo.

Professionisti

1. **Prestazioni** Scrivendo le istruzioni di assemblaggio specifiche per un'operazione, è possibile ottenere prestazioni migliori rispetto al codice assembly generato dal compilatore. Si noti che questi miglioramenti delle prestazioni sono rari. Nella maggior parte dei casi è possibile ottenere un miglioramento delle prestazioni semplicemente riorganizzando il codice C in modo che l'ottimizzatore possa svolgere il proprio lavoro.
2. **Interfaccia hardware Per il** codice di avvio del driver o del processore del dispositivo potrebbe essere necessario un codice di assembly per accedere ai registri corretti e per garantire che determinate operazioni si verifichino in un ordine specifico con uno specifico ritardo tra le operazioni.

Contro

1. **La** sintassi di **portabilità del compilatore** per l'assemblaggio in linea non è garantita per essere uguale da un compilatore all'altro. Se si sta scrivendo un codice con un assembly inline che deve essere supportato da compilatori diversi, utilizzare le macro del preprocessore (`#ifdef`) per controllare quale compilatore viene utilizzato. Quindi, scrivi una sezione di assembly inline separata per ciascun compilatore supportato.
2. **Portabilità del processore** Non è possibile scrivere assembly in linea per un processore x86 e aspettarsi che funzioni su un processore ARM. L'assembly in linea è progettato per essere scritto per un processore specifico o una famiglia di processori. Se si dispone di un assembly in linea che si desidera supportare su processori diversi, utilizzare le macro del preprocessore per verificare quale processore viene compilato il codice e per selezionare la sezione del codice assembly appropriata.
3. **Modifiche prestazionali future L'** assemblaggio in linea può essere scritto in attesa di ritardi basati su una determinata velocità di clock del processore. Se il programma è compilato per un processore con un clock più veloce, il codice assembly potrebbe non funzionare come previsto.

Examples

gcc Supporto di base asm

Il supporto dell'assembly di base con gcc ha la seguente sintassi:

```
asm [ volatile ] ( AssemblerInstructions )
```

dove `AssemblerInstructions` è il codice di assemblaggio diretto per il processore specificato. La parola chiave volatile è facoltativa e non ha alcun effetto in quanto gcc non ottimizza il codice all'interno di un'istruzione asm di base. `AssemblerInstructions` può contenere più istruzioni di assemblaggio. Un'istruzione asm di base viene utilizzata se si dispone di una routine asm che deve esistere al di fuori di una funzione C. Il seguente esempio è tratto dal manuale GCC:

```
 /* Note that this code will not compile with -masm=intel */
 #define DebugBreak() asm("int $3")
```

In questo esempio, è possibile utilizzare `DebugBreak()` in altri punti del codice e verrà eseguita l'istruzione di assemblaggio `int $3` . Nota che anche se gcc non modificherà alcun codice in un'istruzione asm di base, l'ottimizzatore potrebbe comunque spostare istruzioni asm consecutive attorno. Se si dispone di più istruzioni di assemblaggio che devono essere eseguite in un ordine specifico, includerle in una dichiarazione asm.

gcc Supporto esteso asm

Il supporto esteso di asm in gcc ha la seguente sintassi:

```
asm [volatile] ( AssemblerTemplate
                  : OutputOperands
                  [ : InputOperands
                  [ : Clobbers ] ])

 asm [volatile] goto ( AssemblerTemplate
                       :
                       : InputOperands
                       : Clobbers
                       : GotoLabels)
```

dove `AssemblerTemplate` è il modello per l'istruzione assembler, `OutputOperands` sono tutte le variabili C che possono essere modificate dal codice assembly, `InputOperands` sono tutte le variabili C utilizzate come parametri di input, `Clobbers` sono un elenco o registri modificati dal codice assembly e `GotoLabels` sono tutte le etichette di istruzioni goto che possono essere utilizzate nel codice assembly.

Il formato esteso viene utilizzato all'interno delle funzioni C ed è l'utilizzo più tipico dell'assembly inline. Di seguito è riportato un esempio del kernel di Linux per lo scambio di byte a 16 bit e numeri a 32 bit per un processore ARM:

```
/* 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
```

Ogni sezione di asm usa la variabile `x` come parametro di input e output. La funzione C restituisce quindi il risultato manipolato.

Con il formato asm esteso, gcc può ottimizzare le istruzioni di assemblaggio in un blocco asm seguendo le stesse regole che usa per l'ottimizzazione del codice C. Se vuoi che la tua sezione asm non venga toccata, usa la parola chiave `volatile` per la sezione asm.

gcc Assieme in linea in macro

Possiamo inserire istruzioni di assemblaggio all'interno di una macro e utilizzare la macro come se fosse chiamata una funzione.

```
#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]);
```

L'uso di istruzioni di assemblaggio inline incorporate nel codice C può migliorare il tempo di esecuzione di un programma. Questo è molto utile in situazioni di tempo critico come algoritmi crittografici come AES. Ad esempio, per una semplice operazione di spostamento che è necessaria nell'algoritmo AES, possiamo sostituire un'istruzione di montaggio `Rotate Right` diretta con l'operatore C shift `>>` .

In un'implementazione di 'AES256', nella funzione 'AddRoundKey ()' abbiamo alcune istruzioni come questa:

```
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
```

Assegnano semplicemente il valore di bit di `w` all'array di `subkey` .

Possiamo cambiare tre turni + assegnare e assegnare un'espressione C con un solo assieme `Rotate Right` .

```
__asm__ ("l.ror  %0,%1,%2" : "=r" (* (unsigned int *) subkey)  : "r" (w), "r" (0x10));
```

Il risultato finale è esattamente lo stesso.

Leggi Assemblaggio in linea online: https://riptutorial.com/it/c/topic/4263/assemblaggio-in-linea

Capitolo 9: Asserzione

introduzione

Un'affermazione è un predicato che la condizione presentata deve essere vera nel momento in cui l'asserzione viene rilevata dal software. Più comuni sono ***le asserzioni semplici*** , che vengono convalidate al momento dell'esecuzione. Tuttavia, ***le asserzioni statiche*** vengono controllate al momento della compilazione.

Sintassi

- affermare (espressione)
- static_assert (espressione, messaggio)
- _Static_assert (espressione, messaggio)

Parametri

Parametro	Dettagli
espressione	espressione di tipo scalare.
Messaggio	stringa letterale da includere nel messaggio di diagnostica.

Osservazioni

Sia `assert` che `static_assert` sono macro definite in `assert.h` .

La definizione di `assert` dipende dalla macro `NDEBUG` che non è definita dalla libreria standard. Se `NDEBUG` è definito, `assert` è un no-op:

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

L'opinione varia a seconda che `NDEBUG` debba essere sempre utilizzato per le compilation di produzione.

- Il campo professionale sostiene che `assert` chiamate `abort` e che i messaggi di asserzione non sono utili per gli utenti finali, quindi il risultato non è utile all'utente. Se si verificano condizioni fatali per il controllo del codice di produzione, è necessario utilizzare le normali condizioni `if/else` e `exit` o `quick_exit` per terminare il programma. Al contrario di `abort` , questi permettono al programma di fare un po 'di pulizia (tramite funzioni registrate con

`atexit` o `at_quick_exit`).

- Il campo di concentramento sostiene che le chiamate di `assert` non dovrebbero mai attivare il codice di produzione, ma se lo fanno, la condizione che viene controllata significa che c'è qualcosa di drammaticamente sbagliato e il programma si comporterebbe in modo peggiore se l'esecuzione dovesse continuare. Pertanto, è meglio avere le asserzioni attive nel codice di produzione perché se sparano, l'inferno si è già scatenato.
- Un'altra opzione consiste nell'utilizzare un sistema di asserzioni home-brew che esegue sempre il controllo ma gestisce gli errori in modo diverso tra lo sviluppo (dove l' `abort` è appropriata) e la produzione (dove un 'errore interno imprevisto - contattare l'assistenza tecnica' potrebbe essere più appropriato).

`static_assert` espande in `_Static_assert` che è una parola chiave. La condizione viene verificata al momento della compilazione, quindi la `condition` deve essere un'espressione costante. Non è necessario che questo sia gestito in modo diverso tra sviluppo e produzione.

Examples

Presupposto e Postcondizione

Un caso d'uso per l'asserzione è precondizione e post-condizionale. Questo può essere molto utile per mantenere invariabile e progettare per contratto . Ad esempio, una lunghezza è sempre zero o positiva, quindi questa funzione deve restituire uno zero o un valore positivo.

```
#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;
}
```

Asserzione semplice

Un'asserzione è un'affermazione usata per affermare che un fatto deve essere vero quando viene raggiunta quella linea di codice. Le asserzioni sono utili per garantire che le condizioni previste siano soddisfatte. Quando la condizione passata a un'affermazione è vera, non c'è azione. Il comportamento su false condizioni dipende dai flag del compilatore. Quando le asserzioni sono abilitate, un falso input provoca l'arresto immediato del programma. Quando sono disabilitati, non viene intrapresa alcuna azione. È pratica comune abilitare le asserzioni nelle build interne e di debug e disabilitarle nelle build di rilascio, sebbene le asserzioni siano spesso abilitate nel rilascio. (Se la terminazione è migliore o peggiore degli errori dipende dal programma.) Le asserzioni dovrebbero essere utilizzate solo per rilevare errori di programmazione interni, che in genere significa essere passati parametri non validi.

```
#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;
}
```

Possibile output con `NDEBUG` indefinito:

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

Possibile output con `NDEBUG` definito:

```
x = -1
```

È buona norma definire `NDEBUG` livello globale, in modo da poter facilmente compilare il codice con tutte le asserzioni `NDEBUG` o disattivate. Un modo semplice per farlo è definire `NDEBUG` come opzione per il compilatore, o definirlo in un'intestazione di configurazione condivisa (es. `config.h`).

Asserzione statica

C11

Le asserzioni statiche vengono utilizzate per verificare se una condizione è vera quando il codice è compilato. In caso contrario, è richiesto al compilatore di inviare un messaggio di errore e interrompere il processo di compilazione.

Un'asserzione statica è quella che viene verificata al momento della compilazione, non il tempo di esecuzione. La condizione deve essere un'espressione costante e se false genererà un errore del compilatore. Il primo argomento, la condizione che viene controllata, deve essere un'espressione costante e il secondo una stringa letterale.

A differenza `_Static_assert` , `_Static_assert` è una parola chiave. Una macro di convenienza `static_assert` è definita 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

Prima di C11, non esisteva alcun supporto diretto per asserzioni statiche. Tuttavia, in C99, le asserzioni statiche potevano essere emulate con macro che avrebbero attivato un errore di compilazione se la condizione di compilazione era falsa. A differenza di `_Static_assert` , il secondo parametro deve essere un nome di token appropriato in modo che possa essere creato un nome di variabile con esso. Se l'asserzione fallisce, il nome della variabile viene visualizzato nell'errore del compilatore, poiché tale variabile è stata utilizzata in una dichiarazione di array sintatticamente errata.

```
#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 */
```

Prima di C99, non si potevano dichiarare variabili in posizioni arbitrarie in un blocco, quindi si dovrebbe essere estremamente cauti nell'usare questa macro, assicurandosi che appaia solo dove una dichiarazione di variabile sarebbe valida.

Asserzione di codice irraggiungibile

Durante lo sviluppo, quando determinati percorsi di codice devono essere impediti dalla portata del flusso di controllo, è possibile utilizzare `assert(0)` per indicare che tale condizione è errata:

```
switch (color) {
    case COLOR_RED:
    case COLOR_GREEN:
    case COLOR_BLUE:
        break;
```

```
    default:
        assert(0);
}
```

Ogni volta che l'argomento della macro `assert()` valutato falso, la macro scriverà le informazioni diagnostiche sul flusso di errore standard e quindi interromperà il programma. Queste informazioni includono il numero di file e di riga `assert()` e possono essere molto utili nel debugging. Gli avvisi possono essere disabilitati definendo la macro `NDEBUG` .

Un altro modo per terminare un programma quando si verifica un errore sono con le funzioni di libreria standard `exit` , `quick_exit` o `abort` . `exit` e `quick_exit` accettano un argomento che può essere passato al tuo ambiente. `abort()` (e quindi `assert`) può essere una terminazione molto grave del programma, e alcune pulizie che altrimenti verrebbero eseguite alla fine dell'esecuzione, potrebbero non essere eseguite.

Il vantaggio principale di `assert()` è che stampa automaticamente le informazioni di debug. Calling `abort()` ha il vantaggio che non può essere disattivato come un assert, ma non può causare la visualizzazione di informazioni di debug. In alcune situazioni, l'utilizzo di entrambi i costrutti può essere utile:

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

Quando gli `assert()` sono *abilitati* , la chiamata `assert()` stamperà le informazioni di debug e terminerà il programma. L'esecuzione non raggiunge mai la chiamata `abort()` . Quando gli `assert()` sono *disabilitati* , la chiamata `assert()` non fa nulla e viene `abort()` . Ciò garantisce che il programma termini *sempre* per questa condizione di errore; l'abilitazione e la disabilitazione asserisce solo gli effetti indipendentemente dalla stampa dell'output di debug.

Non si dovrebbe mai lasciare una tale `assert` nel codice di produzione, poiché le informazioni di debug non sono utili per gli utenti finali e perché l' `abort` è in genere una terminazione troppo grave che impedisce l'esecuzione di gestori di pulizia installati per l' `exit` o di `quick_exit` .

Segnala messaggi di errore

Esiste un trucco in grado di visualizzare un messaggio di errore insieme a un'asserzione. Normalmente, si dovrebbe scrivere un codice come questo

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

Se l'asserzione falliva, un messaggio di errore sarebbe simile

> Asserzione non riuscita: p! = NULL, file main.c, riga 5

Tuttavia, è possibile utilizzare AND logico (`&&`) per fornire anche un messaggio di errore

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

Ora, se l'asserzione fallisce, un messaggio di errore leggerà qualcosa del genere

> Asserzione non riuscita: p! = NULL && "funzione f: p non può essere NULL", file main.c, riga 5

La ragione per cui questo funziona è che una stringa letterale valuta sempre a non zero (true). Aggiungere `&& 1` a un'espressione booleana non ha alcun effetto. Pertanto, l'aggiunta di `&& "error message"` non ha alcun effetto, ad eccezione del fatto che il compilatore visualizzerà l'intera espressione che ha avuto esito negativo.

Leggi Asserzione online: https://riptutorial.com/it/c/topic/555/asserzione

Capitolo 10: Atomics

Sintassi

- `#ifdef __STDC_NO_ATOMICS__`
- `# error this implementation needs atomics`
- `#endif`
- `#include <stdatomic.h>`
- _ contatore asimmetrico = ATOMIC_VAR_INIT (0);

Osservazioni

Atomics come parte del linguaggio C è una funzionalità opzionale disponibile dal C11.

Il loro scopo è quello di garantire l'accesso senza vincoli alle variabili condivise tra diversi thread. Senza qualifica atomica, lo stato di una variabile condivisa non sarebbe definito se due thread accedono contemporaneamente. Ad esempio, un'operazione di incremento (`++`) potrebbe essere suddivisa in diverse istruzioni di assemblatore, una lettura, l'aggiunta stessa e un'istruzione di memorizzazione. Se un altro thread eseguisse la stessa operazione, le loro due sequenze di istruzioni potrebbero essere intrecciate e portare a un risultato incoerente.

- **Tipi:** tutti i tipi di oggetto, ad eccezione dei tipi di array, possono essere qualificati con `_Atomic` .
- **Operatori:** tutti gli **operatori di** lettura-modifica-scrittura (ad esempio `++` o `*=`) su questi sono garantiti per essere atomici.
- **Operazioni:** ci sono alcune altre operazioni che sono specificate come funzioni generiche di tipo, ad esempio `atomic_compare_exchange` .
- **Discussioni: l'** accesso ad esse è garantito per non produrre corse di dati quando vi si accede da thread diversi.
- **Gestori del segnale: i** tipi atomici sono chiamati senza *blocco* se tutte le operazioni su di essi sono *prive di* stato. In tal caso possono anche essere usati per gestire cambiamenti di stato tra il normale flusso di controllo e un gestore di segnale.
- Esiste un solo tipo di dati che è sicuro di essere privo di blocco: `atomic_flag` . Questo è un tipo minimale le cui operazioni devono essere mappate su efficienti istruzioni hardware test-and-set.

Altri strumenti per evitare le condizioni di gara sono disponibili nell'interfaccia di thread di C11, in particolare un mutex di tipo `mtx_t` per escludere reciprocamente i thread dall'accesso a dati critici o sezioni critiche di codice. Se gli atomici non sono disponibili, questi devono essere usati per prevenire le gare.

Examples

atomica e operatori

Le variabili atomiche possono essere consultate contemporaneamente tra diversi thread senza creare condizioni di competizione.

```
/* 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;
}
```

Tutte le operazioni di lvalue (operazioni che modificano l'oggetto) consentite per il tipo di base sono consentite e non porteranno a condizioni di competizione tra diversi thread che le accedono.

- Le operazioni sugli oggetti atomici sono generalmente di ordine di grandezza più lente delle normali operazioni aritmetiche. Questo include anche semplici operazioni di carico o negozio. Quindi dovresti usarli solo per compiti critici.
- Operazioni aritmetiche e assegnazioni usuali come `a = a+1;` sono infatti tre operazioni su `a` : primo carico, poi aggiunta e infine un negozio. Questa *non* è *una* gara libera. Solo l'operazione `a += 1;` e `a++;` siamo.

Leggi Atomics online: https://riptutorial.com/it/c/topic/4924/atomics

Capitolo 11: booleano

Osservazioni

Per utilizzare il tipo predefinito `_Bool` e l'intestazione `<stdbool.h>` , è necessario utilizzare le versioni C99 / C11 di C.

Per evitare avvisi del compilatore ed eventualmente errori, dovresti usare l'esempio `typedef` / `define` se stai usando C89 e le versioni precedenti della lingua.

Examples

Utilizzando stdbool.h

C99

Utilizzando il file di intestazione di sistema `stdbool.h` consente di utilizzare `bool` come tipo di dati booleano. `true` restituisce `1` e `false` valuta a `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` è solo una bella ortografia per il tipo di dati `_Bool` . Ha regole speciali quando vengono convertiti numeri o puntatori.

Utilizzando #define

C di tutte le versioni, tratterà efficacemente qualsiasi valore intero diverso da `0` come `true` per gli operatori di confronto e il valore intero `0` come `false` . Se non si dispone di `_Bool` o `bool` partire da C99, è possibile simulare un tipo di dati booleano in C utilizzando le macro `#define` e si potrebbero ancora trovare tali elementi nel codice legacy.

```
#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!");
    }
}
```

Non introdurre questo nel nuovo codice poiché la definizione di queste macro potrebbe essere in conflitto con gli usi moderni di `<stdbool.h>` .

Uso del tipo _Bool intrinseco (incorporato)

C99

Aggiunto nella versione C standard C99, `_Bool` è anche un tipo di dati C nativo. È in grado di contenere i valori `0` (per *falso*) e `1` (per *vero*).

```
#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` è un tipo intero ma ha regole speciali per le conversioni di altri tipi. Il risultato è analogo all'uso di altri tipi in `if` espressioni . Nel seguente

```
_Bool z = X;
```

- Se `X` ha un tipo aritmetico (è un qualsiasi tipo di numero), `z` diventa `0` se `X == 0` . Altrimenti, `z` diventa `1` .
- Se `X` ha un tipo di puntatore, `z` diventa `0` se `X` è un puntatore nullo e `1` altrimenti.

Per usare le `<stdbool.h>` più belle `bool` , `false` e `true` devi usare `<stdbool.h>` .

Numeri interi e puntatori nelle espressioni booleane.

Tutti gli interi oi puntatori possono essere utilizzati in un'espressione interpretata come "valore di verità".

```
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");
  }
...
```

L'espressione `argc % 4` viene valutata e porta a uno dei valori `0` , `1` , `2` o `3` . Il primo, `0` è l'unico valore che è "falso" e porta l'esecuzione nella parte `else` . Tutti gli altri valori sono "veri" e vanno nella parte `if` .

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

Qui viene valutato il puntatore `A` e, se si tratta di un puntatore nullo, viene rilevato un errore e il programma viene chiuso.

Molte persone preferiscono scrivere qualcosa come `A == NULL` , invece, ma se si dispone di confronti di puntatore come parte di altre espressioni complicate, le cose diventano rapidamente difficili da leggere.

```
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]);
}
```

Per fare ciò, dovresti eseguire la scansione di un codice complicato nell'espressione e accertarti delle preferenze dell'operatore.

```
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]);
}
```

è relativamente facile da catturare: se il puntatore è valido controlliamo se il primo carattere è diverso da zero e quindi controlla se è una lettera.

Definire un tipo di bool usando typedef

Considerando che la maggior parte dei debugger non conoscono le macro `#define` , ma possono controllare le costanti `enum` , potrebbe essere opportuno fare qualcosa del genere:

```
#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;
```

Ciò consente ai compilatori per le versioni storiche di C di funzionare, ma rimane compatibile in avanti se il codice è compilato con un compilatore C moderno.

Per maggiori informazioni su `typedef` , vedi Typedef , per ulteriori informazioni su `enum` consulta Enumerazioni

Leggi booleano online: https://riptutorial.com/it/c/topic/3336/booleano

Capitolo 12: Classi di archiviazione

introduzione

Una classe di archiviazione viene utilizzata per impostare l'ambito di una variabile o funzione. Conoscendo la classe di archiviazione di una variabile, possiamo determinare il tempo di vita di tale variabile durante il tempo di esecuzione del programma.

Sintassi

- [auto | register | static | extern] <Tipo di dati> <Nome variabile> [= <Valore>];
- [static _Thread_local | extern _Thread_local | _Thread_local] <Tipo di dati> <Nome variabile> [= <Valore>]; / * poiché> = C11 * /
- Esempi:
- typedef int *foo* ;
- extern int *foo* [2];

Osservazioni

Gli identificatori di classe di archiviazione sono le parole chiave che possono essere visualizzate accanto al tipo di livello superiore di una dichiarazione. L'utilizzo di queste parole chiave influisce sulla durata e sul collegamento all'archiviazione dell'oggetto dichiarato, a seconda che sia dichiarato nell'ambito di un file o nell'ambito di un blocco:

Parola chiave	Durata di archiviazione	collegamento	Osservazioni
`static`	Statico	Interno	Imposta il collegamento interno per gli oggetti nell'ambito del file; imposta la durata di archiviazione statica per gli oggetti nell'ambito del blocco.
`extern`	Statico	Esterno	Implicito e quindi ridondante per oggetti definiti nell'ambito del file che hanno anche un inizializzatore. Se utilizzato in una dichiarazione sullo scope del file senza un inizializzatore, suggerisce che la definizione deve essere trovata in un'altra unità di traduzione e verrà risolta al momento del collegamento.
`auto`	Automatico	non pertinente	Implicito e quindi ridondante per oggetti

Parola chiave	Durata di archiviazione	collegamento	Osservazioni
			dichiarati a scopo di blocco.
`register`	Automatico	non pertinente	Rilevante solo per gli oggetti con durata di archiviazione automatica. Fornisce un suggerimento per cui la variabile deve essere memorizzata in un registro. Un vincolo imposto è che non è possibile utilizzare l'operatore unario `&` "indirizzo di" su tale oggetto e pertanto l'oggetto non può essere sottoposto a aliasing.
`typedef`	non pertinente	non pertinente	Non è un identificatore di classe di archiviazione in pratica, ma funziona come uno da un punto di vista sintattico. L'unica differenza è che l'identificatore dichiarato è un tipo, piuttosto che un oggetto.
`_Thread_local`	Filo	Interno esterno	Introdotto in C11, per rappresentare la *durata della memorizzazione dei thread* . Se utilizzato a livello di blocco, deve includere anche `extern` o `static` .

Ogni oggetto ha una durata di archiviazione associata (indipendentemente dall'ambito) e un collegamento (rilevante solo per le dichiarazioni nell'ambito del file), anche quando queste parole chiave vengono omesse.

L'ordine degli specificatori della classe di memoria rispetto agli identificatori di tipo di primo livello (`int` , `unsigned` , `short` , ecc.) E qualificatori di tipo di primo livello (`const` , `volatile`) non viene applicato, quindi entrambe le dichiarazioni sono valide:

```
int static const unsigned a = 5; /* bad practice */
static const unsigned int b = 5; /* good practice */
```

Tuttavia, è considerata una buona pratica inserire prima gli identificatori della classe di memoria, quindi qualsiasi qualificatore di tipo, quindi lo specificatore di tipo (`void` , `char` , `int` , `signed long` , `unsigned long long` , `long double` ...).

Non tutti gli specificatori di classe di archiviazione sono legali in un determinato ambito:

```
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;
```

Durata di archiviazione

La durata dell'archiviazione può essere statica o automatica. Per un oggetto dichiarato, viene determinato in base all'ambito e agli identificatori della classe di archiviazione.

Durata dell'archiviazione statica

Le variabili con durata dell'archiviazione statica vivono durante l'intera esecuzione del programma e possono essere dichiarate sia nell'ambito di file (con o senza `static`) sia nell'ambito di un blocco (mettendo `static` esplicito). Solitamente vengono allocati e inizializzati dal sistema operativo all'avvio del programma e recuperati al termine del processo. In pratica, i formati eseguibili hanno sezioni dedicate per tali variabili (`data` , `bss` e `rodata`) e queste intere sezioni del file vengono mappate in memoria in determinati intervalli.

Durata della memorizzazione del thread

C11

Questa durata di archiviazione è stata introdotta in C11. Questo non era disponibile negli standard C precedenti. Alcuni compilatori forniscono un'estensione non standard con semantica simile. Ad esempio, gcc supporta l' `__thread` che può essere utilizzato negli standard C precedenti che non avevano `_Thread_local` .

Le variabili con durata di memorizzazione del thread possono essere dichiarate sia nell'ambito del file che nell'ambito del blocco. Se dichiarato a livello di blocco, deve anche utilizzare `static` `extern` memorizzazione `static` o `extern` . La sua vita è l'intera esecuzione del *thread* in cui è stato creato. Questo è l'unico identificatore di archiviazione che può apparire accanto a un altro identificatore di memorizzazione.

Durata archiviazione automatica

Le variabili con durata di memorizzazione automatica possono essere dichiarate solo a livello di blocco (direttamente all'interno di una funzione o all'interno di un blocco in quella funzione). Sono utilizzabili solo nel periodo tra l'entrata e l'uscita dalla funzione o dal blocco. Una volta che la variabile esce dall'ambito (o ritornando dalla funzione o abbandonando il blocco), la sua memoria viene automaticamente deallocata. Qualsiasi ulteriore riferimento alla stessa variabile dai puntatori non è valido e porta a un comportamento non definito.

Nelle implementazioni tipiche, le variabili automatiche si trovano a determinati offset nel frame dello stack di una funzione o nei registri.

Collegamento esterno e interno

Il collegamento è rilevante solo per gli oggetti (funzioni e variabili) dichiarati nell'ambito del file e influisce sulla loro visibilità su diverse unità di traduzione. Gli oggetti con collegamento esterno sono visibili in ogni altra unità di traduzione (a condizione che sia inclusa la dichiarazione appropriata). Gli oggetti con collegamento interno non sono esposti ad altre unità di traduzione e possono essere utilizzati solo nell'unità di traduzione dove sono definiti.

Examples

typedef

Definisce un nuovo tipo basato su un tipo esistente. La sua sintassi rispecchia quella di una dichiarazione di variabile.

```
/* 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);
```

Pur non essendo tecnicamente una classe di archiviazione, un compilatore lo tratterà come uno poiché nessuna delle altre classi di memoria è consentita se viene utilizzata la parola chiave `typedef` .

I `typedef` sono importanti e non dovrebbero essere sostituiti con la macro `#define` .

```
typedef int newType;
newType *ptr;        // ptr is pointer to variable of type 'newType' aka int
```

Però,

```
#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

Questa classe di memoria indica che un identificatore ha una durata di archiviazione automatica. Ciò significa che una volta terminato lo scope in cui è stato definito l'identificatore, l'oggetto denotato dall'identificatore non è più valido.

Poiché tutti gli oggetti, che non vivono in ambito globale o sono dichiarati `static` , hanno una durata di archiviazione automatica per impostazione predefinita quando definita, questa parola chiave è per lo più di interesse storico e non deve essere utilizzata:

```
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. */
```

statico

La classe di archiviazione `static` scopi diversi, a seconda della posizione della dichiarazione nel file:

1. Per limitare l'identificatore solo a quell'unità di traduzione (scope = file).

```
/* 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. Per salvare i dati da utilizzare con la prossima chiamata di una funzione (scope = 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;
}
```

Questo codice stampa:

```
 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
```

Le variabili statiche mantengono il loro valore anche se chiamate da più thread differenti.

C99

3. Si prevede che nei parametri di funzione per indicare che un array abbia un numero minimo costante di elementi e un parametro non nullo:

```
/* 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]);
}
```

Il numero richiesto di elementi (o anche un puntatore non nullo) non è necessariamente controllato dal compilatore, e i compilatori non sono tenuti a notificare in alcun modo se non si dispone di elementi sufficienti. Se un programmatore supera meno di 512 elementi o un puntatore nullo, il risultato è un comportamento non definito. Poiché è impossibile applicarlo, è necessario prestare maggiore attenzione quando si passa un valore per tale parametro a tale funzione.

extern

Usato per **dichiarare un oggetto o una funzione** che è definita altrove (e che ha *un collegamento esterno*). In generale, viene utilizzato per dichiarare un oggetto o una funzione da utilizzare in un modulo che non è quello in cui è definito l'oggetto o la funzione corrispondente:

```
/* 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

Le cose si fanno leggermente più interessanti con l'introduzione della parola chiave `inline` 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);
```

Registrare

Suggerimenti per il compilatore che l'accesso a un oggetto dovrebbe essere il più veloce possibile. Se il compilatore utilizza effettivamente il suggerimento è definito dall'implementazione; può semplicemente trattarlo come equivalente ad `auto` .

L'unica proprietà che è definitivamente diversa per tutti gli oggetti dichiarati con il `register` è che non è possibile calcolare il loro indirizzo. In tal modo il `register` può essere un buon strumento per garantire determinate ottimizzazioni:

```
register size_t size = 467;
```

è un oggetto che non può mai *alias* perché nessun codice può passare il suo indirizzo a un'altra funzione in cui potrebbe essere cambiato in modo imprevisto.

Questa proprietà implica anche una matrice

```
register int array[5];
```

non può decadere in un puntatore al suo primo elemento (cioè l' `array` diventa `&array[0]`). Ciò significa che non è possibile accedere agli elementi di tale array e che lo stesso array non può essere passato a una funzione.

In effetti, l'unico utilizzo legale di un array dichiarato con una classe di archiviazione del `register` è l'operatore `sizeof` ; qualsiasi altro operatore richiederebbe l'indirizzo del primo elemento dell'array. Per questo motivo, gli array in genere non dovrebbero essere dichiarati con la parola chiave `register` poiché li rende inutili per qualcosa di diverso dal calcolo della dimensione dell'intero array, il che può essere fatto altrettanto facilmente senza la parola chiave `register` .

La classe di archiviazione del `register` è più appropriata per le variabili definite all'interno di un blocco e accessibili con alta frequenza. Per esempio,

```
/* 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

L'operatore `_Alignof` può anche essere utilizzato con `register` array di `register` .

_Thread_local

C11

Questo era un nuovo identificatore di memoria introdotto in C11 insieme al multi-threading. Questo non è disponibile negli standard C precedenti.

Indica la *durata della memorizzazione del thread* . Una variabile dichiarata con lo specificatore di memoria `_Thread_local` denota che l'oggetto è *locale a quel thread* e la sua durata è l'intera esecuzione del thread in cui è stato creato. Può anche apparire insieme a `static` o `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);
    }
}
```

Leggi Classi di archiviazione online: https://riptutorial.com/it/c/topic/3597/classi-di-archiviazione

Capitolo 13: Commenti

introduzione

I commenti sono usati per indicare qualcosa alla persona che legge il codice. I commenti sono trattati come un vuoto dal compilatore e non cambiano nulla nel significato attuale del codice. Esistono due sintassi utilizzate per i commenti in C, l'originale `/* */` e il leggermente più nuovo `//` . Alcuni sistemi di documentazione usano commenti appositamente formattati per aiutare a produrre la documentazione per il codice.

Sintassi

- `/*...*/`
- `//...` (solo C99 e successivi)

Examples

/ * * / commenti delimitati

Un commento inizia con una barra diretta seguita immediatamente da un asterisco (`/*`) e termina non appena si incontra un asterisco immediatamente seguito da una barra (`*/`). Tutto in mezzo a queste combinazioni di caratteri è un commento ed è trattato come un vuoto (sostanzialmente ignorato) dal compilatore.

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

Il commento sopra è un commento a riga singola. I commenti di questo `/*` tipo possono estendersi su più righe, in questo modo:

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

Sebbene non sia strettamente necessario, una convenzione di stile comune con commenti su più righe consiste nel mettere gli spazi iniziali e gli asterischi sulle linee successive alla prima, e `/*` e `*/` sulle nuove linee, in modo che vengano allineate tutte:

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

Gli asterischi extra non hanno alcun effetto funzionale sul commento in quanto nessuno di essi ha una barra diretta correlata.

Questi `/*` tipo di commenti possono essere utilizzati sulla propria riga, alla fine di una riga di codice o anche all'interno di righe di codice:

```
/* 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 */
    }
}
```

I commenti non possono essere nidificati. Questo perché qualsiasi `/*` successivo verrà ignorato (come parte del commento) e il primo `*/` raggiunto sarà trattato come terminando il commento. Il commento nel seguente esempio *non funzionerà* :

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

Per commentare blocchi di codice che contengono commenti di questo tipo, che verrebbero altrimenti annidati, vedi Commentare usando l' esempio del preprocessore sotto

// commenti delimitati

C99

C99 ha introdotto l'uso di commenti a riga singola in stile C ++. Questo tipo di commento inizia con due barre e scorre fino alla fine di una riga:

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

Questo tipo di commento non consente commenti su più righe, sebbene sia possibile creare un blocco di commenti aggiungendo più commenti a riga singola uno dopo l'altro:

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

Questo tipo di commento può essere usato su una propria riga o alla fine di una riga di codice. Tuttavia, perché corrono *alla fine della linea,* essi *non* possono essere utilizzati in una linea di codice

```
// 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
}
```

Commentando usando il preprocessore

Grandi blocchi di codice possono anche essere "commentati" usando le direttive del

preprocessore `#if 0` e `#endif` . Ciò è utile quando il codice contiene commenti su più righe che altrimenti non sarebbero nidificati.

```
#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) */
...
```

Possibile insidia dovuta ai trigrafi

C99

Durante la scrittura di `//` commenti delimitati, è possibile fare un errore tipografico che influenza le loro operazioni previste. Se uno digita:

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

Il `/` alla fine è stato un refuso ma ora verrà interpretato in `\` . Questo perché il `??/` forma un trigramma .

Il `??/` trigraph è in realtà una notazione a mano lunga per `\` , che è il simbolo di continuazione della linea. Ciò significa che il compilatore pensa che la riga successiva sia una continuazione della linea corrente, ovvero una continuazione del commento, che potrebbe non essere ciò che è inteso.

```
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;
```

Leggi Commenti online: https://riptutorial.com/it/c/topic/10670/commenti

Capitolo 14: Compilazione

introduzione

Il linguaggio C è tradizionalmente un linguaggio compilato (al contrario di interpretato). Lo standard C definisce le ***fasi di traduzione*** e il prodotto di applicarle è un'immagine di programma (o un programma compilato). In c11 , le fasi sono elencate in §5.1.1.2.

Osservazioni

Estensione del nome file	Descrizione
`.c`	File sorgente. Di solito contiene definizioni e codice.
`.h`	File di intestazione. Di solito contiene dichiarazioni.
`.o`	File oggetto Codice compilato nel linguaggio macchina.
`.obj`	Estensione alternativa per i file oggetto.
`.a`	File di libreria Pacchetto di file oggetto.
`.dll`	Libreria di collegamento dinamico su Windows.
`.so`	Oggetto condiviso (libreria) su molti sistemi simili a Unix.
`.dylib`	Libreria di collegamento dinamico su OSX (variante Unix).
`.exe` , `.com`	File eseguibile di Windows. Creato collegando file di oggetti e file di libreria. Nei sistemi di tipo Unix, non esiste un'estensione di nome file speciale per il file eseguibile.

Bandiere del compilatore POSIX c99	Descrizione
`-o filename`	Nome del file di output es. (`bin/program.exe` , `program`)
`-I directory`	cercare intestazioni in `direrctory` .
`-D name`	definire il `name` macro
`-L directory`	cerca le biblioteche nella `directory` .
`-l name`	link library `libname` .

I compilatori su piattaforme POSIX (Linux, mainframe, Mac) di solito accettano queste opzioni, anche se non sono chiamate `c99` .

- Vedi anche c99 - compilare programmi C standard

Bandiere GCC (GNU Compiler Collection)	Descrizione
`-Wall`	Abilita tutti i messaggi di avviso comunemente accettati come utili.
`-Wextra`	Abilita più messaggi di avviso, può essere troppo rumoroso.
`-pedantic`	Forza avvisi in cui il codice viola lo standard scelto.
`-Wconversion`	Abilita gli avvisi sulla conversione implicita, usa con cautela.
`-c`	Compila i file sorgente senza collegamento.
`-v`	Stampa informazioni di compilazione.

- `gcc` accetta i flag POSIX più molti altri.
- Molti altri compilatori su piattaforme POSIX (`clang` , compilatori specifici del fornitore) utilizzano anche i flag elencati sopra.
- Vedi anche Invocazione di GCC per molte altre opzioni.

Bandiere TCC (Tiny C Compiler)	Descrizione
`-Wimplicit-function-declaration`	Avvisa sulla dichiarazione di funzione implicita.
`-Wunsupported`	Avvisa sulle funzionalità GCC non supportate ignorate da TCC.
`-Wwrite-strings`	Rendi costanti stringa di tipo const char * invece di char *.
`-Werror`	Compilazione di interruzione se vengono emessi avvisi.
`-Wall`	Attiva tutti gli avvisi, ad eccezione di `-Werror` , `-Wunusupported` e `-Wwrite strings` .

Examples

Il linker

Il lavoro del linker consiste nel collegare insieme un gruppo di file oggetto (file `.o`) in un eseguibile binario. Il processo di *collegamento* coinvolge principalmente la *risoluzione di indirizzi simbolici in indirizzi numerici* . Il risultato del processo di collegamento è normalmente un programma eseguibile.

Durante il processo di collegamento, il linker raccoglierà tutti i moduli oggetto specificati sulla riga di comando, aggiungerà un *codice di avvio* specifico del sistema e tenterà di risolvere tutti *i* riferimenti *esterni* nel modulo oggetto con *definizioni esterne* in altri file oggetto (file oggetto) può essere specificato direttamente sulla riga di comando o può essere aggiunto implicitamente tramite le librerie). Assegna quindi gli *indirizzi di carico* per i file oggetto, cioè specifica dove il codice e i dati finiranno nello spazio degli indirizzi del programma finito. Una volta che ha gli indirizzi di carico, può sostituire tutti gli indirizzi simbolici nel codice oggetto con indirizzi "reali" numerici nello spazio degli indirizzi del target. Il programma è pronto per essere eseguito ora.

Ciò include sia i file oggetto che il compilatore ha creato dai file del codice sorgente sia i file oggetto che sono stati precompilati per te e raccolti in file di libreria. Questi file hanno nomi che terminano in `.a` o `.so` , e normalmente non è necessario conoscerli, poiché il linker sa dove si trova la maggior parte di essi e li collegherà automaticamente in base alle necessità.

Invocazione implicita del linker

Come il pre-processore, il linker è un programma separato, spesso chiamato `ld` (ma Linux usa `collect2` , ad esempio). Analogamente al pre-processore, il linker viene richiamato automaticamente quando si utilizza il compilatore. Pertanto, il modo normale di utilizzare il linker è il seguente:

```
% gcc foo.o bar.o baz.o -o myprog
```

Questa riga indica al compilatore di collegare insieme tre file oggetto (`foo.o` , `bar.o` e `baz.o`) in un file eseguibile binario chiamato `myprog` . Ora hai un file chiamato `myprog` che puoi eseguire e che si spera possa fare qualcosa di interessante e / o utile.

Invocazione esplicita del linker

È possibile richiamare direttamente il linker, ma questo è raramente consigliabile ed è tipicamente molto specifico della piattaforma. Ovvero, le opzioni che funzionano su Linux non funzioneranno necessariamente su Solaris, AIX, macOS, Windows e in modo simile per qualsiasi altra piattaforma. Se lavori con GCC, puoi usare `gcc -v` per vedere cosa viene eseguito per tuo conto.

Opzioni per il linker

Il linker accetta anche alcuni argomenti per modificarne il comportamento. Il seguente comando direbbe a gcc di collegare `foo.o` e `bar.o` , ma include anche la libreria `ncurses` .

```
% gcc foo.o bar.o -o foo -lncurses
```

Questo è in realtà (più o meno) equivalente a

```
% gcc foo.o bar.o /usr/lib/libncurses.so -o foo
```

(anche se `libncurses.so` potrebbe essere `libncurses.a` , che è solo un archivio creato con `ar`). Si noti che è necessario elencare le librerie (tramite il nome del percorso o tramite `-lname` opzioni `-lname`) dopo i file oggetto. Con le librerie statiche, l'ordine in cui vengono specificate è importante; spesso, con le librerie condivise, l'ordine non ha importanza.

Si noti che su molti sistemi, se si utilizzano funzioni matematiche (da `<math.h>`), è necessario specificare `-lm` per caricare la libreria matematica, ma Mac OS X e macOS Sierra non richiedono questo. Ci sono altre librerie che sono librerie separate su Linux e altri sistemi Unix, ma non su macOS - thread POSIX e POSIX realtime, e le librerie di rete sono esempi. Di conseguenza, il processo di collegamento varia tra piattaforme.

Altre opzioni di compilazione

Questo è tutto ciò che devi sapere per iniziare a compilare i tuoi programmi C. In generale, ti consigliamo anche di utilizzare l' `-Wall` della `-Wall` comando `-Wall` :

```
% gcc -Wall -c foo.cc
```

L'opzione `-Wall` fa in modo che il compilatore ti avvisi su costrutti di codice legali ma dubbi e ti aiuterà a catturare molti bug molto presto.

Se vuoi che il compilatore lanci più avvertimenti su di te (comprese le variabili dichiarate ma non utilizzate, dimenticando di restituire un valore ecc.), Puoi usare questo insieme di opzioni, poiché `-Wall` , nonostante il nome, non gira *tutti i possibili avvertimenti* su:

```
% gcc -Wall -Wextra -Wfloat-equal -Wundef -Wcast-align -Wwrite-strings -Wlogical-op \
>     -Wmissing-declarations -Wredundant-decls -Wshadow …
```

Nota che `clang` ha un'opzione `-Weverything` che `-Weverything` realmente tutti gli avvertimenti in `clang` .

Tipi di file

Compilare i programmi C richiede di lavorare con cinque tipi di file:

1. **File di origine** : questi file contengono definizioni di funzioni e hanno nomi che terminano in `.c` per convenzione. Nota: `.cc` e `.cpp` sono file C ++; *non i* file C.
   ad esempio, `foo.c`

2. **File di intestazione** : questi file contengono prototipi di funzioni e varie istruzioni di pre-processore (vedi sotto). Vengono utilizzati per consentire ai file del codice sorgente di accedere a funzioni definite esternamente. I file di intestazione terminano in `.h` per convenzione.
   ad esempio, `foo.h`

3. **File oggetto** : questi file sono prodotti come output del compilatore. Sono costituiti da

definizioni di funzioni in formato binario, ma non sono eseguibili da soli. I file oggetto terminano in `.o` per convenzione, sebbene su alcuni sistemi operativi (es. Windows, MS-DOS), spesso finiscono in `.obj` .
ad esempio, `foo.o foo.obj`

4. **Eseguibili binari** : sono prodotti come output di un programma chiamato "linker". Il linker collega insieme un numero di file oggetto per produrre un file binario che può essere eseguito direttamente. Gli eseguibili binari non hanno suffisso speciale sui sistemi operativi Unix, sebbene generalmente finiscano in `.exe` su Windows.
   ad esempio, `foo foo.exe`

5. **Librerie** : una libreria è un binario compilato ma non è di per sé un eseguibile (cioè, non esiste una funzione `main()` in una libreria). Una libreria contiene funzioni che possono essere utilizzate da più di un programma. Una libreria dovrebbe essere spedita con i file header che contengono prototipi per tutte le funzioni nella libreria; questi file di intestazione dovrebbero essere referenziati (es .: `#include <library.h>`) in qualsiasi file sorgente che usi la libreria. Il linker deve quindi essere indirizzato alla libreria in modo che il programma possa essere compilato correttamente. Esistono due tipi di librerie: statiche e dinamiche.

   - **Libreria statica** : una libreria statica (file `.a` per sistemi POSIX e file `.lib` per Windows - da non confondere con i file di libreria di importazione DLL , che utilizzano anche l'estensione `.lib`) è integrata staticamente nel programma. Le librerie statiche hanno il vantaggio che il programma sa esattamente quale versione di una libreria viene utilizzata. D'altra parte, le dimensioni dei file eseguibili sono maggiori in quanto sono incluse tutte le funzioni di libreria utilizzate.
     ad esempio, `libfoo.a foo.lib`
   - **Libreria dinamica** : una libreria dinamica (file `.so` per la maggior parte dei sistemi POSIX, `.dylib` per OSX e file `.dll` per Windows) è collegata dinamicamente in fase di esecuzione dal programma. A volte vengono anche chiamate librerie condivise perché una sola immagine di libreria può essere condivisa da molti programmi. Le librerie dinamiche hanno il vantaggio di occupare meno spazio su disco se più di un'applicazione utilizza la libreria. Inoltre, consentono aggiornamenti della libreria (correzioni di bug) senza dover ricostruire eseguibili.
     ad esempio, `foo.so foo.dylib foo.dll`

Il preprocessore

Prima che il compilatore C inizi a compilare un file di codice sorgente, il file viene elaborato in una fase di pre-elaborazione. Questa fase può essere eseguita da un programma separato o essere completamente integrata in un eseguibile. In ogni caso, viene richiamato automaticamente dal compilatore prima che inizi la compilazione corretta. La fase di pre-elaborazione converte il codice sorgente in un altro codice sorgente o unità di traduzione applicando sostituzioni testuali. Puoi considerarlo come un codice sorgente "modificato" o "espanso". Questa fonte espansa può esistere come file reale nel file system, oppure può essere memorizzata in memoria per un breve periodo prima di essere ulteriormente elaborata.

I comandi del preprocessore iniziano con il cancelletto ("#"). Esistono diversi comandi per il

preprocessore; due dei più importanti sono:

1. **Definisce** :

   `#define` è utilizzato principalmente per definire le costanti. Per esempio,

   ```
   #define BIGNUM 1000000
   int a = BIGNUM;
   ```

   diventa

   ```
   int a = 1000000;
   ```

   `#define` viene utilizzato in questo modo in modo da evitare di dover scrivere esplicitamente un valore costante in molte posizioni diverse in un file di codice sorgente. Questo è importante nel caso in cui sia necessario modificare il valore costante in seguito; è molto meno incline a cambiarlo una volta, in `#define` , piuttosto che `#define` modificare in più punti sparsi su tutto il codice.

   Poiché `#define` esegue solo la ricerca avanzata e la sostituzione, puoi anche dichiarare macro. Per esempio:

   ```
   #define ISTRUE(stm) do{stm = stm ? 1 : 0;}while(0)
   // in the function:
   a = x;
   ISTRUE(a);
   ```

   diventa:

   ```
   // in the function:
   a = x;
   do {
       a = a ? 1 : 0;
   } while(0);
   ```

   In prima approssimazione, questo effetto è più o meno lo stesso delle funzioni inline, ma il preprocessore non fornisce il controllo di tipo per le macro `#define` . Questo è ben noto per essere soggetto a errori e il loro uso richiede molta cautela.

   Si noti inoltre che il preprocessore dovrebbe anche sostituire i commenti con uno spazio vuoto come spiegato di seguito.

2. **Include** :

   `#include` viene utilizzato per accedere alle definizioni di funzioni definite al di fuori di un file di codice sorgente. Per esempio:

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

   causa al preprocessore di incollare il contenuto di `<stdio.h>` nel file di codice sorgente nel

punto `#include` prima che venga compilato. `#include` è quasi sempre usato per includere i file header, che sono i file che contengono principalmente dichiarazioni di funzioni e dichiarazioni `#define` . In questo caso, usiamo `#include` per poter utilizzare funzioni come `printf` e `scanf` , le cui dichiarazioni si trovano nel file `stdio.h` . I compilatori C non ti permettono di usare una funzione a meno che non sia stata precedentemente dichiarata o definita in quel file; `#include` affermazioni sono quindi il modo di riutilizzare il codice scritto in precedenza nei vostri programmi C.

3. **Operazioni logiche** :

```
#if defined A || defined B
variable = another_variable + 1;
#else
variable = another_variable * 2;
#endif
```

sarà cambiato in:

```
variable = another_variable + 1;
```

se A o B sono stati definiti da qualche parte nel progetto prima. Se questo non è il caso, ovviamente il preprocessore farà questo:

```
variable = another_variable * 2;
```

Questo è spesso usato per il codice, che funziona su diversi sistemi o compila su diversi compilatori. Dato che ci sono definizioni globali, che sono specifiche del compilatore / sistema, è possibile testare quelle che definiscono e lasciare sempre che il compilatore usi il codice che compilerà di sicuro.

4. **Commenti**

Il preprocessore sostituisce tutti i commenti nel file di origine per singoli spazi. I commenti sono indicati da `//` fino alla fine della riga, o una combinazione di parentesi di apertura `/*` e chiusura `*/` commento.

Il compilatore

Dopo che il pre-processore C ha incluso tutti i file header e espanso tutte le macro, il compilatore può compilare il programma. Lo fa girando il codice sorgente C in un file di codice oggetto, che è un file che termina in `.o` che contiene la versione binaria del codice sorgente. Il codice oggetto non è direttamente eseguibile, però. Per rendere un eseguibile, devi anche aggiungere il codice per tutte le funzioni della libreria che erano `#include` d nel file (non è lo stesso che includere le dichiarazioni, che è ciò che `#include`). Questo è il lavoro del linker .

In generale, la sequenza esatta su come richiamare un compilatore C dipende molto dal sistema che si sta utilizzando. Qui stiamo usando il compilatore GCC, sebbene si noti che esistono molti altri compilatori:

```
% gcc -Wall -c foo.c
```

`%` è il prompt dei comandi del sistema operativo. Questo dice al compilatore di eseguire il pre-processore sul file `foo.c` e quindi di compilarlo nel file di codice oggetto `foo.o` L'opzione `-c` significa compilare il file del codice sorgente in un file oggetto ma non invocare il linker. Questa opzione `-c` è disponibile su sistemi POSIX, come Linux o macOS; altri sistemi possono utilizzare una sintassi diversa.

Se l'intero programma si trova in un file di codice sorgente, puoi farlo invece:

```
% gcc -Wall foo.c -o foo
```

Questo dice al compilatore di eseguire il pre-processore su `foo.c` , compilarlo e quindi collegarlo per creare un eseguibile chiamato `foo` . L'opzione `-o` indica che la parola successiva sulla riga è il nome del file eseguibile binario (programma). Se non si specifica `-o` , (se si digita semplicemente `gcc foo.c`), l'eseguibile verrà denominato `a.out` per ragioni storiche.

In generale, il compilatore richiede quattro passaggi durante la conversione di un file `.c` in un eseguibile:

1. **pre-elaborazione** - espande testualmente le direttive `#include` e `#define` nel tuo file `.c`
2. **compilation** - converte il programma in assembly (puoi fermare il compilatore in questa fase aggiungendo l'opzione `-S`)
3. **assembly** - converte l'assembly in codice macchina
4. **linkage** : collega il codice oggetto alle librerie esterne per creare un eseguibile

Nota anche che il nome del compilatore che stiamo usando è GCC, che sta per "GNU C compiler" e "GNU compiler collection", a seconda del contesto. Esistono altri compilatori C. Per i sistemi operativi di tipo Unix, molti di essi hanno il nome `cc` , per "C compilatore", che è spesso un collegamento simbolico con qualche altro compilatore. Sui sistemi Linux, `cc` è spesso un alias per GCC. Su macOS o OS-X, punta al clang.

Gli standard POSIX attualmente `c99` come nome di un compilatore C - per impostazione predefinita supporta lo standard C99. Versioni precedenti di POSIX con mandato `c89` come compilatore. POSIX impone anche che questo compilatore comprenda le opzioni `-c` `-o` che abbiamo usato sopra.

Nota: l'opzione `-Wall` presente in entrambi gli esempi di `gcc` indica al compilatore di stampare avvisi su costruzioni discutibili, che è fortemente raccomandato. È anche una buona idea aggiungere altre opzioni di avviso , ad esempio `-Wextra` .

Le fasi di traduzione

A partire dallo Standard C 2011, elencato in *§5.1.1.2 Fasi di traduzione* , la traduzione del codice sorgente nell'immagine del programma (ad esempio, l'eseguibile) è elencata per essere *eseguita* in 8 passaggi ordinati.

1. L'input del file sorgente viene mappato sul set di caratteri sorgente (se necessario). Trigrams sono sostituiti in questo passaggio.
2. Le linee di continuazione (le righe che terminano con `\`) sono giuntate con la riga successiva.
3. Il codice sorgente viene analizzato in spazi bianchi e token di preelaborazione.
4. Viene applicato il preprocessore, che esegue le direttive, espande i macro e applica i pragma. Ogni file sorgente inserito da `#include` viene sottoposto alle fasi di conversione da 1 a 4 (se necessario ricorsivo). Tutte le direttive relative al preprocessore vengono quindi eliminate.
5. I valori del set di caratteri di origine in costanti di caratteri e valori letterali stringa vengono associati al set di caratteri di esecuzione.
6. I letterali delle stringhe adiacenti l'uno all'altro sono concatenati.
7. Il codice sorgente viene analizzato in token, che comprendono l'unità di traduzione.
8. I riferimenti esterni vengono risolti e viene creata l'immagine del programma.

Un'implementazione di un compilatore C può combinare più passaggi insieme, ma l'immagine risultante deve comunque comportarsi come se i passaggi precedenti si fossero verificati separatamente nell'ordine sopra elencato.

Leggi Compilazione online: https://riptutorial.com/it/c/topic/1337/compilazione

Capitolo 15: Comportamento definito dall'implementazione

Osservazioni

Panoramica

Lo standard C descrive la sintassi del linguaggio, le funzioni fornite dalla libreria standard e il comportamento di processori C conformi (in parole povere, compilatori) e programmi C conformi. Per quanto riguarda il comportamento, lo standard per la maggior parte specifica comportamenti particolari per programmi e processori. D'altra parte, alcune operazioni hanno un *comportamento indefinito* esplicito o implicito - tali operazioni devono sempre essere evitate, poiché non si può fare affidamento su nulla su di esse. Nel mezzo, ci sono una varietà di comportamenti *definiti dall'implementazione* . Questi comportamenti possono variare tra processori C, runtime e librerie standard (collettivamente, *implementazioni*), ma sono coerenti e affidabili per ogni specifica implementazione e le implementazioni conformi documentano il loro comportamento in ciascuna di queste aree.

Talvolta è ragionevole che un programma faccia affidamento su un comportamento definito dall'implementazione. Ad esempio, se il programma è comunque specifico per un particolare ambiente operativo, è improbabile che fare affidamento su comportamenti generali definiti dall'implementazione per i processori comuni per quell'ambiente sia un problema. In alternativa, è possibile utilizzare direttive di compilazione condizionale per selezionare comportamenti definiti dall'implementazione appropriati per l'implementazione in uso. In ogni caso, è essenziale sapere quali operazioni hanno un comportamento definito dall'implementazione, in modo da evitarle o prendere una decisione informata su se e come usarle.

Il bilancio di queste osservazioni costituisce un elenco di tutti i comportamenti e le caratteristiche definiti dall'implementazione specificati nello standard C2011, con riferimenti allo standard. Molti di loro usano la terminologia dello standard . Alcuni altri si basano più generalmente sul contesto dello standard, come le otto fasi di traduzione del codice sorgente in un programma o la differenza tra implementazioni ospitate e indipendenti. Alcuni che possono essere particolarmente sorprendenti o notevoli sono presentati in grassetto. Non tutti i comportamenti descritti sono supportati da precedenti standard C, ma, in generale, hanno un comportamento definito dall'implementazione in tutte le versioni dello standard che li supportano.

Programmi e processori

Generale

- **Il numero di bit in un byte** (3.6 / 3). Almeno `8` , il valore attuale può essere interrogato con la macro `CHAR_BIT` .

- Quali messaggi di output sono considerati "messaggi diagnostici" (3.10 / 1)

Traduzione di origine

- Il modo in cui i caratteri multibyte del file sorgente fisico sono mappati sul set di caratteri sorgente (5.1.1.2/1).
- Se le sequenze non vuote di spazi non di nuova riga sono sostituite da spazi singoli durante la fase di traduzione 3 (5.1.1.2/1)
- I caratteri del set di esecuzioni a cui i caratteri letterali e i caratteri delle costanti di stringa vengono convertiti (durante la fase di traduzione 5) quando non esiste altrimenti un carattere corrispondente (5.1.1.2/1).

Ambiente operativo

- Il modo in cui vengono identificati i messaggi diagnostici da emettere (5.1.1.3/1).
- Il nome e il tipo della funzione chiamata all'avvio in un'implementazione indipendente (5.1.2.1/1).
- Quali strutture di libreria sono disponibili in un'implementazione indipendente, oltre un set minimo specificato (5.1.2.1/1).
- L'effetto della terminazione del programma in un ambiente indipendente (5.1.2.1/2).
- In un ambiente ospitato, qualsiasi firma consentita per la funzione `main()` diversa da `int main(int argc, char *arg[])` e `int main(void)` (5.1.2.2.1 / 1).
- Il modo in cui un'implementazione in hosting definisce le stringhe puntate dal secondo argomento a `main()` (5.1.2.2.1 / 2).
- Cosa costituisce un "dispositivo interattivo" ai fini delle sezioni 5.1.2.3 (Esecuzione del programma) e 7.21.3 (File) (5.1.2.3/7).
- Qualsiasi restrizione sugli oggetti a cui fanno riferimento le routine di gestione degli interrupt in un'implementazione di ottimizzazione (5.1.2.3/10).
- In un'implementazione indipendente, sono supportati più thread di esecuzione (5.1.2.4/1).
- I valori dei membri del set di caratteri di esecuzione (5.2.1 / 1).
- I valori `char` corrispondenti alle sequenze di escape alfabetiche definite (5.2.2 / 3).
- **I limiti numerici e le caratteristiche a virgola mobile e** intera (5.2.4.2/1).
- Precisione delle operazioni aritmetiche in virgola mobile e delle conversioni della libreria standard dalle rappresentazioni in virgola mobile interne alle rappresentazioni di stringhe (5.2.4.2.2 / 6).

- Il valore della macro `FLT_ROUNDS` , che codifica la modalità di arrotondamento a virgola mobile predefinita (5.2.4.2.2 / 8).
- I comportamenti di arrotondamento caratterizzati da valori supportati di `FLT_ROUNDS` superiori a 3 o inferiori a -1 (5.2.4.2.2 / 8).
- Il valore della macro `FLT_EVAL_METHOD` , che caratterizza il comportamento di valutazione a virgola mobile (5.2.4.2.2 / 9).
- Il comportamento caratterizzato da qualsiasi valore supportato di `FLT_EVAL_METHOD` inferiore a -1 (5.2.4.2.2 / 9).
- I valori delle macro `FLT_HAS_SUBNORM` , `DBL_HAS_SUBNORM` e `LDBL_HAS_SUBNORM` , che caratterizzano se i formati standard in virgola mobile supportano numeri subnormali (5.2.4.2.2 / 10)

tipi

- Il risultato del tentativo di accedere (indirettamente) a un oggetto con durata di memorizzazione del thread da un thread diverso da quello a cui è associato l'oggetto (6.2.4 / 4)
- Il valore di un `char` a cui è stato assegnato un carattere al di fuori del set di esecuzione di base (6.2.5 / 3).
- I tipi interi con segno esteso supportati, se presenti, (6.2.5 / 4) e le eventuali parole chiave di estensione utilizzate per identificarli.
- **Se `char` ha la stessa rappresentazione e comportamento del `signed char` o del `unsigned char`** (6.2.5 / 15). Può essere interrogato con `CHAR_MIN` , che è `0` o `SCHAR_MIN` se `char` non è firmato o firmato, rispettivamente.
- **Il numero, l'ordine e la codifica dei byte nelle rappresentazioni di oggetti** , tranne dove specificato esplicitamente dallo standard (6.2.6.1/2).
- **Quale delle tre forme riconosciute di rappresentazione intera si applica in una data situazione e se determinati modelli di bit di oggetti interi sono rappresentazioni di trap** (6.2.6.2/2).
- Il requisito di allineamento di ciascun tipo (6.2.8 / 1).
- Se e in quali contesti sono supportati gli allineamenti estesi (6.2.8 / 3).
- Il set di allineamenti estesi supportati (6.2.8 / 4).
- I ranghi interi di conversione di tutti i tipi interi con segno esteso relativi tra loro (6.3.1.1/1).
- **L'effetto dell'assegnazione di un valore fuori intervallo a un intero con** segno (6.3.1.3/3).
- Quando un valore in-range ma non rappresentabile è assegnato a un oggetto a virgola

mobile, come viene scelto il valore rappresentabile memorizzato nell'oggetto tra i due valori rappresentabili più vicini (6.3.1.4/2 ; 6.3.1.5/1 ; 6.4.4.2 / 3).

- **Il risultato della conversione di un intero in un tipo di puntatore** , ad eccezione delle espressioni costanti intere con valore `0` (6.3.2.3/5).

Modulo di origine

- Le posizioni all'interno delle direttive `#pragma` cui sono riconosciuti i token del nome dell'intestazione (6.4 / 4).
- I caratteri, inclusi i caratteri multibyte, diversi dal carattere di sottolineatura, lettere latine non accentate, nomi di caratteri universali e cifre decimali che possono apparire negli identificatori (6.4.2.1/1).
- **Il numero di caratteri significativi in un identificatore** (6.4.2.1/5).
- Con alcune eccezioni, il modo in cui i caratteri di origine in una costante di carattere intero sono mappati ai caratteri del set di esecuzione (6.4.4.4/2 ; 6.4.4.4/10).
- Le impostazioni locali correnti utilizzate per calcolare il valore di una costante di carattere ampia e la maggior parte degli altri aspetti della conversione per molte di queste costanti (6.4.4.4/11).
- Se i token letterali a stringa ampia con prefisso diverso possono essere concatenati e, in tal caso, il trattamento della sequenza di caratteri multibyte risultante (6.4.5 / 5)
- Le impostazioni internazionali utilizzate durante la fase di conversione 7 per convertire i letterali di stringa ampia in sequenze di caratteri multibyte e il loro valore quando il risultato non è rappresentabile nel set di caratteri di esecuzione (6.4.5 / 6).
- Il modo in cui i nomi delle intestazioni sono mappati ai nomi dei file (6.4.7 / 2).

Valutazione

- Se e in che modo le espressioni floating-point sono contratte quando `FP_CONTRACT` non viene utilizzato (6.5 / 8).
- **I valori dei risultati degli operatori `sizeof` e `_Alignof`** (6.5.3.4/5).
- La dimensione del tipo di risultato della sottrazione del puntatore (6.5.6 / 9).
- **Il risultato di spostare a destra un numero intero con** segno **con un valore negativo** (6.5.7 / 5).

Comportamento di runtime

- La misura in cui la parola chiave del `register` è efficace (6.7.1 / 6).

- Se il tipo di un bitfield dichiarato come `int` è dello stesso tipo di `unsigned int` o `signed int` (6.7.2 / 5).
- Quali tipi di bitfield possono essere utilizzati, ad eccezione di `_Bool` qualificato `_Bool` , `signed int` e `unsigned int` ; se i bitfield possono avere tipi atomici (6.7.2.1/5).
- Aspetti di come le implementazioni definiscono lo spazio per i bitfield (6.7.2.1/11).
- L'allineamento dei membri non bitfield di strutture e sindacati (6.7.2.1/14).
- Il tipo sottostante per ogni tipo enumerato (6.7.2.2/4).
- Che cosa costituisce un "accesso" a un oggetto di tipo `volatile` -qualificato (6.7.3 / 7).
- L'efficacia delle dichiarazioni di funzioni `inline` (6.7.4 / 6).

preprocessore

- Se le costanti di caratteri vengono convertite in valori interi allo stesso modo in condizionali del preprocessore come in espressioni ordinarie e se una costante a carattere singolo può avere un valore negativo (6.10.1 / 4).
- Le posizioni hanno cercato i file designati in una direttiva `#include` (6.10.2 / 2-3).
- Il modo in cui un nome di intestazione è formato dai token di una direttiva `#include` multi-token (6.10.2 / 4).
- Il limite per `#include` nesting (6.10.2 / 6).
- Se un carattere `\` è inserito prima di `\` introdurre un nome di carattere universale nel risultato dell'operatore `#` del preprocessore (6.10.3.2/2).
- Il comportamento della direttiva di pre-elaborazione `#pragma` le direttive diverse da `STDC` (`STDC` / 1).
- Il valore delle macro `__DATE__` e `__TIME__` se non sono disponibili né la data né l'ora della traduzione (6.10.8.1/1).
- La codifica dei caratteri interni utilizzata per `wchar_t` se la macro `__STDC_ISO_10646__` non è definita (6.10.8.2/1).
- La codifica dei caratteri interni utilizzata per `char32_t` se la macro `__STDC_UTF_32__` non è definita (6.10.8.2/1).

Libreria standard

Generale

- Il formato dei messaggi emessi quando le asserzioni falliscono (7.2.1.1/2).

Funzioni ambientali a virgola mobile

- Eventuali eccezioni a virgola mobile aggiuntive oltre a quelle definite dallo standard (7.6 / 6).
- Eventuali modalità di arrotondamento a virgola mobile aggiuntive oltre a quelle definite dallo standard (7.6 / 8).
- Eventuali ambienti aggiuntivi a virgola mobile oltre a quelli definiti dallo standard (7.6 / 10).
- Il valore predefinito dell'interruttore di accesso all'ambiente in virgola mobile (7.6.1 / 2).
- La rappresentazione dei flag di stato a virgola mobile registrati da `fegetexceptflag()` (7.6.2.2/1).
- Se la funzione `feraiseexcept()` solleva ulteriormente l'eccezione "inesatta" a virgola mobile ogni volta che solleva l'eccezione "overflow" o "underflow" a virgola mobile (7.6.2.3/2).

Funzioni relative alle impostazioni locali

- Le stringhe locali diverse da `"C"` supportate da `setlocale()` (7.11.1.1/3).

Funzioni matematiche

- I tipi rappresentati da `float_t` e `double_t` quando la macro `FLT_EVAL_METHOD` ha un valore diverso da `0` , `1` e `2` (7.12 / 2).
- Qualsiasi classificazione a virgola mobile supportata oltre a quelle definite dallo standard (7.12 / 6).
- Il valore restituito dalle funzioni `math.h` in caso di errore di dominio (7.12.1 / 2).
- Il valore restituito dalle funzioni `math.h` in caso di errore polare (7.12.1 / 3).
- Il valore restituito dalle funzioni `math.h` quando il risultato è underflow, e gli aspetti se `errno` è impostato su `ERANGE` e se viene sollevata un'eccezione a virgola mobile in tali circostanze (7.12.1 / 6).
- Il valore predefinito dell'interruttore di contrazione FP (7.12.2 / 2).
- Se le funzioni `fmod()` restituiscono 0 o `fmod()` un errore di dominio quando il loro secondo argomento è 0 (7.12.10.1/3).
- Se le funzioni `remainder()` restituiscono 0 o generano un errore di dominio quando il loro secondo argomento è 0 (7.12.10.2/3).
- Il numero di bit significativi nei moduli dei quozienti calcolati dalle funzioni `remquo()` (7.12.10.3/2).
- Se le funzioni `remquo()` restituiscono 0 o `remquo()` un errore di dominio quando il loro secondo

argomento è 0 (7.12.10.3/3).

segnali

- Il set completo di segnali supportati, la loro semantica e la loro gestione predefinita (7.14 / 4).
- Quando viene alzato un segnale e vi è un gestore personalizzato associato a quel segnale, i segnali, se presenti, vengono bloccati per la durata dell'esecuzione dell'handler (7.14.1.1/3).
- Quali segnali diversi da `SIGFPE` , `SIGILL` e `SIGSEGV` sì che il comportamento al ritorno da un gestore di segnale personalizzato non sia definito (7.14.1.1/3).
- Quali segnali sono inizialmente configurati per essere ignorati (indipendentemente dalla loro gestione predefinita, 7.14.1.1/6).

miscellaneo

- La costante puntatore nullo specifica a cui si espande la macro `NULL` (7.19 / 3).

Funzioni di gestione dei file

- Se l'ultima riga di un flusso di testo richiede una terminazione finale (7.21.2 / 2).
- Il numero di caratteri nulli aggiunti automaticamente a un flusso binario (7.21.2 / 3).
- La posizione iniziale di un file aperta in modalità append (7.21.3 / 1).
- Se una scrittura su un flusso di testo provoca il troncamento del flusso (7.21.3 / 2).
- Supporto per il buffer del flusso (7.21.3 / 3).
- Esistono effettivamente file a lunghezza zero (7.21.3 / 4).
- Le regole per la composizione di nomi di file validi (7.21.3 / 8).
- Se lo stesso file può essere aperto contemporaneamente più volte (7.21.3 / 8).
- La natura e la scelta della codifica per i caratteri multibyte (7.21.3 / 10).
- Il comportamento della funzione `remove()` quando il file di destinazione è aperto (7.21.4.1/2).
- Il comportamento della funzione `rename()` quando esiste già il file di destinazione (7.21.4.2/2).
- Se i file creati tramite la funzione `tmpfile()` vengono rimossi nel caso in cui il programma termini in modo anomalo (7.21.4.3/2).
- Quale modalità cambia in base alle circostanze consentite tramite `freopen()` (7.21.5.4/3).

Funzioni I / O

- Quale tra le rappresentazioni consentite di valori FP infiniti e non-un numero è prodotta dalle funzioni family printf () (7.21.6.1/8).
- Il modo in cui i puntatori sono formattati dalle funzioni `printf()` -family (7.21.6.1/8).
- Il comportamento di `scanf()` -family funziona quando il carattere `-` appare in una posizione interna della lista di scansione di un `[` campo (7.21.6.2/12).
- La maggior parte degli aspetti del passaggio di funzioni `p` di `scanf()` -family (7.21.6.2/12).
- Il valore `errno` impostato da `fgetpos()` in caso di errore (7.21.9.1/2).
- Il valore `errno` impostato da `fsetpos()` in caso di errore (7.21.9.3/2).
- Il valore `errno` impostato da `ftell()` in caso di errore (7.21.9.4/3).
- Il significato delle `strtod()` di alcuni aspetti supportati di una formattazione NaN (7.22.1.3p4).
- Se le `strtod()` impostano `errno` su `ERANGE` quando il risultato è underflow (7.22.1.3/10).

Funzioni di allocazione della memoria

- Il comportamento delle funzioni di allocazione della memoria quando il numero di byte richiesti è 0 (7.22.3 / 1).

Funzioni dell'ambiente di sistema

- Quali puliture, se ve ne sono, vengono eseguite e quale stato viene restituito al SO host quando viene chiamata la funzione `abort()` (7.22.4.1/2).
- Quale stato viene restituito all'ambiente host quando viene chiamato `exit()` (7.22.4.4/5).
- La gestione degli stream aperti e quale stato viene restituito all'ambiente host quando viene chiamato `_Exit()` (7.22.4.5/2).
- L'insieme di nomi di ambiente accessibili tramite `getenv()` e il metodo per alterare l'ambiente (7.22.4.6/2).
- Il valore restituito dalla funzione `system()` (7.22.4.8/3).

Funzioni di data e ora

- Il fuso orario locale e l'ora legale (7.27.1 / 1).
- La gamma e la precisione degli orari rappresentabili tramite i tipi `clock_t` e `time_t` (7.27.1 / 4).

- L'inizio dell'era che funge da riferimento per i tempi restituiti dalla funzione `clock()` (7.27.2.1/3).
- L'inizio dell'epoca che funge da riferimento per i tempi restituiti dalla funzione `timespec_get()` (quando la base dei tempi è `TIME_UTC` ; 7.27.2.5/3).
- La sostituzione `strftime()` per lo specificatore di conversione `%Z` nella locale "C" (7.27.3.5/7).

Funzioni I / O a caratteri ampi

- Quale tra le rappresentazioni consentite di valori FP infiniti e non-un numero è prodotta dalle `wprintf()` -famiglia (7.29.2.1/8).
- Il modo in cui i puntatori sono formattati dalle `wprintf()` funzioni -Family (7.29.2.1/8).
- Il comportamento di `wscanf()` -family funziona quando il carattere `-` appare in una posizione interna della lista di scansione di un `[` campo (7.29.2.2/12).
- La maggior parte degli aspetti dei `wscanf()` -family functions ' `p` (7.29.2.2/12).
- Il significato delle `wstrtod()` di alcuni aspetti supportati della formattazione NaN (7.29.4.1.1 / 4).
- Se le `wstrtod()` impostano `errno` su `ERANGE` quando il risultato è underflow (7.29.4.1.1 / 10).

Examples

Spostamento a destra di un numero intero negativo

```
int signed_integer = -1;

// The right shift operation exhibits implementation-defined behavior:
int result = signed_integer >> 1;
```

Assegnazione di un valore fuori intervallo a un numero intero

```
// 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;
```

Assegnazione di zero byte

```
// The allocation functions have implementation-defined behavior when the requested size
// of the allocation is zero.
void *p = malloc(0);
```

Rappresentazione di interi con segno

Ogni tipo di intero con segno può essere rappresentato in uno qualsiasi dei tre formati; è definito dall'implementazione quale viene utilizzato. L'implementazione in uso per qualsiasi dato intero con segno di almeno ampiezza come `int` può essere determinata in fase di esecuzione dai due bit di ordine inferiore della rappresentazione del valore `-1` in quel tipo, in questo modo:

```
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"); }
}
```

Lo stesso modello si applica alla rappresentazione di tipi più stretti, ma non possono essere testati con questa tecnica perché gli operandi di `&` sono soggetti alle "solite conversioni aritmetiche" prima che il risultato sia calcolato.

Leggi Comportamento definito dall'implementazione online:
https://riptutorial.com/it/c/topic/4832/comportamento-definito-dall-implementazione

Capitolo 16: Comportamento non definito

introduzione

In C, alcune espressioni producono un *comportamento indefinito* . Lo standard sceglie esplicitamente di non definire come dovrebbe comportarsi un compilatore se incontra una tale espressione. Di conseguenza, un compilatore è libero di fare tutto ciò che ritiene opportuno e può produrre risultati utili, risultati inaspettati o addirittura incidenti.

Il codice che richiama UB può funzionare come previsto su un sistema specifico con uno specifico compilatore, ma probabilmente non funzionerà su un altro sistema, o con un compilatore, una versione del compilatore o delle impostazioni del compilatore diversi.

Osservazioni

Che cos'è Undefined Behavior (UB)?

Il comportamento non definito è un termine usato nello standard C. Lo standard C11 (ISO / IEC 9899: 2011) definisce il termine comportamento indefinito come

> comportamento, sull'uso di un costrutto di programma non portatile o errato o di dati errati, per i quali questo Standard internazionale non impone requisiti

Cosa succede se c'è UB nel mio codice?

Questi sono i risultati che possono verificarsi a causa di un comportamento indefinito secondo lo standard:

> NOTA Il possibile comportamento indefinito va dall'ignorare completamente la situazione con risultati imprevedibili, a comportarsi durante la traduzione o l'esecuzione del programma in un modo documentato caratteristico dell'ambiente (con o senza emissione di un messaggio diagnostico), a terminare una traduzione o un'esecuzione (con il emissione di un messaggio diagnostico).

La seguente citazione è spesso usata per descrivere (meno formalmente però) risultati derivanti da un comportamento non definito:

> "Quando il compilatore incontra [un dato costrutto indefinito] è legale che faccia volare fuori i demoni dal naso" (l'implicazione è che il compilatore può scegliere qualsiasi modo arbitrariamente bizzarro per interpretare il codice senza violare lo standard ANSI C)

Perché esiste UB?

Se è così brutto, perché non lo hanno appena definito o reso definitivo?

Il comportamento indefinito consente maggiori opportunità di ottimizzazione; Il compilatore può

legittimamente presumere che qualsiasi codice non contenga un comportamento indefinito, che può consentirgli di evitare i controlli di run-time ed eseguire ottimizzazioni la cui validità sarebbe costosa o impossibile da dimostrare in altro modo.

Perché UB è difficile da rintracciare?

Ci sono almeno due ragioni per cui un comportamento indefinito crea bug difficili da rilevare:

- Il compilatore non è tenuto a - e generalmente non può in modo affidabile - avvisarti riguardo al comportamento non definito. In effetti, la sua richiesta di farlo andrebbe direttamente contro la ragione dell'esistenza di un comportamento non definito.
- I risultati imprevedibili potrebbero non iniziare a svolgersi nel punto esatto dell'operazione in cui si verifica il costrutto il cui comportamento non è definito; Il comportamento indefinito rende l'intera esecuzione e i suoi effetti possono verificarsi in qualsiasi momento: durante, dopo, o anche *prima* del costrutto indefinito.

Considera la dereferenziazione del puntatore nullo: il compilatore non è obbligato a diagnosticare il dereferenziamento del puntatore nullo, e nemmeno potrebbe farlo, poiché in fase di runtime qualsiasi puntatore passato in una funzione o in una variabile globale potrebbe essere nullo. *E quando si verifica la dereferenza del puntatore nullo, lo standard non impone il crash del programma.* Piuttosto, il programma potrebbe bloccarsi prima, dopo o non arrestarsi affatto; potrebbe anche comportarsi come se il puntatore nullo indicasse un oggetto valido, e si comportasse in modo completamente normale, solo per bloccarsi in altre circostanze.

Nel caso della dereferenziazione del puntatore nullo, il linguaggio C differisce dai linguaggi gestiti come Java o C #, in cui viene *definito* il comportamento di dereferenziazione del puntatore nullo: viene generata un'eccezione (`NullPointerException` in Java, `NullReferenceException` in C #) , quindi quelli provenienti da Java o C # potrebbero *erroneamente credere che in tal caso, un programma C deve arrestarsi in modo anomalo, con o senza l'emissione di un messaggio diagnostico* .

Informazioni aggiuntive

Ci sono diverse situazioni simili che dovrebbero essere chiaramente distinte:

- Comportamento esplicitamente indefinito, cioè dove lo standard C ti dice esplicitamente che sei fuori dai limiti.
- Comportamento implicitamente indefinito, in cui non esiste semplicemente un testo nello standard che preveda un comportamento per la situazione in cui hai inserito il tuo programma.

Inoltre, tenere presente che in molti luoghi il comportamento di determinati costrutti è deliberatamente indefinito dallo standard C per lasciare spazio al compilatore e agli implementatori di librerie per elaborare le proprie definizioni. Un buon esempio sono i segnali e i gestori di segnale, in cui le estensioni a C, come lo standard del sistema operativo POSIX, definiscono regole molto più elaborate. In questi casi devi solo controllare la documentazione della tua piattaforma; lo standard C non può dirti nulla.

Si noti inoltre che se si verifica un comportamento non definito nel programma, ciò non significa che solo il punto in cui si è verificato il comportamento non definito è problematico, piuttosto

l'intero programma diventa privo di significato.

A causa di tali preoccupazioni è importante (soprattutto dal momento che i compilatori non ci avvisano sempre di UB) perché la programmazione in C di una persona abbia almeno familiarità con il tipo di cose che scatenano un comportamento indefinito.

Va notato che esistono alcuni strumenti (ad esempio strumenti di analisi statica come PC-Lint) che aiutano a rilevare comportamenti non definiti, ma, ancora una volta, non sono in grado di rilevare tutte le occorrenze di comportamenti non definiti.

Examples

Dereferenziamento di un puntatore nullo

Questo è un esempio di dereferenziazione di un puntatore NULL, che causa un comportamento indefinito.

```
int * pointer = NULL;
int value = *pointer; /* Dereferencing happens here */
```

Un puntatore `NULL` è garantito dallo standard C per confrontare non uguali a qualsiasi puntatore a un oggetto valido e il dereferenziazione richiama un comportamento non definito.

Modifica di qualsiasi oggetto più di una volta tra due punti di sequenza

```
int i = 42;
i = i++; /* Assignment changes variable, post-increment as well */
int a = i++ + i--;
```

Codice come questo spesso porta a speculazioni sul "valore risultante" di `i` . Piuttosto che specificare un risultato, tuttavia, gli standard C specificano che la valutazione di tale espressione produce un *comportamento indefinito* . Prima del C2011, lo standard ha formalizzato queste regole in termini di cosiddetti *punti di sequenza* :

> Tra il punto di sequenza precedente e quello successivo, un oggetto scalare deve avere il suo valore memorizzato modificato al massimo una volta dalla valutazione di un'espressione. Inoltre, il valore precedente deve essere letto solo per determinare il valore da memorizzare.

(Standard C99, sezione 6.5, paragrafo 2)

Questo schema si è dimostrato un po 'troppo rozzo, con il risultato che alcune espressioni esibivano un comportamento indefinito rispetto al C99 che plausibilmente non dovrebbe fare. C2011 conserva punti di sequenza, ma introduce un approccio più sfumato a quest'area basato sul *sequenziamento* e una relazione che chiama "sequenziata prima":

> Se un effetto collaterale su un oggetto scalare è ingiustificato rispetto a un effetto collaterale diverso sullo stesso oggetto scalare o un calcolo del valore che utilizza il

> valore dello stesso oggetto scalare, il comportamento non è definito. Se esistono più ordinamenti consentiti delle sottoespressioni di un'espressione, il comportamento non è definito se si verifica un effetto collaterale non eseguito in uno degli ordini.

(Standard C2011, sezione 6.5, paragrafo 2)

I dettagli completi della relazione "sequenziata prima" sono troppo lunghi per essere descritti qui, ma integrano i punti di sequenza anziché sostituirli, quindi hanno l'effetto di definire il comportamento per alcune valutazioni il cui comportamento precedentemente non era definito. In particolare, se c'è un punto di sequenza tra due valutazioni, allora quella prima del punto di sequenza è "sequenziata prima" di quella successiva.

L'esempio seguente ha un comportamento ben definito:

```
int i = 42;
i = (i++, i+42); /* The comma-operator creates a sequence point */
```

L'esempio seguente ha un comportamento non definito:

```
int i = 42;
printf("%d %d\n", i++, i++); /* commas as separator of function arguments are not comma-
operators */
```

Come in ogni forma di comportamento non definito, l'osservazione del comportamento effettivo della valutazione di espressioni che violano le regole di sequenziamento non è informativa, se non in senso retrospettivo. Lo standard linguistico non fornisce alcuna base per prevedere che tali osservazioni siano predittive anche del comportamento futuro dello stesso programma.

Istruzione di ritorno mancante nella funzione di ritorno del valore

```
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;
}
```

Quando una funzione viene dichiarata per restituire un valore, deve farlo su ogni possibile percorso di codice attraverso di essa. Il comportamento non definito si verifica non appena il chiamante (che si aspetta un valore di ritorno) tenta di utilizzare il valore restituito [1] .

Si noti che il comportamento non definito si verifica *solo se* il chiamante tenta di utilizzare / accedere al valore dalla funzione. Per esempio,

```
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

La funzione `main()` è un'eccezione a questa regola in quanto è possibile che venga terminata senza un'istruzione return poiché in questo caso verrà automaticamente utilizzato un valore di ritorno ipotizzato pari a `0` [2] .

---

[1] (*ISO / IEC 9899: 201x* , 6.9.1 / 12)

> Se viene raggiunto il} che termina una funzione e il chiamante chiama il valore della chiamata della funzione, il comportamento non è definito.

[2] (*ISO / IEC 9899:* 201x, 5.1.2.2.3 / 1)

> raggiungendo il} che termina la funzione principale restituisce un valore di 0.

Overflow intero con segno

Per il paragrafo 6.5 / 5 di C99 e C11, la valutazione di un'espressione produce un comportamento non definito se il risultato non è un valore rappresentabile del tipo dell'espressione. Per i tipi aritmetici, si parla di *overflow* . L'aritmetica dei numeri interi senza segno non si sovrappone perché si applica il paragrafo 6.2.5 / 9, facendo sì che qualsiasi risultato senza segno che altrimenti sarebbe fuori intervallo venga ridotto a un valore di intervallo. Non v'è alcuna disposizione analoga per i tipi interi *firmati,* tuttavia; questi possono e superano, producendo un comportamento indefinito. Per esempio,

```
#include <limits.h>      /* to get INT_MAX */

int main(void) {
    int i = INT_MAX + 1; /* Overflow happens here */
    return 0;
}
```

La maggior parte delle istanze di questo tipo di comportamento non definito sono più difficili da riconoscere o prevedere. L'overflow può in linea di massima derivare da qualsiasi operazione di addizione, sottrazione o moltiplicazione su interi con segno (soggetti alle consuete conversioni aritmetiche) in cui non esistono limiti effettivi o una relazione tra gli operandi per prevenirla. Ad esempio, questa funzione:

```
int square(int x) {
    return x * x;  /* overflows for some values of x */
}
```

è ragionevole, e fa la cosa giusta per valori di argomenti sufficientemente piccoli, ma il suo comportamento non è definito per valori di argomenti più grandi. Non si può giudicare dalla sola funzione se i programmi che lo chiamano mostrano un comportamento indefinito come risultato. Dipende da quali argomenti gli passano.

D'altra parte, si consideri questo banale esempio di aritmetica di interi con segno sicuro overflow:

```
int zero(int x) {
    return x - x;  /* Cannot overflow */
}
```

La relazione tra gli operandi dell'operatore di sottrazione assicura che la sottrazione non trabocchi mai. Oppure considera questo esempio un po 'più pratico:

```
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 */
}
```

Fintanto che i contatori non hanno un overflow individuale, gli operandi della sottrazione finale saranno entrambi non negativi. Tutte le differenze tra due valori di questo tipo sono rappresentabili come `int` .

Uso di una variabile non inizializzata

```
int a;
printf("%d", a);
```

La variabile `a` è un `int` con durata di archiviazione automatica. Il codice di esempio di cui sopra sta tentando di stampare il valore di una variabile non inizializzata (`a` non è mai stato inizializzato). Le variabili automatiche non inizializzate hanno valori indeterminati; l'accesso a questi può portare a comportamenti non definiti.

Nota: le variabili con memoria locale statica o thread, comprese le variabili globali senza la parola chiave `static` , vengono inizializzate su zero o il loro valore inizializzato. Quindi il seguente è legale.

```
static int b;
printf("%d", b);
```

Un errore molto comune è quello di non inizializzare le variabili che fungono da contatori a 0. Si aggiungono valori a loro, ma poiché il valore iniziale è spazzatura, si invocherà il **comportamento indefinito** , come nella domanda La compilazione sul terminale emette un avviso del puntatore e simboli strani .

Esempio:

```
#include <stdio.h>

int main(void) {
    int i, counter;
    for(i = 0; i < 10; ++i)
        counter += i;
    printf("%d\n", counter);
    return 0;
}
```

Produzione:

```
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
```

Le regole di cui sopra sono applicabili anche per i puntatori. Ad esempio, i seguenti risultati nel comportamento non definito

```
int main(void)
{
    int *p;
    p++; // Trying to increment an uninitialized pointer.
}
```

Si noti che il codice precedente potrebbe non causare un errore o un errore di segmentazione, ma provare a dereferenziare questo puntatore in un secondo momento causerebbe un comportamento indefinito.

Dereferenziare un puntatore alla variabile oltre la sua durata

```
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;
}
```

Alcuni compilatori lo segnalano utilmente. Ad esempio, `gcc` avvisa con:

```
warning: function returns address of local variable [-Wreturn-local-addr]
```

e `clang` avverte con:

```
warning: address of stack memory associated with local variable 'baz' returned
[-Wreturn-stack-address]
```

per il codice sopra. Ma i compilatori potrebbero non essere in grado di aiutare nel codice complesso.

(1) Il ritorno del riferimento alla variabile dichiarata `static` è un comportamento definito, in quanto la variabile non viene distrutta dopo aver abbandonato l'ambito corrente.

(2) Secondo ISO / IEC 9899: 2011 6.2.4 §2, "Il valore di un puntatore diventa indeterminato quando l'oggetto a cui punta raggiunge la fine della sua vita."

(3) Il dereferenziamento del puntatore restituito dalla funzione `foo` è un comportamento indefinito in quanto la memoria a cui fa riferimento detiene un valore indeterminato.

Divisione per zero

```
int x = 0;
int y = 5 / x;  /* integer division */
```

o

```
double x = 0.0;
double y = 5.0 / x;  /* floating point division */
```

o

```
int x = 0;
int y = 5 % x;  /* modulo operation */
```

Per la seconda riga di ogni esempio, in cui il valore del secondo operando (x) è zero, il comportamento non è definito.

Si noti che la maggior parte delle implementazioni della matematica in virgola mobile seguiranno uno standard (ad es. IEEE 754), nel qual caso le operazioni come lo split-per-zero avranno risultati coerenti (ad esempio `INFINITY`) anche se lo standard C dice che l'operazione non è definita.

Accesso alla memoria oltre il blocco assegnato

Un puntatore a un pezzo di memoria contenente `n` elementi può essere dereferenziato solo se si trova nella `memory` dell'intervallo e nella `memory + (n - 1)` . Il dereferenziamento di un puntatore al di fuori di tale intervallo comporta un comportamento indefinito. Ad esempio, considera il seguente codice:

```
int array[3];
int *beyond_array = array + 3;
*beyond_array = 0; /* Accesses memory that has not been allocated. */
```

La terza riga accede al quarto elemento in un array che ha una lunghezza di soli 3 elementi, il che porta a un comportamento non definito. Analogamente, anche il comportamento della seconda riga nel seguente frammento di codice non è ben definito:

```
int array[3];
array[3] = 0;
```

Si noti che il puntamento oltre l'ultimo elemento di un array non è un comportamento non definito (`beyond_array = array + 3` è ben definito qui), ma il dereferenziamento è (`*beyond_array` è un comportamento non definito). Questa regola vale anche per la memoria allocata dinamicamente (come i buffer creati tramite `malloc`).

Copia della memoria sovrapposta

Un'ampia varietà di funzioni di libreria standard hanno tra i loro effetti la copia di sequenze di byte da una regione di memoria a un'altra. La maggior parte di queste funzioni ha un comportamento non definito quando le regioni di origine e di destinazione si sovrappongono.

Ad esempio, questo ...

```
#include <string.h> /* for memcpy() */

char str[19] = "This is an example";
memcpy(str + 7, str, 10);
```

... tenta di copiare 10 byte in cui le aree di memoria di origine e destinazione si sovrappongono di tre byte. Per visualizzare:

```
               overlapping area
               |
               _ _
              |   |
              v   v
T h i s   i s   a n   e x a m p l e \0
^             ^
|             |
|             destination
|
source
```

A causa della sovrapposizione, il comportamento risultante non è definito.

Tra le funzioni di libreria standard con una limitazione di questo tipo sono `memcpy()` , `strcpy()` , `strcat()` , `sprintf()` e `sscanf()` . Lo standard dice di queste e di molte altre funzioni:

> Se la copia avviene tra oggetti che si sovrappongono, il comportamento non è definito.

La funzione `memmove()` è l'eccezione principale a questa regola. La sua definizione specifica che la funzione si comporta come se i dati di origine fossero stati prima copiati in un buffer temporaneo e quindi scritti nell'indirizzo di destinazione. Non ci sono eccezioni per la sovrapposizione di regioni di origine e destinazione, né alcuna necessità per una, quindi `memmove()` ha un comportamento ben definito in questi casi.

La distinzione riflette un'efficienza *vs.* compromesso di generalità. La copia di tali funzioni si verifica in genere tra regioni disgiunte della memoria e spesso è possibile sapere al momento dello sviluppo se una determinata istanza di copia della memoria si troverà in quella categoria. Supponendo che la non sovrapposizione offra implementazioni relativamente più efficienti che non producono in modo affidabile risultati corretti quando l'ipotesi non regge. La maggior parte delle funzioni della libreria C è permessa le implementazioni più efficienti e `memmove()` riempie gli spazi vuoti, servendo i casi in cui l'origine e la destinazione possono o si sovrappongono. Tuttavia, per produrre l'effetto corretto in tutti i casi, è necessario eseguire ulteriori test e / o impiegare un'implementazione relativamente meno efficiente.

Leggere un oggetto non inizializzato che non è supportato dalla memoria

C11

La lettura di un oggetto causerà un comportamento indefinito, se l'oggetto è [1] :

- inizializzata
- definito con durata di archiviazione automatica
- il suo indirizzo non è mai stato preso

La variabile a nell'esempio seguente soddisfa tutte queste condizioni:

```
void Function( void )
{
    int a;
    int b = a;
}
```

---

[1] (Citato da: ISO: IEC 9899: 201X 6.3.2.1 Lvalues, matrici e designatori di funzioni 2)
Se il lvalue designa un oggetto di durata di archiviazione automatica che avrebbe potuto essere dichiarato con la classe di archiviazione del registro (non ha mai avuto l'indirizzo preso), e quell'oggetto non è inizializzato (non dichiarato con un inizializzatore e non è stato eseguito alcun incarico prima dell'uso), il comportamento non è definito.

Gara di dati

C11

C11 ha introdotto il supporto per più thread di esecuzione, che offre la possibilità di gare di dati. Un programma contiene una corsa di dati se si accede a un oggetto in esso [1] da due thread diversi, in cui almeno uno degli accessi è non atomico, almeno uno modifica l'oggetto e la semantica del programma non riesce a garantire che i due accessi non si sovrappongano temporalmente. [2] Si noti bene che la concomitanza effettiva degli accessi coinvolti non è una condizione per una corsa di dati; le gare di dati coprono una più ampia classe di problemi derivanti da (ammesse) incoerenze nelle diverse visualizzazioni di memoria dei thread.

Considera questo esempio:

```
#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 );
}
```

I principali thread chiama `thrd_create` per iniziare una nuova funzione di thread in esecuzione `Function` . Il secondo thread modifica `a` , e il thread principale legge `a` . Nessuno di questi accessi è atomico, ei due thread non fanno nulla né individualmente né congiuntamente per garantire che non si sovrappongano, quindi c'è una corsa ai dati.

Tra i modi in cui questo programma potrebbe evitare la corsa dei dati sono

- il thread principale potrebbe eseguire la lettura di `a` prima di iniziare l'altro thread;
- il thread principale potrebbe eseguire la sua lettura di `a` dopo assicurandosi tramite `thrd_join` che l'altro è terminato;
- i thread potevano sincronizzare i loro accessi tramite un mutex, ognuno bloccando quel mutex prima di accedere `a` e sbloccarlo in seguito.

Come dimostra l'opzione mutex, evitare una corsa di dati non richiede di garantire uno specifico ordine di operazioni, come ad esempio il thread figlio che modifica `a` prima che il thread principale lo legga; è sufficiente (per evitare una corsa di dati) assicurarsi che per una determinata esecuzione, un accesso avverrà prima dell'altro.

---

[1] Modifica o lettura di un oggetto.

[2] (Citato da ISO: IEC 9889: 201x, sezione 5.1.2.4 "Esecuzioni multi-threaded e corse di dati") L'esecuzione di un programma contiene una corsa di dati se contiene due azioni in conflitto in thread diversi, almeno uno dei quali non è atomico, e nessuno dei due avviene prima dell'altro. Qualsiasi razza di dati di questo genere ha un comportamento indefinito.

Leggi il valore del puntatore che è stato liberato

Anche solo la **lettura** del valore di un puntatore che è stato liberato (cioè senza cercare di dereferenziare il puntatore) è un comportamento indefinito (UB), ad es.

```
char *p = malloc(5);
free(p);
if (p == NULL) /* NOTE: even without dereferencing, this may have UB */
{

}
```

Citando **ISO / IEC 9899: 2011** , sezione 6.2.4 §2:

> [...] Il valore di un puntatore diventa indeterminato quando l'oggetto a cui punta (o appena passato) raggiunge la fine della sua vita.

L'uso della memoria indeterminata per qualsiasi cosa, incluso il confronto apparentemente innocuo o l'aritmetica, può avere un comportamento indefinito se il valore può essere una rappresentazione trap per il tipo.

Modifica la stringa letterale

In questo esempio di codice, il puntatore char `p` viene inizializzato all'indirizzo di una stringa letterale. Il tentativo di modificare la stringa letterale ha un comportamento indefinito.

```
char *p = "hello world";
p[0] = 'H'; // Undefined behavior
```

Tuttavia, modificare un array mutevole di `char` direttamente o tramite un puntatore non è naturalmente un comportamento indefinito, anche se il suo inizializzatore è una stringa letterale. Quanto segue va bene:

```
char a[] = "hello, world";
char *p = a;

a[0] = 'H';
p[7] = 'W';
```

Questo perché la stringa letterale viene effettivamente copiata nell'array ogni volta che l'array viene inizializzato (una volta per le variabili con durata statica, ogni volta che la matrice viene creata per le variabili con durata automatica o di thread - le variabili con durata allocata non vengono inizializzate) e va bene per modificare i contenuti dell'array.

Liberare memoria due volte

Liberare la memoria due volte è un comportamento indefinito, ad es

```
int * x = malloc(sizeof(int));
*x = 9;
free(x);
free(x);
```

Citazione dallo standard (7.20.3.2. La funzione libera di C99):

> Altrimenti, se l'argomento non corrisponde a un puntatore precedentemente restituito dalla funzione calloc, malloc o realloc, o se lo spazio è stato deallocato da una chiamata a free o realloc, il comportamento non è definito.

Utilizzo di un identificatore di formato errato in printf

L'utilizzo di un identificatore di formato non corretto nel primo argomento di `printf` richiama il comportamento non definito. Ad esempio, il codice seguente richiama il comportamento non definito:

```
long z = 'B';
printf("%c\n", z);
```

Ecco un altro esempio

```
printf("%f\n",0);
```

La linea superiore del codice è un comportamento non definito. `%f` aspetta il doppio. Tuttavia 0 è di tipo `int` .

Nota che il tuo compilatore di solito può aiutarti a evitare casi come questi, se attivi i flag appropriati durante la compilazione (`-Wformat` in `clang` e `gcc`). Dall'ultimo esempio:

```
warning: format specifies type 'double' but the argument has type
      'int' [-Wformat]
    printf("%f\n",0);
             ~~    ^
             %d
```

La conversione tra i tipi di puntatore produce risultati allineati in modo errato

Quanto segue *potrebbe* avere un comportamento indefinito a causa di un allineamento errato del puntatore:

```
char *memory_block = calloc(sizeof(uint32_t) + 1, 1);
uint32_t *intptr = (uint32_t*)(memory_block + 1);  /* possible undefined behavior */
uint32_t mvalue = *intptr;
```

Il comportamento non definito si verifica quando il puntatore viene convertito. Secondo C11, se una *conversione tra due tipi di puntatore produce un risultato allineato in modo errato (6.3.2.3), il comportamento non è definito* . Qui un `uint32_t` potrebbe richiedere l'allineamento di 2 o 4, ad esempio.

`calloc` d'altra parte è necessario per restituire un puntatore che sia adeguatamente allineato per qualsiasi tipo di oggetto; quindi `memory_block` è correttamente allineato per contenere un `uint32_t` nella sua parte iniziale. Quindi, su un sistema in cui `uint32_t` ha richiesto l'allineamento di 2 o 4, `memory_block + 1` sarà un indirizzo *dispari* e quindi non correttamente allineato.

Osservare che lo standard C richiede che l'operazione di cast sia già definita. Questo è imposto perché su piattaforme in cui gli indirizzi sono segmentati, l'indirizzo di byte `memory_block + 1` potrebbe non avere nemmeno una rappresentazione corretta come puntatore intero.

Lanciare `char *` a puntatori ad altri tipi senza alcuna preoccupazione per i requisiti di allineamento viene talvolta erroneamente utilizzato per la decodifica di strutture compresse come intestazioni di file o pacchetti di rete.

È possibile evitare il comportamento non definito derivante dalla conversione del puntatore disallineata utilizzando `memcpy` :

```
memcpy(&mvalue, memory_block + 1, sizeof mvalue);
```

Qui non avviene alcuna conversione del puntatore a `uint32_t*` e i byte vengono copiati uno per uno.

Questa operazione di copia per il nostro esempio porta solo al valore valido del valore `mvalue` perché:

- Abbiamo usato `calloc` , quindi i byte sono inizializzati correttamente. Nel nostro caso tutti i byte hanno valore `0` , ma qualsiasi altra inizializzazione corretta dovrebbe fare.
- `uint32_t` è un tipo di larghezza esatta e non ha bit di riempimento
- Qualsiasi modello di bit arbitrario è una rappresentazione valida per qualsiasi tipo senza segno.

Aggiunta o sottrazione del puntatore non propriamente limitato

Il seguente codice ha un comportamento non definito:

```
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 */
```

Secondo C11, se l'addizione o la sottrazione di un puntatore in, o appena oltre, un oggetto array e un tipo intero produce un risultato che non punta, o appena oltre, lo stesso oggetto array, il comportamento non è definito (6.5.6).

Inoltre è naturalmente un comportamento indefinito a *dereferenziare* un puntatore che punta appena oltre l'array:

```
char buffer[6] = "hello";
char *ptr3 = buffer + 6;  /* OK, pointing to just beyond */
char value = *ptr3;       /* undefined behavior */
```

Modifica di una variabile const mediante un puntatore

```
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;
}
```

Citando *ISO / IEC 9899: 201x* , sezione 6.7.3 §2:

> Se viene effettuato un tentativo di modificare un oggetto definito con un tipo qualificato const tramite l'utilizzo di un valore lvalue con tipo non const-qualificato, il comportamento non è definito. [...]

---

(1) In GCC questo può lanciare il seguente avvertimento: `warning: assignment discards 'const' qualifier from pointer target type [-Wdiscarded-qualifiers]`

Passare un puntatore nullo alla conversione di printf% s

La conversione `%s` di `printf` indica che l'argomento corrispondente è *un puntatore all'elemento iniziale di una matrice di tipo carattere* . Un puntatore nullo non punta all'elemento iniziale di qualsiasi matrice di tipo di carattere e pertanto il comportamento di quanto segue non è definito:

```
char *foo = NULL;
printf("%s", foo); /* undefined behavior */
```

Tuttavia, il comportamento non definito non sempre significa che il programma si arresta in modo anomalo: alcuni sistemi adottano misure per evitare l'arresto anomalo che normalmente si verifica quando un puntatore nullo viene dereferenziato. Ad esempio, Glibc è noto per la stampa

```
(null)
```

per il codice sopra. Tuttavia, aggiungi (solo) una nuova riga alla stringa di formato e otterrai un arresto anomalo:

```
char *foo = 0;
printf("%s\n", foo); /* undefined behavior */
```

In questo caso, succede perché GCC ha un'ottimizzazione che trasforma `printf("%s\n", argument);` in una chiamata a `puts` con `puts(argument)` e `puts` in Glibc non gestisce i puntatori nulli. Tutto questo comportamento è conforme allo standard.

Si noti che il *puntatore nullo* è diverso da una *stringa vuota* . Quindi, il seguente è valido e non ha un comportamento indefinito. Stamperà solo una *nuova riga* :

```
char *foo = "";
printf("%s\n", foo);
```

Collegamento incoerente di identificatori

```
extern int var;
static int var; /* Undefined behaviour */
```

C11, §6.2.2, 7 dice:

> Se, all'interno di un'unità di traduzione, lo stesso identificatore appare con il collegamento sia interno che esterno, il comportamento è indefinito.

Si noti che se una dichiarazione preventiva di un identificatore è visibile, avrà il collegamento della dichiarazione precedente. *C11, §6.2.2, 4* consente:

> Per un identificatore dichiarato con la classe di memoria speci fi catore esterno in uno scope in cui è visibile una dichiarazione preliminare di tale identificatore, 31) se la dichiarazione precedente specifica il collegamento interno o esterno, il collegamento dell'identi fi catore alla dichiarazione successiva è lo stesso di il collegamento speci fi cato alla dichiarazione precedente. Se nessuna dichiarazione precedente è visibile, o se la dichiarazione precedente non specifica alcun collegamento, allora l'identificatore ha un collegamento esterno.

```
/* 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;
```

Usando fflush su un flusso di input

Gli standard POSIX e C affermano esplicitamente che l'uso di `fflush` su un flusso di input è un comportamento indefinito. Il `fflush` è definito solo per i flussi di output.

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

```
int main()
{
    int i;
    char input[4096];

    scanf("%i", &i);
    fflush(stdin); // <-- undefined behavior
    gets(input);

    return 0;
}
```

Non esiste un modo standard per scartare i caratteri non letti da un flusso di input. D'altra parte, alcune implementazioni usano `fflush` per cancellare il buffer di `stdin` . Microsoft definisce il comportamento di `fflush` su un flusso di input: se il flusso è aperto per l'input, `fflush` cancella il contenuto del buffer. Secondo POSIX.1-2008, il comportamento di `fflush` non è definito a meno che il file di input non sia ricercabile.

Vedi Usare `fflush(stdin)` per molti più dettagli.

Spostamento di bit usando conteggi negativi o oltre la larghezza del tipo

Se il valore del *conteggio dei turni* è un **valore negativo,** entrambe le operazioni di *spostamento a sinistra* e di *spostamento a destra* non sono definite [1] :

```
int x = 5 << -3; /* undefined */
int x = 5 >> -3; /* undefined */
```

Se lo *spostamento a sinistra* viene eseguito su un **valore negativo** , non è definito:

```
int x = -5 << 3; /* undefined */
```

Se lo *spostamento a sinistra* viene eseguito su un **valore positivo** e il risultato del valore matematico **non** è rappresentabile nel tipo, non è definito [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;
```

Si noti che il *passaggio a destra* su un **valore negativo** (.eg `-5 >> 3`) *non* è indefinito ma definito *dall'implementazione* .

---

[1] Citando *ISO / IEC 9899: 201x* , sezione 6.5.7:

> Se il valore dell'operando di destra è negativo o è maggiore o uguale alla larghezza dell'operando di sinistra promosso, il comportamento non è definito.

Modifica della stringa restituita dalle funzioni getenv, strerror e setlocale

La modifica delle stringhe restituite dalle funzioni standard `getenv()` , `strerror()` e `setlocale()` non è definita. Pertanto, le implementazioni potrebbero utilizzare l'archiviazione statica per queste stringhe.

La funzione getenv (), C11, §7.22.4.7, 4 , dice:

> La funzione getenv restituisce un puntatore a una stringa associata al membro della lista corrispondente. La stringa puntata su non deve essere modificata dal programma, ma può essere sovrascritta da una chiamata successiva alla funzione getenv.

La funzione strerror (), C11, §7.23.6.3, 4 dice:

> La funzione strerror restituisce un puntatore alla stringa, il cui contenuto è localespeci fi c. L'array puntato su non deve essere modificato dal programma, ma può essere sovrascritto da una chiamata successiva alla funzione strerror.

La funzione setlocale (), C11, §7.11.1.1, 8 dice:

> Il puntatore alla stringa restituito dalla funzione setlocale è tale che una chiamata successiva con quel valore stringa e la relativa categoria ripristinerà quella parte delle impostazioni locali del programma. La stringa puntata su non deve essere modificata dal programma, ma può essere sovrascritta da una chiamata successiva alla funzione setlocale.

Allo stesso modo la funzione `localeconv()` restituisce un puntatore a `struct lconv` che non deve essere modificato.

La funzione localeconv (), C11, §7.11.2.1, 8 dice:

> La funzione localeconv restituisce un puntatore all'oggetto compilato. La struttura indicata dal valore di ritorno non deve essere modificata dal programma, ma può essere sovrascritta da una chiamata successiva alla funzione localeconv.

Di ritorno da una funzione dichiarata con l'identificatore di funzione `_Noreturn` o` noreturn`

C11

Lo specificatore della funzione `_Noreturn` stato introdotto in C11. L'intestazione `<stdnoreturn.h>` fornisce una macro `noreturn` che si espande in `_Noreturn` . Quindi usare `_Noreturn` o `noreturn` da `<stdnoreturn.h>` va bene ed è equivalente.

Una funzione dichiarata con `_Noreturn` (o `noreturn`) non può tornare al suo chiamante. Se tale funzione *non* tornare al chiamante, il comportamento è indefinito.

Nell'esempio seguente, `func()` è dichiarato con specificatore `noreturn` ma ritorna al suo chiamante.

```
#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` e `clang` producono avvisi per il programma di cui sopra:

```
$ 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]
}
^
```

Un esempio di `noreturn` che ha un comportamento ben definito:

```
#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;
}
```

Leggi Comportamento non definito online: https://riptutorial.com/it/c/topic/364/comportamento-non-definito

Capitolo 17: Conversioni implicite ed esplicite

Sintassi

- Conversione esplicita (aka "Casting"): (tipo) espressione

Osservazioni

" *Conversione esplicita* " è anche comunemente chiamato "casting".

Examples

Conversioni intere in chiamate di funzione

Dato che la funzione ha un prototipo corretto, gli interi vengono ampliati per le chiamate alle funzioni secondo le regole della conversione dei numeri interi, C11 6.3.1.3.

> **6.3.1.3 Numeri interi firmati e non firmati**
> Quando un valore con tipo intero viene convertito in un altro tipo intero diverso da _Bool, se il valore può essere rappresentato dal nuovo tipo, non viene modificato.
>
> Altrimenti, se il nuovo tipo non è firmato, il valore viene convertito aggiungendo o sottraendo ripetutamente un valore superiore al valore massimo che può essere rappresentato nel nuovo tipo finché il valore non si trova nell'intervallo del nuovo tipo.
>
> Altrimenti, il nuovo tipo è firmato e il valore non può essere rappresentato in esso; o il risultato è definito dall'implementazione o viene generato un segnale definito dall'implementazione.

Di solito non si deve troncare un ampio tipo firmato con un tipo firmato più stretto, perché ovviamente i valori non possono essere adattati e non vi è alcun chiaro significato che ciò dovrebbe avere. Lo standard C sopra citato definisce questi casi come "definiti dall'implementazione", cioè non sono portabili.

L'esempio seguente suppone che `int` sia a 32 bit di larghezza.

```
#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;
}
```

Conversioni puntatore nelle chiamate di funzione

Le conversioni del puntatore a `void*` sono implicite, ma qualsiasi altra conversione del puntatore deve essere esplicita. Mentre il compilatore consente una conversione esplicita da qualsiasi tipo di

puntatore a dati a qualsiasi altro tipo di puntatore a dati, l'accesso a un oggetto tramite un puntatore errato è errato e porta a un comportamento indefinito. L'unico caso in cui questi sono consentiti sono se i tipi sono compatibili o se il puntatore con cui stai guardando l'oggetto è un tipo di carattere.

```
#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;
}
```

Leggi Conversioni implicite ed esplicite online: https://riptutorial.com/it/c/topic/2529/conversioni-implicite-ed-esplicite

Capitolo 18: Crea e includi i file di intestazione

introduzione

Nella moderna C, i file di intestazione sono strumenti cruciali che devono essere progettati e utilizzati correttamente. Consentono al compilatore di eseguire il controllo incrociato delle parti compilate in modo indipendente di un programma.

Le intestazioni dichiarano tipi, funzioni, macro ecc. Che sono necessari ai consumatori di una serie di servizi. Tutto il codice che utilizza una di queste strutture include l'intestazione. Tutto il codice che definisce queste strutture include l'intestazione. Ciò consente al compilatore di verificare che gli usi e le definizioni corrispondano.

Examples

introduzione

Ci sono un certo numero di linee guida da seguire quando si creano e si usano i file di intestazione in un progetto C:

- Idemopotence

  Se un file di intestazione è incluso più volte in un'unità di traduzione (TU), non dovrebbe interrompere le build.

- Self-contenimento

  Se hai bisogno delle strutture dichiarate in un file di intestazione, non dovresti includere esplicitamente nessun altro header.

- minimalità

  Non dovresti essere in grado di rimuovere alcuna informazione da un'intestazione senza causare fallimenti di build.

- Includi cosa usi (IWYU)

  Di maggiore preoccupazione per C ++ rispetto a C, ma comunque importante anche in C. Se il codice in una TU (chiamiamolo `code.c`) utilizza direttamente le caratteristiche dichiarate da un'intestazione (chiamatelo `"headerA.h"`), quindi `code.c` dovrebbe `#include "headerA.h"` direttamente, anche se la TU include un'altra intestazione (chiamiamola `"headerB.h"`) che al momento include `"headerA.h"` .

Occasionalmente, ci potrebbero essere buoni motivi per rompere una o più di queste linee guida, ma entrambi dovresti essere consapevole del fatto che stai infrangendo la regola ed essere

consapevole delle conseguenze di ciò prima di romperlo.

idempotence

Se un particolare file di intestazione è incluso più di una volta in un'unità di traduzione (TU), non dovrebbero esserci problemi di compilazione. Questo è chiamato 'idempotence'; le tue intestazioni dovrebbero essere idempotenti. Pensa a quanto sarebbe difficile la vita se dovessi assicurarti che `#include <stdio.h>` stato incluso solo una volta.

Esistono due modi per ottenere l'idempotence: header guards e `#pragma once` directive.

Guardie di testa

Le protezioni dell'intestazione sono semplici e affidabili e conformi allo standard C. Le prime righe senza commento in un file di intestazione dovrebbero essere del formato:

```
#ifndef UNIQUE_ID_FOR_HEADER
#define UNIQUE_ID_FOR_HEADER
```

L'ultima riga di non commento deve essere `#endif` , facoltativamente con un commento dopo di esso:

```
#endif /* UNIQUE_ID_FOR_HEADER */
```

Tutto il codice operativo, comprese altre `#include` direttive, dovrebbe essere tra queste linee.

Ogni nome deve essere unico. Spesso, viene utilizzato uno schema di nomi come `HEADER_H_INCLUDED` . Alcuni codici meno `#ifndef BUFSIZ` utilizzano un simbolo definito come intestazione (ad esempio `#ifndef BUFSIZ` in `<stdio.h>`), ma non è affidabile come un nome univoco.

Un'opzione consisterebbe nell'utilizzare un hash MD5 (o altro) generato per il nome della guardia dell'intestazione. Evitare di emulare gli schemi utilizzati dalle intestazioni di sistema che utilizzano frequentemente i nomi riservati all'implementazione - i nomi iniziano con un trattino basso seguito da un altro carattere di sottolineatura o da una lettera maiuscola.

Il `#pragma once` direttiva

In alternativa, alcuni compilatori supportano la direttiva `#pragma once` che ha lo stesso effetto delle tre linee mostrate per le guardie di intestazione.

```
#pragma once
```

I compilatori che supportano `#pragma once` includono MS Visual Studio e GCC e Clang. Tuttavia, se la portabilità è un problema, è preferibile utilizzare le protezioni intestazione o utilizzare entrambe. I compilatori moderni (quelli che supportano C89 o successivi) devono ignorare, senza commenti, pragma che non riconoscono ("Qualsiasi pragma di questo tipo che non è riconosciuto

dall'implementazione viene ignorato") ma le vecchie versioni di GCC non erano così indulgenti.

Self-contenimento

Le intestazioni moderne dovrebbero essere autarchiche, il che significa che un programma che deve utilizzare le funzionalità definite da `header.h` può includere quell'intestazione (`#include "header.h"`) e non preoccuparsi se altre intestazioni devono essere incluse per prime.

Raccomandazione: i file di intestazione dovrebbero essere autonomi.

Regole storiche

Storicamente, questo è stato un argomento moderatamente controverso.

Nel corso di un altro millennio, gli standard AT & T Indian Hill C e gli standard di codifica hanno dichiarato:

> I file di intestazione non devono essere nidificati. Il prologo per un file di intestazione dovrebbe, quindi, descrivere quali altre intestazioni devono essere `#include` d perché l'intestazione sia funzionale. In casi estremi, in cui è necessario includere un numero elevato di file di intestazione in diversi file di origine, è accettabile inserire tutti i `#include` s comuni in un file di inclusione.

Questa è l'antitesi del self-contenimento.

Regole moderne

Tuttavia, da allora, l'opinione ha teso nella direzione opposta. Se un file sorgente deve utilizzare le strutture dichiarate da un'intestazione `header.h` , il programmatore dovrebbe essere in grado di scrivere:

```
#include "header.h"
```

e (soggetto solo ad avere i percorsi di ricerca corretti impostati sulla riga di comando), tutti gli header necessari per i prerequisiti saranno inclusi da `header.h` senza bisogno di ulteriori intestazioni aggiunte al file sorgente.

Ciò fornisce una migliore modularità per il codice sorgente. Protegge inoltre la fonte dall'enigma "indovina perché questa intestazione è stata aggiunta" che si verifica dopo che il codice è stato modificato e violato per un decennio o due.

Gli standard di codifica Goddard Space Flight Center (GSFC) della NASA per C sono uno degli standard più moderni, ma ora è un po 'difficile da rintracciare. Dichiara che le intestazioni dovrebbero essere autonome. Fornisce anche un modo semplice per garantire che le intestazioni siano autonome: il file di implementazione per l'intestazione dovrebbe includere l'intestazione

come prima intestazione. Se non è autonomo, quel codice non verrà compilato.

La motivazione fornita da GSFC include:

> §2.1.1 L'intestazione include la logica
>
> Questo standard richiede che l'intestazione di un'unità contenga le istruzioni `#include` per tutte le altre intestazioni richieste dall'intestazione dell'unità. Inserire `#include` per l'intestazione dell'unità prima nel corpo dell'unità consente al compilatore di verificare che l'intestazione contenga tutte le istruzioni `#include` richieste.
>
> Un disegno alternativo, non consentito da questo standard, non consente di `#include` istruzioni `#include` nelle intestazioni; tutti gli `#includes` sono fatti nei file del corpo. I file di intestazione dell'unità devono quindi contenere istruzioni #ifdef che controllano che le intestazioni richieste siano incluse nell'ordine corretto.
>
> Un vantaggio del design alternativo è che l'elenco `#include` nel file body è esattamente l'elenco delle dipendenze necessario in un makefile e questo elenco viene controllato dal compilatore. Con la progettazione standard, è necessario utilizzare uno strumento per generare l'elenco delle dipendenze. Tuttavia, tutti gli ambienti di sviluppo consigliati dalla filiale forniscono uno strumento del genere.
>
> Uno svantaggio principale del design alternativo è che se l'elenco di intestazioni richiesto di un'unità cambia, ogni file che utilizza quell'unità deve essere modificato per aggiornare l'elenco di istruzioni `#include` . Inoltre, l'elenco di intestazioni richiesto per un'unità di libreria del compilatore può essere diverso su target diversi.
>
> Un altro svantaggio della progettazione alternativa è che i file di intestazione della libreria del compilatore e altri file di terze parti devono essere modificati per aggiungere le istruzioni `#ifdef` richieste.

Quindi, l'auto-contenimento significa che:

- Se un'intestazione `header.h` bisogno di una nuova intestazione nidificata `extra.h` , non è necessario controllare ogni file sorgente che utilizza `header.h` per vedere se è necessario aggiungere `extra.h` .
- Se un'intestazione `header.h` non deve più includere un'intestazione specifica `notneeded.h` , non è necessario controllare ogni file sorgente che utilizza `header.h` per vedere se è possibile rimuovere `notneeded.h` modo sicuro (ma vedere Includi ciò che si usa .
- Non è necessario stabilire la sequenza corretta per includere le intestazioni pre-requisito (che richiede un ordinamento topologico per eseguire correttamente il lavoro).

Controllo di auto-contenimento

Vedi Collegamento a una libreria statica per uno script `chkhdr` che può essere utilizzato per testare l'idempotenza e l'autocontenimento di un file di intestazione.

minimalità

Le intestazioni sono un meccanismo di verifica della coerenza fondamentale, ma dovrebbero essere il più possibile ridotte. In particolare, ciò significa che un'intestazione non dovrebbe includere altre intestazioni solo perché il file di implementazione avrà bisogno delle altre intestazioni. Un'intestazione deve contenere solo le intestazioni necessarie per un consumatore dei servizi descritti.

Ad esempio, un'intestazione di progetto non dovrebbe includere `<stdio.h>` meno che una delle interfacce di funzione non utilizzi il tipo `FILE *` (o uno degli altri tipi definiti esclusivamente in `<stdio.h>`). Se un'interfaccia utilizza `size_t` , l'intestazione più piccola che è sufficiente è `<stddef.h>` . Ovviamente, se è inclusa un'altra intestazione che definisce `size_t` , non è necessario includere anche `<stddef.h>` .

Se le intestazioni sono minime, mantiene anche il tempo di compilazione al minimo.

È possibile ideare intestazioni il cui unico scopo è includere molte altre intestazioni. Queste raramente risultano essere una buona idea a lungo termine perché pochi file sorgente avranno effettivamente bisogno di tutte le funzionalità descritte da tutte le intestazioni. Ad esempio, potrebbe essere ideato un `<standard-ch>` che includa tutti gli header C standard - con attenzione dato che alcune intestazioni non sono sempre presenti. Tuttavia, pochissimi programmi utilizzano effettivamente le funzionalità di `<locale.h>` o `<tgmath.h>` .

- Vedi anche Come collegare più file di implementazione in C?

Includi cosa usi (IWYU)

Il progetto Includi cosa utilizzi di Google o IWYU garantisce che i file di origine includano tutte le intestazioni utilizzate nel codice.

Supponiamo che un file sorgente `source.c` includa un'intestazione `arbitrary.h` che a sua volta include casualmente `freeloader.h` , ma il file sorgente utilizza anche esplicitamente e in modo indipendente le strutture da `freeloader.h` . Tutto va bene per cominciare. Quindi un giorno `arbitrary.h` viene cambiato in modo che i suoi clienti non `freeloader.h` più bisogno delle funzionalità di `freeloader.h` . All'improvviso `source.c` interrompe la compilazione perché non soddisfa i criteri IWYU. Poiché il codice in `source.c` utilizzava esplicitamente le funzionalità di `freeloader.h` , avrebbe dovuto includere ciò che utilizzava - avrebbe dovuto esserci un esplicito `#include "freeloader.h"` nella sorgente. (L' idempionalità avrebbe assicurato che non c'era un problema.)

La filosofia IWYU massimizza la probabilità che il codice continui a compilare anche con modifiche ragionevoli apportate alle interfacce. Chiaramente, se il tuo codice chiama una funzione che viene successivamente rimossa dall'interfaccia pubblicata, nessuna quantità di preparazione può impedire che le modifiche diventino necessarie. Questo è il motivo per cui le modifiche alle API vengono evitate quando possibile e perché ci sono cicli di deprecazione su più versioni, ecc.

Questo è un problema particolare in C ++ perché le intestazioni standard possono essere incluse tra loro. Il file di origine `file.cpp` potrebbe includere un'intestazione `header1.h` che su una piattaforma include un'altra intestazione `header2.h` . `file.cpp` potrebbe risultare utilizzare anche le funzionalità di `header2.h` . Inizialmente questo non sarebbe un problema: il codice verrebbe

compilato perché `header1.h` include `header2.h` . Su un'altra piattaforma, o un aggiornamento della piattaforma corrente, `header1.h` potrebbe essere rivisto in modo che non includa più `header2.h` , e quindi `file.cpp` smetterebbe di compilare come risultato.

IWYU individuerebbe il problema e raccomanderebbe che `header2.h` fosse incluso direttamente in `file.cpp` . Ciò garantirebbe che continui a compilare. Considerazioni analoghe si applicano anche al codice C.

Notazione e Miscellanea

Lo standard C dice che c'è ben poca differenza tra le `#include <header.h>` e `#include "header.h"` .

> [`#include <header.h>`] ricerca una sequenza di luoghi definiti dall'implementazione per un'intestazione identificata univocamente dalla sequenza specificata tra i delimitatori `<` e `>` e causa la sostituzione di tale direttiva con l'intero contenuto dell'intestazione. Come vengono specificati i posti o l'intestazione identificata è definita dall'implementazione.
>
> [`#include "header.h"`] causa la sostituzione di quella direttiva con l'intero contenuto del file sorgente identificato dalla sequenza specificata tra i delimitatori `"…"` . Il file di origine denominato viene cercato in un modo definito dall'implementazione. Se questa ricerca non è supportata, o se la ricerca fallisce, la direttiva viene rielaborata come se leggesse [`#include <header.h>`] ...

Pertanto, la forma con doppia citazione può apparire in più punti rispetto alla forma con parentesi angolare. Lo standard specifica per esempio che le intestazioni standard dovrebbero essere incluse tra parentesi angolari, anche se la compilazione funziona se si utilizzano invece le virgolette doppie. Allo stesso modo, standard come POSIX usano il formato parentesi angolare - e dovresti farlo anche tu. Prenota intestazioni a doppia quotazione per intestazioni definite dal progetto. Per le intestazioni definite esternamente (comprese le intestazioni di altri progetti su cui si basa il progetto), la notazione a parentesi angolare è più appropriata.

Nota che dovrebbe esserci uno spazio tra `#include` e l'intestazione, anche se i compilatori non accetteranno spazio. Gli spazi sono economici.

Un numero di progetti utilizza una notazione come:

```
#include <openssl/ssl.h>
#include <sys/stat.h>
#include <linux/kernel.h>
```

Dovresti considerare se usare quel controllo del namespace nel tuo progetto (è probabilmente una buona idea). Dovresti evitare i nomi usati dai progetti esistenti (in particolare, sia `sys` che `linux` sarebbero scelte sbagliate).

Se lo usi, il tuo codice dovrebbe essere attento e coerente nell'uso della notazione.

Non usare la notazione `#include "../include/header.h"` .

I file di intestazione dovrebbero raramente definire le variabili. Sebbene manterrai le variabili globali al minimo, se hai bisogno di una variabile globale, la dichiarerai in un'intestazione e la definirai in un file sorgente adatto, e quel file di origine includerà l'intestazione per verificare la dichiarazione e la definizione e tutti i file sorgente che usano la variabile useranno l'intestazione per dichiararlo.

Corollario: non dichiarerai le variabili globali in un file sorgente: un file sorgente conterrà solo definizioni.

I file di intestazione dovrebbero dichiarare raramente funzioni `static` , con la notevole eccezione delle funzioni `static inline` che saranno definite nelle intestazioni se la funzione è necessaria in più di un file sorgente.

- I file di origine definiscono le variabili globali e le funzioni globali.
- I file di origine non dichiarano l'esistenza di variabili o funzioni globali; includono l'intestazione che dichiara la variabile o la funzione.
- I file di intestazione dichiarano variabile globale e funzioni (e tipi e altro materiale di supporto).
- I file di intestazione non definiscono variabili o funzioni eccetto `static` funzioni `inline` (`static`).

Riferimenti incrociati

- Dove documentare le funzioni in C?
- Elenco di file di intestazione standard in C e C ++
- È in `inline` senza `static` o `extern` mai utile in C99?
- Come utilizzare `extern` per condividere le variabili tra i file di origine?
- Quali sono i vantaggi di un percorso relativo come `"../include/header.h"` per un'intestazione?
- Ottimizzazione dell'inclusione dell'intestazione
- Dovrei includere ogni intestazione?

Leggi Crea e includi i file di intestazione online: https://riptutorial.com/it/c/topic/6257/crea-e-includi-i-file-di-intestazione

Capitolo 19: Dichiarazione vs definizione

Osservazioni

Fonte: qual è la differenza tra una definizione e una dichiarazione?

Sorgente (per simboli deboli e forti): https://www.amazon.com/Computer-Systems-Programmers-Perspective-2nd/dp/0136108040/

Examples

Capire Dichiarazione e Definizione

Una dichiarazione introduce un identificatore e ne descrive il tipo, sia esso un tipo, un oggetto o una funzione. Una dichiarazione è ciò che il compilatore deve accettare i riferimenti a quell'identificatore. Queste sono dichiarazioni:

```
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                         */
```

Una definizione effettivamente istanzia / implementa questo identificatore. È ciò di cui il linker ha bisogno per collegare riferimenti a tali entità. Queste sono definizioni corrispondenti alle dichiarazioni di cui sopra:

```
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) */
```

Una definizione può essere utilizzata al posto di una dichiarazione.

Tuttavia, deve essere definito esattamente una volta. Se ti dimentichi di definire qualcosa che è stato dichiarato e referenziato da qualche parte, allora il linker non sa a cosa collegare i riferimenti e si lamenta di un simbolo mancante. Se si definisce qualcosa più di una volta, il linker non sa quale delle definizioni per collegare i riferimenti e si lamenta dei simboli duplicati.

Eccezione:

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

Questa eccezione può essere spiegata usando i concetti di "Simboli forti vs simboli deboli" (dal punto di vista di un linker). Per favore guarda qui (Diapositiva 22) per ulteriori spiegazioni.

```
/* 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 */
```

Leggi Dichiarazione vs definizione online: https://riptutorial.com/it/c/topic/3104/dichiarazione-vs-definizione

Capitolo 20: dichiarazioni

Osservazioni

La dichiarazione di identificatore riferita ad oggetto o funzione viene spesso definita in breve come semplice dichiarazione di oggetto o funzione.

Examples

Chiamare una funzione da un altro file C.

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!")`
     */
}
```

main.c

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

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

Compila e collega

Per prima cosa, *compiliamo* sia `foo.c` sia `main.c` sui *file oggetto* . Qui usiamo il compilatore `gcc` , il tuo compilatore può avere un nome diverso e ha bisogno di altre opzioni.

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

Ora li colleghiamo insieme per produrre il nostro eseguibile finale:

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

Utilizzando una variabile globale

L'uso di variabili globali è generalmente scoraggiato. Rende il tuo programma più difficile da capire e più difficile da eseguire il debug. Ma a volte l'uso di una variabile globale è accettabile.

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. */
```

main.c

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

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

Vedi anche Come utilizzare `extern` per condividere variabili tra file sorgente?

Utilizzo di costanti globali

Le intestazioni possono essere utilizzate per dichiarare risorse di sola lettura utilizzate globalmente, ad esempio le tabelle di stringhe.

Dichiara quelli in un'intestazione separata che viene inclusa da qualsiasi file (" *Unità di traduzione* ") che vuole farne uso. È utile utilizzare la stessa intestazione per dichiarare un'enumerazione correlata per identificare tutte le risorse stringa:

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
```

Per definire effettivamente le risorse create un file .c correlato, vale a dire un'altra unità di traduzione che contiene le istanze effettive di ciò che è stato dichiarato nel relativo file di intestazione (.h):

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"
};
```

Un programma che usa questo potrebbe assomigliare a questo:

main.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;
}
```

Compilare i tre file sopra usando GCC e collegarli per diventare il file `main` del programma, per esempio usando questo:

```
gcc -Wall -Wextra -pedantic -Wconversion -g  main.c resources.c -o main
```

(usa questi `-Wall -Wextra -pedantic -Wconversion` per rendere il compilatore davvero pignolo, in modo da non perdere nulla prima di postare il codice in SO, dirà il mondo, o anche se vale la pena di distribuirlo in produzione)

Esegui il programma creato:

```
$ ./main
```

E prendi:

```
resource ID: 0, resource: '<unknown>'
resource ID: 1, resource: 'OK'
resource ID: 2, resource: 'Cancel'
```

```
resource ID: 3, resource: 'Abort'
```

introduzione

Esempi di dichiarazioni sono:

```
int a; /* declaring single identifier of type int */
```

La suddetta dichiarazione dichiara un identificatore singolo denominato `a` che si riferisce ad un oggetto con tipo `int` .

```
int a1, b1; /* declaring 2 identifiers of type int */
```

La seconda dichiarazione dichiara 2 identificatori denominati `a1` e `b1` che si riferiscono ad alcuni altri oggetti con lo stesso tipo `int` .

Fondamentalmente, il modo in cui funziona è come questo: prima si inserisce un tipo , quindi si scrive una singola o più espressioni separate da una virgola (`,`) (**che non verrà valutata a questo punto) e che dovrebbero altrimenti essere definite come dichiaratori in questo contesto**). Nella scrittura di tali espressioni, è consentito applicare solo gli operatori di riferimento indiretto (`*`), di chiamata di funzione (`( )`) o di indice (o di indicizzazione di matrice - `[ ]`) su un identificatore (non è possibile utilizzare alcun operatore). L'identificatore utilizzato non è richiesto per essere visibile nell'ambito corrente. Qualche esempio:

```
/* 1 */ int /* 2 */ (*z) /* 3 */ , /* 4 */ *x , /* 5 */ **c /* 6 */ ;
```

#	Descrizione
1	Il nome del tipo intero.
2	Espressione non valutata che applica indiretta a qualche identificatore `z` .
3	Abbiamo una virgola che indica che seguirà un'altra espressione nella stessa dichiarazione.
4	Espressione non valutata che applica indiretta ad un altro identificatore `x` .
5	Espressione non valutata che applica indiretta al valore dell'espressione `(*c)` .
6	Fine della dichiarazione

Si noti che nessuno degli identificatori di cui sopra era visibile prima di questa dichiarazione e quindi le espressioni utilizzate non sarebbero valide prima di essa.

Dopo ciascuna di tali espressioni, l'identificatore utilizzato in esso viene introdotto nell'ambito corrente. (Se l'identificatore ha assegnato il collegamento a esso, può anche essere dichiarato nuovamente con lo stesso tipo di collegamento in modo che entrambi gli identificatori si riferiscano

allo stesso oggetto o funzione)

Inoltre, per l'inizializzazione può essere utilizzato il segno di operatore uguale (=). Se un'espressione non valutata (dichiarante) è seguita da = all'interno della dichiarazione, diciamo che anche l'identificatore che viene introdotto viene inizializzato. Dopo il segno = possiamo mettere nuovamente un'espressione, ma questa volta verrà valutata e il suo valore sarà usato come iniziale per l'oggetto dichiarato.

Esempi:

```
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];
```

Più tardi nel tuo codice, puoi scrivere esattamente la stessa espressione dalla parte di dichiarazione dell'identificatore appena introdotto, dandoti un oggetto del tipo specificato all'inizio della dichiarazione, assumendo che tu abbia assegnato valori validi a tutti gli accessi oggetti nel modo. Esempi:

```
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 */
}
```

La dichiarazione di `b3` specifica che è possibile utilizzare potenzialmente il valore `b3` come media per accedere a qualche oggetto intero.

Ovviamente, per applicare l'indirezione (`*`) a `b3` , dovresti anche avere un valore appropriato memorizzato in esso (vedi puntatori per maggiori informazioni). Si dovrebbe anche prima memorizzare qualche valore in un oggetto prima di provare a recuperarlo (si può vedere di più su questo problema qui). Abbiamo fatto tutto questo negli esempi sopra.

```
int a3(); /* you should be able to call a3 */
```

Questo dice al compilatore che `a3` chiamare `a3` . In questo caso, `a3` riferisce alla funzione anziché a un oggetto. Una differenza tra oggetto e funzione è che le funzioni avranno sempre una sorta di collegamento. Esempi:

```
void f1()
{
    {
        int f2(); /* 1 refers to some function f2 */
    }

    {
        int f2(); /* refers to the exact same function f2 as (1) */
    }
}
```

Nell'esempio sopra, le 2 dichiarazioni si riferiscono alla stessa funzione `f2` , mentre se dichiarano oggetti, in questo contesto (avendo 2 diversi ambiti di blocco), sarebbero 2 oggetti distinti diversi.

```
int (*a3)(); /* you should be able to apply indirection to `a3` and then call it */
```

Ora sembra che si stia complicando, ma se conosci la precedenza degli operatori, avrai 0 problemi a leggere la dichiarazione di cui sopra. Le parentesi sono necessarie perché l'operatore `*` ha meno precedenza di quello `( )` .

Nel caso dell'uso dell'operatore di pedice, l'espressione risultante non sarebbe effettivamente valida dopo la dichiarazione perché l'indice utilizzato in esso (il valore all'interno `[` e `]`) sarà sempre 1 sopra il valore massimo consentito per questo oggetto / funzione.

```
int a4[5]; /* here a4 shouldn't be accessed using the index 5 later on */
```

Ma dovrebbe essere accessibile da tutti gli altri indici inferiori a 5. Esempi:

```
a4[0], a4[1]; a4[4];
```

`a4[5]` si tradurrà in UB. Ulteriori informazioni sugli array sono disponibili qui .

```
int (*a5)[5](); /* here a4 could be applied indirection
                   indexed up to (but not including) 5
                   and called */
```

Sfortunatamente per noi, sebbene sintatticamente possibile, la dichiarazione di `a5` è proibita dallo standard attuale.

typedef

I `typedef` sono dichiarazioni che hanno la parola chiave `typedef` davanti e prima del tipo. Per esempio:

```
typedef int (*(*t0)())[5];
```

(*puoi inserire tecnicamente anche il typedef dopo il tipo - come questo* `int typedef (*(*t0)())[5];` *ma questo è scoraggiato*)

Le dichiarazioni di cui sopra dichiarano un identificatore per un nome typedef. Puoi usarlo in questo modo in seguito:

```
t0 pf;
```

Quale avrà lo stesso effetto della scrittura:

```
int (*(*pf)())[5];
```

Come puoi vedere il nome typedef "salva" la dichiarazione come un tipo da usare successivamente per altre dichiarazioni. In questo modo puoi salvare alcune sequenze di tasti. Anche come dichiarazione che usa `typedef` è ancora una dichiarazione che non sei limitato solo dall'esempio sopra:

```
t0 (*pf1);
```

Equivale a:

```
int (*(**pf1)())[5];
```

Utilizzare la regola destra o sinistra per decifrare la dichiarazione C

La regola "destra-sinistra" è una regola completamente regolare per decifrare le dichiarazioni C. Può anche essere utile per crearli.

Leggi i simboli mentre li incontri nella dichiarazione ...

```
*   as "pointer to"          - always on the left side
[]  as "array of"            - always on the right side
()  as "function returning"  - always on the right side
```

Come applicare la regola

PASSO 1

Trova l'identificatore. Questo è il tuo punto di partenza. Quindi di 'te stesso, "l'identificativo è". Hai iniziato la tua dichiarazione

PASSO 2

Guarda i simboli a destra dell'identificatore. Se, per esempio, trovi `()` lì, allora sai che questa è la dichiarazione per una funzione. Quindi avresti allora *"identificatore è funzione di ritorno"* . O se hai trovato un `[]` lì, diresti *"identificatore è matrice di"* . Continua a destra finché non finisci i simboli OPPURE premi una parentesi tonda `)` . (Se si preme una parentesi tonda `(` , questo è l'inizio di un simbolo `()` , anche se c'è qualcosa tra le parentesi.

PASSAGGIO 3

Guarda i simboli a sinistra dell'identificatore. Se non è uno dei nostri simboli sopra (diciamo qualcosa come "int"), dillo e basta. Altrimenti, traducilo in inglese usando la tabella sopra. Continuate a sinistra finché non finite i simboli O colpite una parentesi sulla sinistra `(` .

Ora ripeti i passaggi 2 e 3 finché non hai formato la tua dichiarazione.

Ecco alcuni esempi:

```
int *p[];
```

Innanzitutto, trova l'identificatore:

```
int *p[];
     ^
```

"p è"

Ora, spostati verso destra finché non escono simboli o la parentesi tonda colpisce.

```
int *p[];
      ^^
```

"p è matrice di"

Non riesci più a muoverti (fuori dai simboli), quindi vai a sinistra e trova:

```
int *p[];
    ^
```

"p è una matrice di puntatore a"

Continua a sinistra e trova:

```
int *p[];
^^^
```

"p è una matrice di puntatore a int".

(o *"p è un array in cui ogni elemento è di tipo puntatore a int"*)

Un altro esempio:

```
int *(*func())();
```

Trova l'identificatore.

```
int *(*func())();
       ^^^^
```

"func is"

Vai a destra.

```
int *(*func())();
           ^^
```

"func is function return"

Non posso più muoverti a destra a causa della parentesi giusta, quindi vai a sinistra.

```
int *(*func())();
      ^
```

"func è la funzione che restituisce il puntatore a"

Non riesci più a spostarti a sinistra a causa della parentesi sinistra, quindi continua a andare a destra.

```
int *(*func())();
              ^^
```

"func è la funzione che restituisce il puntatore alla funzione che restituisce"

Non posso più muoverti perché non abbiamo più simboli, quindi vai a sinistra.

```
int *(*func())();
    ^
```

"func è la funzione che restituisce il puntatore alla funzione restituendo il puntatore a"

E infine, continua a sinistra, perché non è rimasto nulla a destra.

```
int *(*func())();
^^^
```

"func è la funzione che restituisce il puntatore alla funzione restituisce il puntatore a int".

Come puoi vedere, questa regola può essere abbastanza utile. Puoi anche usarlo per controllare se stessi mentre stai creando dichiarazioni e per darti un suggerimento su dove mettere il prossimo simbolo e se sono necessarie le parentesi.

Alcune dichiarazioni sembrano molto più complicate di quanto non siano dovute alle dimensioni degli array e agli elenchi di argomenti in forma di prototipo. Se vedi `[3]` , viene letto come *"array (size 3) of ..."* . Se vedi `(char *,int)` che viene letto come * "function expect (char *, int) e return ..."* .

Ecco uno divertente:

```
int (*(*fun_one)(char *,double))[9][20];
```

Non passerò attraverso ciascuno dei passaggi per decifrare questo.

* "fun_one è puntatore alla funzione che si aspetta (char *, double) e restituisce il puntatore all'array (size 9) dell'array (size 20) di int."*

Come puoi vedere, non è così complicato se ti sbarazzi delle dimensioni dell'array e degli elenchi di argomenti:

```
int (*(*fun_one)())[][];
```

È possibile decifrarlo in questo modo e quindi inserire le dimensioni dell'array e gli elenchi di argomenti in un secondo momento.

Alcune parole finali:

È abbastanza possibile fare dichiarazioni illegali usando questa regola, quindi è necessaria una certa conoscenza di ciò che è legale in C. Ad esempio, se quanto sopra fosse stato:

```
int *((*fun_one)())[][];
```

avrebbe letto *"fun_one è un puntatore alla funzione che restituisce una matrice di puntatore a int"* . Poiché una funzione non può restituire un array, ma solo un puntatore a un array, tale dichiarazione è illegale.

Le combinazioni illegali includono:

```
[]() - cannot have an array of functions
()() - cannot have a function that returns a function
()[] - cannot have a function that returns an array
```

In tutti i casi di cui sopra, è necessario un insieme di parentesi per associare un simbolo `*` a sinistra tra questi simboli `()` e `[]` destra, affinché la dichiarazione sia legale.

Ecco alcuni esempi:

---

legale

```
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
```

Illegale

```
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
```

Fonte: http://ieng9.ucsd.edu/~cs30x/rt_lt.rule.html

Leggi dichiarazioni online: https://riptutorial.com/it/c/topic/3729/dichiarazioni

Capitolo 21: Dichiarazioni di selezione

Examples

if () Dichiarazioni

Uno dei modi più semplici per controllare il flusso del programma è quello di utilizzare `if` istruzioni di selezione. Questa affermazione può decidere se un blocco di codice deve essere eseguito o non eseguito.

La sintassi per `if` dichiarazione selezione in C potrebbe essere la seguente:

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

Per esempio,

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

Dove `a > 1` è una *condizione* che deve essere valutata a `true` per eseguire le istruzioni all'interno del blocco `if` . In questo esempio "a è maggiore di 1" viene stampato solo se `a > 1` è vero.

`if` istruzioni di selezione possono tralasciare le parentesi graffe `{` e `}` se c'è una sola affermazione all'interno del blocco. L'esempio sopra può essere riscritto a

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

Tuttavia, per eseguire più istruzioni all'interno di un blocco, le parentesi devono essere utilizzate.

La *condizione* per `if` può includere più espressioni. `if` eseguirà l'azione solo se il risultato finale dell'espressione è vero.

Per esempio

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

eseguirà solo `printf` e `a++` se **sia** `a` che `b` sono maggiori di `1` .

if () ... else istruzioni e sintassi

Mentre `if` esegue un'azione solo quando la sua condizione viene valutata su `true` , `if` / `else` consente di specificare le diverse azioni quando la condizione è `true` e quando la condizione è `false` .

Esempio:

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

Proprio come il `if` dichiarazione, quando il blocco all'interno di `if` o `else` è composto da una sola istruzione, quindi le parentesi graffe può essere omessa (ma così facendo non è raccomandato in quanto può facilmente introdurre problemi involontariamente). Tuttavia, se c'è più di un'istruzione nel blocco `if` o `else` , le parentesi devono essere utilizzate su quel particolare blocco.

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

switch () Dichiarazioni

`switch` istruzioni `switch` sono utili quando si desidera che il programma esegua molte cose diverse in base al valore di una particolare variabile di test.

Un esempio di utilizzo dell'istruzione `switch` è simile a questo:

```
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;
}
```

Questo esempio è equivalente a

```
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");
}
```

Se il valore di `a` è 1 quando viene utilizzata l'istruzione `switch` , verrà stampato `a is 1` . Se il valore di `a` è 2 allora, `a is 2` verrà stampato. Altrimenti, `a is neither 1 nor 2` verrà stampato `a is neither 1 nor 2` .

`case n:` è usato per descrivere dove salterà il flusso di esecuzione quando il valore passato all'istruzione `switch` è *n* . *n* deve essere una costante in fase di compilazione e lo stesso *n* può esistere al massimo una volta in un'unica istruzione `switch` .

`default:` è usato per descrivere che quando il valore non corrisponde a nessuna delle scelte per il `case n:` È buona norma includere un caso `default` in ogni istruzione switch per rilevare comportamenti imprevisti.

Una `break;` è richiesta una dichiarazione per saltare fuori dal blocco `switch` .

Nota: se dimentichi accidentalmente di aggiungere `break` dopo la fine di un `case` , il compilatore presupporrà che intendi "fallire" e tutte le successive istruzioni caso, se presenti, verranno eseguite (a meno che non venga trovata un'istruzione *break)* uno qualsiasi dei casi successivi), indipendentemente dal fatto che le dichiarazioni dei casi susseguenti corrispondano o meno. Questa particolare proprietà viene utilizzata per implementare il dispositivo di Duff . Questo comportamento è spesso considerato un difetto nelle specifiche del linguaggio C.

Di seguito è riportato un esempio che mostra gli effetti dell'assenza di `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;
}
```

Quando il valore di `a` è 1 o 2, `a is 1 or 2` e `a is 1, 2 or 3` saranno entrambi stampati. Quando `a` è 3, verrà stampato solo `a is 1, 2 or 3` . Altrimenti, `a is neither 1, 2 nor 3` stampato `a is neither 1, 2 nor 3` .

Notare che il caso `default` non è necessario, specialmente quando il set di valori che si ottiene nello `switch` è finito e noto al momento della compilazione.

Il miglior esempio è usare un `switch` su un `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;
  }
}
```

Ci sono molti vantaggi nel fare questo:

- la maggior parte dei compilatori segnalerà un avviso se non si gestisce un valore (questo non verrebbe segnalato se fosse presente un caso `default`)
- per lo stesso motivo, se aggiungi un nuovo valore `enum` , riceverai una notifica di tutti i luoghi in cui hai dimenticato di gestire il nuovo valore (con un caso `default` , avresti bisogno di esplorare manualmente il tuo codice alla ricerca di tali casi)
- Il lettore non ha bisogno di capire "cosa è nascosto dal `default:` ", se esistono altri valori `enum` o se è una protezione per "just in case". E se ci sono altri valori di `enum` , il programmatore ha intenzionalmente usato il caso `default` o c'è un bug che è stato introdotto quando ha aggiunto il valore?
- la gestione di ogni valore `enum` rende il codice auto esplicativo in quanto non è possibile nascondersi dietro una wild card, è necessario gestirne esplicitamente ciascuno.

Tuttavia, non puoi impedire a qualcuno di scrivere codice malvagio come:

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

Quindi puoi aggiungere un controllo extra prima del tuo passaggio per rilevarlo, se davvero ne hai bisogno.

```
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 Concatenare due o più if () ... else statements

Mentre l'istruzione `if ()... else` consente di definire solo un comportamento (predefinito) che si verifica quando la condizione all'interno di `if ()` non viene soddisfatta, concatenando due o più `if () ... else`

istruzioni consentono di definire una coppia più comportamenti prima di andare all'ultimo `else` ramo che funge da "default", se presenti.

Esempio:

```
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");
}
```

Nested if () ... else VS if () .. else Ladder

Nested `if()...else` istruzioni richiedono più tempo di esecuzione (sono più lente) rispetto a `if()...else` ladder perché le istruzioni nidificate `if()...else` controllano tutte le istruzioni condizionali interne una volta che l'esterno l'istruzione condizionale `if()` è soddisfatta, mentre `if()..else` ladder interrompe il test di condizione una volta che una delle istruzioni condizionali `if()` o `else if()` sono vere.

An `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;
}
```

È, nel caso generale, considerato migliore del nidificato equivalente `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;
}
```

Leggi Dichiarazioni di selezione online: https://riptutorial.com/it/c/topic/3073/dichiarazioni-di-selezione

Capitolo 22: Discussioni (native)

Sintassi

- `#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);`

Osservazioni

I thread C11 sono una funzionalità opzionale. La loro assenza può essere verificata con `__STDC__NO_THREAD__` . Attualmente (luglio 2016) questa funzione non è ancora implementata da tutte le librerie C che altrimenti supportano C11.

Le librerie C note per supportare i thread C11 sono:

- MUSL

Le librerie C che non supportano i thread C11, tuttavia:

- gnu libc

Examples

Inizia diversi thread

```
#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;
}
```

Inizializzazione da un thread

Nella maggior parte dei casi tutti i dati a cui si accede da più thread devono essere inizializzati prima della creazione dei thread. Ciò garantisce che tutti i thread inizino con uno stato chiaro e non *si* verifichi alcuna *condizione di competizione* .

Se questo non è possibile, puoi utilizzare `once_flag` e `call_once`

```
#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;
}
```

Il `once_flag` viene utilizzato per coordinare diversi thread che potrebbero voler inizializzare gli stessi dati `Big` . La chiamata a `call_once` garantisce

- `initBig` viene chiamato esattamente una volta
- `call_once` blocca fino a `initBig` è stata effettuata una chiamata a `initBig` , dallo stesso o da un altro thread.

Oltre all'allocazione, una cosa tipica da fare in una funzione così definita è l'inizializzazione dinamica di strutture di dati di controllo del thread come `mtx_t` o `cnd_t` che non possono essere inizializzate staticamente, usando rispettivamente `mtx_init` o `cnd_init` .

Leggi Discussioni (native) online: https://riptutorial.com/it/c/topic/4432/discussioni--native-

Capitolo 23: Effetti collaterali

Examples

Operatori di Pre / Post Increment / Decrement

In C ci sono due operatori unari - '++' e '-' che sono una fonte molto comune di confusione. L'operatore `++` è chiamato l' *operatore di incremento* e l'operatore `--` è chiamato *operatore di decremento* . Possono essere utilizzati entrambi in forma di *prefisso* o *postfix* . La sintassi per il modulo prefisso per l'operatore `++` è l' `++operand` e la sintassi per il modulo postfix è `operand++` . Quando viene utilizzato nel modulo prefisso, l'operando viene incrementato per primo di `1` e il valore risultante dell'operando viene utilizzato nella valutazione dell'espressione. Considera il seguente esempio:

```
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 ; */
```

Se utilizzato nel modulo postfix, il valore corrente dell'operando viene utilizzato nell'espressione e quindi il valore dell'operando viene incrementato di `1` . Considera il seguente esempio:

```
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; */
```

Il funzionamento dell'operatore decremento `--` può essere compresa in modo simile.

Il seguente codice dimostra cosa fa ognuno

```
    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;
    }
```

Da quanto sopra è chiaro che gli operatori postali restituiscono il valore corrente di una variabile e *quindi la* modificano, ma i pre operatori modificano la variabile e *quindi* restituiscono il valore modificato.

In tutte le versioni di C, l'ordine di valutazione degli operatori pre e post non è definito, quindi il

codice seguente può restituire output non previsti:

```
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;
}
```

Si noti che è anche una buona pratica usare gli operatori pre-post quando usati da soli in una dichiarazione. Guarda il codice sopra per questo.

Si noti inoltre che quando viene chiamata una funzione, tutti gli effetti collaterali sugli argomenti devono aver luogo prima dell'esecuzione della funzione.

```
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;
}
```

Leggi Effetti collaterali online: https://riptutorial.com/it/c/topic/7094/effetti-collaterali

Capitolo 24: enumerazioni

Osservazioni

Le enumerazioni sono costituite dalla parola chiave `enum` e da un *identificativo* facoltativo seguito da una *lista di enumeratori* racchiusa tra parentesi graffe.

Un *identificatore* è di tipo `int` .

L' *elenco enumeratore* ha almeno un elemento *enumeratore* .

Opzionalmente, un *enumeratore* può essere "assegnato" a un'espressione costante di tipo `int` .

Un *enumeratore* è costante ed è compatibile con un `char` , un intero con segno o un intero senza segno. Che mai usato è definito dall'implementazione . In ogni caso il tipo utilizzato dovrebbe essere in grado di rappresentare tutti i valori definiti per l'enumerazione in questione.

Se nessuna espressione costante è "assegnato" per un *enumeratore* ed è il 1 ° ingresso in un *enumeratore-list* prende valore `0` , altrimenti ottenere prende il valore della voce precedente nella *enumeratore-list* più 1.

L'utilizzo di più "assegnamenti" può portare a diversi *enumeratori* della stessa enumerazione con gli stessi valori.

Examples

Enumerazione semplice

Un'enumerazione è un tipo di dati definito dall'utente costituito da costanti integrali e a ciascuna costante integrale viene assegnato un nome. L'enumerazione delle parole `enum` viene utilizzata per definire il tipo di dati enumerati.

Se si utilizza `enum` anziché `int` o `string/ char*` , si aumenta il controllo in fase di compilazione e si evitano errori nel passaggio di costanti non valide e si documentano quali valori sono legali da utilizzare.

Esempio 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);
}
```

Con una funzione principale definita come segue (ad esempio):

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

C99

Esempio 2

(Questo esempio utilizza inizializzatori designati che sono standardizzati dal C99.)

```
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]);
}
```

Lo stesso esempio usando il controllo del range:

```
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

Ci sono diverse possibilità e convenzioni per nominare un'enumerazione. Il primo è usare un *nome di tag* subito dopo la parola chiave `enum` .

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

Questa enumerazione deve quindi essere sempre utilizzata con la parola chiave *e* il tag in questo modo:

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

Se usiamo `typedef` direttamente quando dichiariamo l' `enum` , possiamo omettere il nome del tag e quindi usare il tipo senza la parola chiave `enum` :

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

color chosenColor = RED;
```

Ma in quest'ultimo caso non possiamo usarlo come `enum color` , perché non abbiamo usato il nome del tag nella definizione. Una convenzione comune è usare entrambi, in modo che lo stesso nome possa essere usato con o senza la parola chiave `enum` . Questo ha il particolare vantaggio di essere compatibile con C ++

```
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;
```

Funzione:

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

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

Per ulteriori informazioni su `typedef` vedere Typedef

Enumerazione con valore duplicato

Un valore di enumerazioni non deve in alcun modo essere univoco:

```
#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;
}
```

Il campione stampa:

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

costante di enumerazione senza nome tipografico

I tipi di enumerazione possono anche essere dichiarati senza dare loro un nome:

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

Questo ci consente di definire costanti di tempo di compilazione di tipo `int` che possono essere usate come lunghezza di array come in questo esempio.

Leggi enumerazioni online: https://riptutorial.com/it/c/topic/5460/enumerazioni

Capitolo 25: File e flussi I / O

Sintassi

- #include <stdio.h> / * Includi questo per utilizzare una delle seguenti sezioni * /
- FILE * fopen (percorso const char *, modalità const char *); / * Apri un flusso sul file nel *percorso* con la *modalità* specificata * /
- FILE * freopen (percorso const char *, modo const char *, flusso FILE *); / * Riapri un flusso esistente sul file nel *percorso* con la *modalità* specificata * /
- int fclose (flusso FILE *); / * Chiudi uno stream aperto * /
- size_t fread (void * ptr, size_t size, size_t nmemb, FILE * stream); / * Legge al massimo *nmemb di* elementi di *dimensione* ciascuno dallo *stream* e li scrive in *ptr* . Restituisce il numero di elementi letti. * /
- size_t fwrite (const void * ptr, size_t size, size_t nmemb, FILE * stream); / * Scrivi *nmemb* elementi di byte di *dimensione* ciascuno da *ptr* allo *stream* . Restituisce il numero di elementi scritti. * /
- int fseek (flusso FILE *, offset lungo, int quale); / * Imposta il cursore del flusso da *sfalsare* , relativo all'offset detto da *dove* , e restituisce 0 se ha avuto successo. * /
- ftell lungo (flusso FILE *); / * Restituisce l'offset della posizione corrente del cursore dall'inizio del flusso. * /
- void rewind (flusso FILE *); / * Imposta la posizione del cursore all'inizio del file. * /
- int fprintf (FILE * fout, const char * fmt, ...); / * Scrive la stringa di formato printf su fout * /
- FILE * stdin; / * Flusso di input standard * /
- FILE * stdout; / * Flusso di output standard * /
- FILE * stderr; / * Flusso di errore standard * /

Parametri

Parametro	Dettagli
const char * mode	Una stringa che descrive la modalità di apertura del flusso supportato da file. Vedi osservazioni per possibili valori.
int whence	Può essere `SEEK_SET` per impostare dall'inizio del file, `SEEK_END` per impostare dalla sua fine, o `SEEK_CUR` per impostare relativo al valore corrente del cursore. Nota: `SEEK_END` non è portabile.

Osservazioni

Stringhe di modalità:

Le stringhe di modalità in `fopen()` e `freopen()` possono essere uno di quei valori:

- `"r"` : apre il file in modalità di sola lettura, con il cursore impostato all'inizio del file.
- `"r+"` : apre il file in modalità lettura-scrittura, con il cursore impostato all'inizio del file.
- `"w"` : apre o crea il file in modalità solo scrittura, con il contenuto troncato a 0 byte. Il cursore è impostato all'inizio del file.
- `"w+"` : apre o crea il file in modalità lettura-scrittura, con il contenuto troncato a 0 byte. Il cursore è impostato all'inizio del file.
- `"a"` : apre o crea il file in modalità solo scrittura, con il cursore impostato alla fine del file.
- `"a+"` : apre o crea il file in modalità lettura-scrittura, con il cursore di lettura impostato all'inizio del file. L'output, tuttavia, verrà *sempre* aggiunto alla fine del file.

Ognuna di queste modalità di file può avere una `b` aggiunta dopo la lettera iniziale (es. `"rb"` o `"a+b"` o `"ab+"`). Il `b` significa che il file deve essere trattato come un file binario invece di un file di testo su quei sistemi in cui vi è una differenza. Non fa differenza sui sistemi di tipo Unix; è importante sui sistemi Windows. (Inoltre, Windows `fopen` consente una `t` esplicita invece di `b` per indicare 'file di testo' - e numerose altre opzioni specifiche della piattaforma.)

C11

- `"wx"` : crea un file di testo in modalità di sola scrittura. *Il file potrebbe non esistere* .
- `"wbx"` : crea un file binario in modalità solo scrittura. *Il file potrebbe non esistere* .

La `x` , se presente, deve essere l'ultimo carattere nella stringa della modalità.

Examples

Apri e scrivi su file

```
#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;
}
```

Questo programma apre il file con il nome dato nell'argomento principale, per default su `output.txt` se non viene fornito alcun argomento. Se esiste già un file con lo stesso nome, il suo contenuto viene scartato e il file viene trattato come un nuovo file vuoto. Se i file non esistono già, la chiamata `fopen()` lo crea.

Se la chiamata `fopen()` fallisce per qualche motivo, restituisce un valore `NULL` e imposta il valore della variabile `errno` globale. Ciò significa che il programma può testare il valore restituito dopo la chiamata a `fopen()` e usare `perror()` se `fopen()` fallisce.

Se la chiamata `fopen()` positivo, restituisce un puntatore `FILE` valido. Questo puntatore può quindi essere utilizzato per fare riferimento a questo file finché non viene chiamato `fclose()` .

La funzione `fputs()` scrive il testo specificato nel file aperto, sostituendo qualsiasi contenuto precedente del file. Analogamente a `fopen()` , la funzione `fputs()` imposta anche il valore `errno` se fallisce, anche se in questo caso la funzione restituisce `EOF` per indicare il fail (altrimenti restituisce un valore non negativo).

La `fclose()` svuota tutti i buffer, chiude il file e libera la memoria indicata da `FILE *` . Il valore restituito indica il completamento proprio come fa `fputs()` (sebbene restituisca '0' se ha successo), anche in questo caso imposta nuovamente il valore `errno` nel caso di un errore.

fprintf

È possibile utilizzare `fprintf` su un file proprio come si potrebbe su una console con `printf` . Ad esempio, per tenere traccia delle vittorie, perdite e legami del gioco che potresti scrivere

```
/* 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);
}
```

Una nota a margine: alcuni sistemi (famigeratamente, Windows) non usano quello che la maggior parte dei programmatori chiamerebbe "normale" terminazioni di linea. Mentre i sistemi simili a UNIX usano \ n per terminare le linee, Windows usa una coppia di caratteri: \ r (ritorno a capo) e \ n (avanzamento riga). Questa sequenza è comunemente chiamata CRLF. Tuttavia, ogni volta che si utilizza C, non è necessario preoccuparsi di questi dettagli altamente dipendenti dalla piattaforma. Il compilatore AC è necessario per convertire ogni istanza di \ n alla terminazione della linea della piattaforma corretta. Quindi un compilatore di Windows converte \ n in \ r \ n, ma un compilatore UNIX lo manterrebbe così com'è.

Esegui processo

```
#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;
}
```

Questo programma esegue un processo (`netstat`) tramite `popen()` e legge tutto lo standard output dal processo e lo fa eco allo standard output.

Nota: `popen()` non esiste nella libreria C standard , ma è piuttosto una parte di POSIX C)

Ottieni linee da un file usando getline ()

La libreria POSIX C definisce la funzione `getline()` . Questa funzione alloca un buffer per contenere il contenuto della riga e restituisce la nuova riga, il numero di caratteri nella riga e la dimensione del buffer.

Esempio di programma che ottiene ogni riga da `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;
}
```

Inserisci il file `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.
```

Produzione

```
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:
```

Nell'esempio, `getline()` viene inizialmente chiamato senza alcun buffer allocato. Durante questa prima chiamata, `getline()` assegna un buffer, legge la prima riga e posiziona il contenuto della linea nel nuovo buffer. Nelle chiamate successive, `getline()` aggiorna lo stesso buffer e rialloca solo il buffer quando non è più abbastanza grande da adattarsi all'intera linea. Il buffer temporaneo viene quindi liberato quando abbiamo finito con il file.

Un'altra opzione è `getdelim()` . È lo stesso di `getline()` tranne per il carattere di fine riga. Questo è necessario solo se l'ultimo carattere della linea per il tuo tipo di file non è '\ n'. `getline()` funziona anche con i file di testo di Windows perché con la fine della riga multibyte (`"\r\n"`) '\ n'` è ancora l'ultimo carattere sulla riga.

Esempio di implementazione di `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
```

Apri e scrivi su un file binario

```
#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;
}
```

Questo programma crea e scrive testo in forma binaria tramite la `fwrite` funzione al file `output.bin` .

Se esiste già un file con lo stesso nome, il suo contenuto viene scartato e il file viene trattato come un nuovo file vuoto.

Un flusso binario è una sequenza ordinata di caratteri che può registrare in modo trasparente i dati interni. In questa modalità, i byte vengono scritti tra il programma e il file senza alcuna interpretazione.

Per scrivere in modo portabile gli interi, è necessario sapere se il formato del file li prevede in

formato big o little-endian e le dimensioni (in genere 16, 32 o 64 bit). Il cambio di bit e il mascheramento possono quindi essere utilizzati per scrivere i byte nell'ordine corretto. Non è garantito che gli interi in C abbiano la rappresentazione del complemento a due (sebbene quasi tutte le implementazioni lo facciano). Fortunatamente una conversione a unsigned è garantita per utilizzare due complementi. Il codice per scrivere un intero con segno su un file binario è quindi un po 'sorprendente.

```
/* 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;
}
```

Le altre funzioni seguono lo stesso schema con modifiche minori per dimensioni e ordine dei byte.

fscanf ()

Diciamo che abbiamo un file di testo e vogliamo leggere tutte le parole in quel file, al fine di fare alcuni requisiti.

file.txt :

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

Questa è la funzione principale:

```
#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++;
    }
}
```

L'output sarà:

```
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()
```

Leggi le righe da un file

L'intestazione `stdio.h` definisce la funzione `fgets()` . Questa funzione legge una riga da uno stream e la memorizza in una stringa specificata. La funzione interrompe la lettura del testo dallo stream quando vengono letti `n - 1` carattere, viene letto il carattere di nuova riga (`'\n'`) o viene raggiunta la fine del file (EOF).

```
#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);
    }
}
```

Chiamando il programma con un argomento che è un percorso di un file contenente il seguente testo:

```
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.
```

Risulterà nel seguente output:

```
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]:
```

Questo esempio molto semplice consente una lunghezza massima della linea fissa, in modo tale che le linee più lunghe vengano effettivamente contate come due linee. La funzione `fgets()` richiede che il codice chiamante fornisca la memoria da utilizzare come destinazione per la linea che viene letta.

POSIX rende disponibile la funzione `getline()` che invece alloca internamente la memoria per ingrandire il buffer come necessario per una linea di qualsiasi lunghezza (purché vi sia memoria sufficiente).

Leggi File e flussi I / O online: https://riptutorial.com/it/c/topic/507/file-e-flussi-i---o

Capitolo 26: Generazione di numeri casuali

Osservazioni

A causa dei difetti di `rand()` , molte altre implementazioni predefinite sono emerse nel corso degli anni. Tra questi ci sono:

- `arc4random()` (disponibile su OS X e BSD)
- `random()` (disponibile su Linux)
- `drand48()` (disponibile su POSIX)

Examples

Generazione di numeri casuali di base

La funzione `rand()` può essere utilizzata per generare un valore intero pseudo-casuale compreso tra `0` e `RAND_MAX` (incluso `0` e `RAND_MAX`).

`srand(int)` viene utilizzato per seminare il generatore di numeri pseudo-casuali. Ogni volta che `rand()` viene seminato con lo stesso seme, deve produrre la stessa sequenza di valori. Dovrebbe essere seminato solo una volta prima di chiamare `rand()` . Non dovrebbe essere ripetutamente seminato o risettato ogni volta che si desidera generare una nuova serie di numeri pseudocasuali.

La pratica standard consiste nell'usare il risultato del `time(NULL)` come seme. Se il generatore di numeri casuali richiede una sequenza deterministica, è possibile inizializzare il generatore con lo stesso valore all'inizio di ciascun programma. Generalmente non è richiesto per il codice di rilascio, ma è utile nelle esecuzioni di debug per rendere riproducibili i bug.

Si consiglia di seminare il generatore sempre, se non seminato, si comporta come se fosse seminato con `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;
}
```

Uscita possibile:

```
Random value between [0, 2147483647]: 823321433
```

Gli appunti:

Lo standard C non garantisce la qualità della sequenza casuale prodotta. In passato, alcune implementazioni di `rand()` avevano gravi problemi nella distribuzione e casualità dei numeri generati. **L'uso di `rand()` non è raccomandato per esigenze serie di generazione di numeri casuali, come la crittografia.**

Generatore Congruenziale Permutato

Ecco un generatore di numeri casuali standalone che non si basa su `rand()` o funzioni di libreria simili.

Perché vorresti una cosa del genere? Forse non ti fidi del generatore di numeri casuali incorporato della tua piattaforma, o forse vuoi una fonte riproducibile di casualità indipendente da una particolare implementazione di libreria.

Questo codice è PCG32 da pcg-random.org , un RNG moderno, veloce e generico con eccellenti proprietà statistiche. Non è crittograficamente sicuro, quindi non usarlo per la crittografia.

```
#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);
}
```

Ed ecco come chiamarlo:

```
#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;
}
```

Limita la generazione a un determinato intervallo

Di solito quando si generano numeri casuali è utile generare numeri interi all'interno di un intervallo, o un valore di ap compreso tra 0.0 e 1.0. Mentre l'operazione di modulo può essere utilizzata per ridurre il seme a un numero intero basso utilizza i bit bassi, che spesso passano attraverso un ciclo breve, risultando in una leggera distorsione della distribuzione se N è grande in proporzione a RAND_MAX.

La macro

```
#define uniform() (rand() / (RAND_MAX + 1.0))
```

produce un valore di ap su 0.0 a 1.0 - epsilon, quindi

```
i = (int)(uniform() * N)
```

imposterà `i` a un numero casuale uniforme compreso tra 0 e N - 1.

Sfortunatamente c'è un difetto tecnico, nel senso che RAND_MAX può essere più grande di una variabile di tipo `double` può rappresentare con precisione. Ciò significa che `RAND_MAX + 1.0` restituisce RAND_MAX e la funzione restituisce occasionalmente unità. Questo è improbabile comunque.

Generazione Xorshift

Un'alternativa buona e facile alle procedure `rand()` imperfette, è *xorshift* , una classe di generatori di numeri pseudo-casuali scoperti da George Marsaglia . Il generatore di xorshift è tra i più veloci generatori di numeri casuali non crittograficamente sicuri. Ulteriori informazioni e altre implementazioni di esempio sono disponibili sulla pagina xorshift di Wikipedia

Esempio di implementazione

```
#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;
}
```

Leggi Generazione di numeri casuali online: https://riptutorial.com/it/c/topic/365/generazione-di-numeri-casuali

Capitolo 27: Gestione degli errori

Sintassi

- #include <errno.h>
- int errno; / * implementazione definita * /
- #include <string.h>
- char * strerror (int errnum);
- #include <stdio.h>
- void perror (const char * s);

Osservazioni

Tieni presente che `errno` non è necessariamente una variabile ma che la sintassi è solo un'indicazione di come *potrebbe* essere stata dichiarata. Su molti sistemi moderni con interfacce di thread `errno` è una macro che si risolve in un oggetto che è locale al thread corrente.

Examples

errno

Quando una funzione di libreria standard fallisce, spesso imposta errno al codice di errore appropriato. Lo standard C richiede almeno 3 valori per errno essere impostato:

Valore	Senso
EDOM	Errore di dominio
ERANGE	Errore di intervallo
EILSEQ	Sequenza di caratteri multi-byte non valida

strerror

Se `perror` non è abbastanza flessibile, è possibile ottenere una descrizione di errore leggibile dall'utente chiamando `strerror` da `<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

Per stampare un messaggio di errore leggibile dall'utente su `stderr` , chiamare `perror` da <stdio.h>.

```
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;
}
```

Questo stamperà un messaggio di errore relativo al valore corrente di `errno` .

Leggi Gestione degli errori online: https://riptutorial.com/it/c/topic/2486/gestione-degli-errori

Capitolo 28: Gestione del segnale

Sintassi

- void (* signal (int sig, void (* func) (int))) (int);

Parametri

Parametro	Dettagli
sig	Il segnale per impostare il gestore di segnali su `SIGABRT` , `SIGFPE` , `SIGILL` , `SIGTERM` , `SIGINT` , `SIGSEGV` o qualche valore definito dall'implementazione
func	Il gestore di segnale, che è uno dei seguenti: `SIG_DFL` , per il gestore predefinito, `SIG_IGN` per ignorare il segnale, o un puntatore a funzione con la firma `void foo(int sig);` .

Osservazioni

L'uso di gestori di segnale con le sole garanzie dello standard C impone varie limitazioni che possono o non possono essere eseguite nel gestore di segnali definito dall'utente.

- Se la funzione definita dall'utente restituisce durante la gestione di `SIGSEGV` , `SIGFPE` , `SIGILL` o qualsiasi altro interrupt hardware definito dall'implementazione, il comportamento non è definito dallo standard C. Ciò è dovuto al fatto che l'interfaccia di C non fornisce i mezzi per modificare lo stato di errore (ad es. Dopo una divisione di `0`) e quindi quando restituisce il programma è esattamente lo stesso stato errato di prima dell'interruzione dell'hardware.
- Se la funzione definita dall'utente è stata chiamata come risultato di una chiamata per `abort` , o `raise` , il gestore di segnali non può chiamare di nuovo `raise` .
- I segnali possono arrivare nel bel mezzo di qualsiasi operazione, e quindi l'indivisibilità delle operazioni non può generalmente essere garantita né la gestione dei segnali funziona bene con l'ottimizzazione. Pertanto tutte le modifiche ai dati in un gestore di segnali devono essere relative a variabili
    - di tipo `sig_atomic_t` (tutte le versioni) o un tipo atomico senza blocco (dal C11, opzionale)
    - che sono qualificati `volatile` .
- Altre funzioni dalla libreria standard C di solito non rispettano queste restrizioni, perché possono cambiare le variabili nello stato globale del programma. Lo standard C fornisce solo garanzie per l' `abort` , `_Exit` (dal C99), `quick_exit` (dal C11), il `signal` (per lo stesso numero di segnale) e alcune operazioni atomiche (dal C11).

Il comportamento non è definito dallo standard C se una qualsiasi delle regole precedenti viene violata. Le piattaforme possono avere estensioni specifiche, ma generalmente non sono portatili oltre tale piattaforma.

- Di solito i sistemi dispongono di una propria lista di funzioni che sono *sicure per il segnale asincrono* , ovvero delle funzioni della libreria C che possono essere utilizzate da un gestore di segnali. Ad esempio, `printf` è tra queste funzioni.
- In particolare, lo standard C non definisce molto l'interazione con la sua interfaccia di thread (dal C11) o alcuna libreria di thread specifica della piattaforma come i thread POSIX. Tali piattaforme devono specificare l'interazione di tali librerie di thread con i segnali da soli.

Examples

Gestione dei segnali con "signal ()"

I numeri di segnale possono essere sincroni (come `SIGSEGV` - errore di segmentazione) quando vengono attivati da un malfunzionamento del programma stesso o asincrono (come `SIGINT` - attenzione interattiva) quando vengono avviati dall'esterno del programma, ad esempio da un tasto premuto come `Cntrl-C` .

La funzione `signal()` è parte dello standard ISO C e può essere utilizzata per assegnare una funzione per gestire un segnale specifico

```
#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;
}
```

L'uso di `signal()` impone limitazioni importanti a ciò che si è autorizzati a fare all'interno dei gestori di segnale, vedere le osservazioni per ulteriori informazioni.

POSIX raccomanda l'uso di `sigaction()` invece di `signal()` , a causa del suo comportamento sotto-specificato e delle significative variazioni di implementazione. POSIX definisce anche molti più

segnali rispetto allo standard ISO C, inclusi `SIGUSR1` e `SIGUSR2` , che possono essere utilizzati liberamente dal programmatore per qualsiasi scopo.

Leggi Gestione del segnale online: https://riptutorial.com/it/c/topic/453/gestione-del-segnale

Capitolo 29: Gestione della memoria

introduzione

Per la gestione della memoria allocata dinamicamente, la libreria C standard fornisce le funzioni `malloc()` , `calloc()` , `realloc()` e `free()` . In C99 e `aligned_alloc()` successive, c'è anche `aligned_alloc()` . Alcuni sistemi forniscono anche `alloca()` .

Sintassi

- void * aligned_alloc (size_t alignment, size_t size); / * Solo dopo 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); / * da alloca.h, non standard, non portatile, pericoloso. * /

Parametri

nome	descrizione
dimensione (`malloc` , `realloc` e `aligned_alloc`)	dimensione totale della memoria in byte. Per `aligned_alloc` la dimensione deve essere un multiplo integrale di allineamento.
dimensione (`calloc`)	dimensione di ogni elemento
Nelements	numero di elementi
PTR	puntatore alla memoria allocata precedentemente restituita da `malloc` , `calloc` , `realloc` o `aligned_alloc`
allineamento	allineamento della memoria allocata

Osservazioni

C11

Notare che `aligned_alloc()` è solo definito per C11 o successivo.

Sistemi come quelli basati su POSIX forniscono altri modi per allocare memoria allineata (ad es. `posix_memalign()`) e hanno anche altre opzioni di gestione della memoria (ad esempio `mmap()`).

Examples

Liberare memoria

È possibile rilasciare memoria allocata dinamicamente chiamando free () .

```
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 */
```

La memoria puntata da `p` viene recuperata (sia dall'implementazione di libc o dal sistema operativo sottostante) dopo la chiamata a `free()` , quindi l'accesso a quel blocco di memoria liberato tramite `p` porterà a un comportamento indefinito . I puntatori che fanno riferimento a elementi di memoria che sono stati liberati vengono comunemente chiamati puntatori penzolanti e presentano un rischio per la sicurezza. Inoltre, lo standard C afferma che anche l' accesso al valore di un puntatore pendente ha un comportamento indefinito. Si noti che il puntatore `p` può essere riproposto come mostrato sopra.

Si noti che è possibile chiamare `free()` sui puntatori che sono stati restituiti direttamente dalle funzioni `malloc()` , `calloc()` , `realloc()` e `aligned_alloc()` o dove la documentazione indica che la memoria è stata allocata in quel modo (funzioni come `strdup ()` sono esempi notevoli). Liberare un puntatore che è,

- ottenuto usando l'operatore `&` su una variabile, o
- nel mezzo di un blocco assegnato,

è vietato. Tale errore di solito non verrà diagnosticato dal compilatore, ma condurrà l'esecuzione del programma in uno stato indefinito.

Esistono due strategie comuni per prevenire tali casi di comportamento non definito.

Il primo e preferibile è semplice: lo stesso `p` cessa di esistere quando non è più necessario, ad esempio:

```
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);
}
```

Chiamando `free()` direttamente prima della fine del blocco contenitore (cioè il `}`), `p` stesso cessa di esistere. Il compilatore darà un errore di compilazione su ogni tentativo di usare `p` dopo quello.

Un secondo approccio è anche quello di invalidare il puntatore stesso dopo aver rilasciato la memoria a cui punta:

```
free(p);
p = NULL;     // you may also use 0 instead of NULL
```

Argomenti per questo approccio:

- Su molte piattaforme, un tentativo di dereferenziare un puntatore nullo causerà un arresto anomalo istantaneo: errore di segmentazione. Qui, otteniamo almeno una traccia stack che punta alla variabile che è stata utilizzata dopo essere stata liberata.

  Senza impostare il puntatore su `NULL` abbiamo il puntatore che penzola. Molto probabilmente il programma si bloccherà ancora, ma in seguito, perché la memoria a cui punta il puntatore verrà automaticamente danneggiata. Tali bug sono difficili da rintracciare perché possono causare uno stack di chiamate completamente estraneo al problema iniziale.

  Questo approccio segue quindi il concetto di fail-fast .

- È sicuro liberare un puntatore nullo. Lo standard C specifica che `free(NULL)` non ha alcun effetto:

  > La funzione libera fa sì che lo spazio puntato da ptr sia deallocato, cioè reso disponibile per un'ulteriore allocazione. Se ptr è un puntatore nullo, non si verifica alcuna azione. Altrimenti, se l'argomento non corrisponde a un puntatore precedentemente restituito dalla funzione `calloc` , `malloc` o `realloc` , o se lo spazio è stato deallocato da una chiamata a `free` o `realloc` , il comportamento non è definito.

- A volte il primo approccio non può essere utilizzato (ad esempio, la memoria viene allocata in una funzione e rilasciata molto più tardi in una funzione completamente diversa)

Allocazione della memoria

Assegnazione standard

Le funzioni di allocazione della memoria dinamica C sono definite nell'intestazione `<stdlib.h>` . Se si desidera allocare lo spazio di memoria per un oggetto in modo dinamico, è possibile utilizzare il seguente codice:

```
int *p = malloc(10 * sizeof *p);
if (p == NULL)
{
    perror("malloc() failed");
    return -1;
}
```

Questo calcola il numero di byte che dieci `int` occupano in memoria, quindi richiede molti byte da `malloc` e assegna il risultato (cioè l'indirizzo iniziale del blocco di memoria appena creato usando `malloc`) a un puntatore denominato `p` .

È buona pratica usare `sizeof` per calcolare la quantità di memoria da richiedere poiché il risultato di `sizeof` è definito dall'implementazione (eccetto per i *tipi di carattere* , che sono `char` , `signed char` e `unsigned char` , per cui `sizeof` è definito per dare sempre `1`).

Poiché `malloc` potrebbe non essere in grado di soddisfare la richiesta, potrebbe restituire un puntatore nullo. È importante controllare questo per impedire tentativi successivi di dereferenziare il puntatore nullo.

La memoria allocata dinamicamente usando `malloc()` può essere ridimensionata usando `realloc()` o, quando non è più necessaria, rilasciata usando `free()` .

In alternativa, dichiarando `int array[10];` allocherebbe la stessa quantità di memoria. Tuttavia, se è dichiarato all'interno di una funzione senza la parola chiave `static` , sarà utilizzabile solo all'interno della funzione in cui è dichiarato e delle funzioni che chiama (poiché la matrice verrà allocata nello stack e lo spazio verrà rilasciato per il riutilizzo quando la funzione ritorna). In alternativa, se è definito con `static` all'interno di una funzione o se è definito al di fuori di qualsiasi funzione, la sua durata è la durata del programma. I puntatori possono anche essere restituiti da una funzione, tuttavia una funzione in C non può restituire un array.

Memoria azzerata

La memoria restituita da `malloc` potrebbe non essere inizializzata ad un valore ragionevole, e bisogna fare attenzione a azzerare la memoria con `memset` o copiare immediatamente un valore appropriato in essa. In alternativa, `calloc` restituisce un blocco della dimensione desiderata in cui tutti i bit sono inizializzati su `0` . Questo non deve essere uguale alla rappresentazione di zero virgola mobile o di una costante puntatore nullo.

```
int *p = calloc(10, sizeof *p);
if (p == NULL)
{
    perror("calloc() failed");
    return -1;
}
```

Una nota su `calloc` : la maggior parte delle implementazioni (comunemente usate) ottimizzerà `calloc()` per le prestazioni, quindi sarà più veloce di chiamare `malloc()` , quindi `memset()` , anche se l'effetto netto è identico.

Memoria allineata

C11

C11 ha introdotto una nuova funzione `aligned_alloc()` che alloca lo spazio con l'allineamento specificato. Può essere utilizzato se la memoria da allocare è necessaria per essere allineata a determinati limiti che non possono essere soddisfatti da `malloc()` o `calloc()` . `malloc()` funzioni `malloc()` e `calloc()` allocano memoria che è adeguatamente allineata per *qualsiasi* tipo di oggetto (cioè l'allineamento è `alignof(max_align_t)`). Ma con `aligned_alloc()` possono essere richiesti allineamenti maggiori.

```
/* Allocates 1024 bytes with 256 bytes alignment. */
char *ptr = aligned_alloc(256, 1024);
if (ptr) {
    perror("aligned_alloc()");
    return -1;
}
free(ptr);
```

Lo standard C11 impone due restrizioni: 1) la *dimensione* (secondo argomento) richiesta deve essere un multiplo integrale *dell'allineamento* (primo argomento) e 2) il valore *dell'allineamento* dovrebbe essere un allineamento valido supportato dall'implementazione. Il mancato rispetto di uno di questi risultati comporta un comportamento indefinito .

Riallocazione della memoria

Potrebbe essere necessario espandere o ridurre lo spazio di archiviazione del puntatore dopo aver allocato memoria ad esso. La funzione `void *realloc(void *ptr, size_t size)` rilascia il vecchio oggetto puntato da `ptr` e restituisce un puntatore a un oggetto che ha le dimensioni specificate dalla `size` . `ptr` è il puntatore a un blocco di memoria precedentemente assegnato con `malloc` , `calloc` o `realloc` (o un puntatore nullo) per essere riallocato. Il contenuto massimo possibile della memoria originale è conservato. Se la nuova dimensione è più grande, qualsiasi memoria aggiuntiva oltre la vecchia dimensione non è inizializzata. Se la nuova dimensione è più breve, il contenuto della parte ristretta viene perso. Se `ptr` è NULL, un nuovo blocco viene allocato e un puntatore ad esso viene restituito dalla funzione.

```
#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;
}
```

L'oggetto riallocato può avere o meno lo stesso indirizzo di `*p` . Pertanto è importante acquisire il valore di ritorno da `realloc` che contiene il nuovo indirizzo se la chiamata ha esito positivo.

Assicurati di assegnare il valore di ritorno di `realloc` ad un `temporary` invece dell'originale `p` . `realloc` restituirà null in caso di errore, che sovrascriverebbe il puntatore. Ciò perderebbe i tuoi dati e creerebbe una perdita di memoria.

Matrici multidimensionali di dimensioni variabili

C99

Poiché C99, C ha matrici di lunghezza variabile, VLA, quelle matrici di modelli con limiti noti solo al momento dell'inizializzazione. Mentre devi stare attento a non allocare VLA troppo grande (potrebbero distruggere il tuo stack), usare i *puntatori a VLA* e usarli in `sizeof` espressioni va bene.

```
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);
```

```
}
```

Qui `matrix` è un puntatore a elementi di tipo `double[m]` , e il `sizeof` espressione con `double[n][m]` assicura che contiene lo spazio per `n` tali elementi.

Tutto questo spazio è assegnato in modo contiguo e può quindi essere deallocato da una singola chiamata a `free` .

La presenza di VLA nella lingua influisce anche sulle possibili dichiarazioni di matrici e puntatori nelle intestazioni di funzione. Ora, è consentita un'espressione generale intera all'interno `[]` dei parametri dell'array. Per entrambe le funzioni le espressioni in `[]` usano i parametri che sono stati dichiarati prima nella lista dei parametri. Per `sumAll` queste sono le lunghezze che il codice utente si aspetta per la matrice. Come per tutti i parametri di funzione dell'array in C, la dimensione più interna viene riscritta su un tipo di puntatore, quindi questo è equivalente alla dichiarazione

```
  double sumAll(size_t n, size_t m, double (*A)[m]);
```

Cioè, `n` non fa realmente parte dell'interfaccia della funzione, ma le informazioni possono essere utili per la documentazione e potrebbero anche essere utilizzate dai limiti che controllano i compilatori per avvisare dell'accesso fuori dai limiti.

Probabilmente, per `main` , l'espressione `argc+1` è la lunghezza minima che lo standard C prescrive per l'argomento `argv` .

Si noti che ufficialmente il supporto VLA è facoltativo in C11, ma non conosciamo alcun compilatore che implementa C11 e che non li ha. Puoi testare con la macro `__STDC_NO_VLA__` se devi.

realloc (ptr, 0) non è equivalente a free (ptr)

`realloc` è *concettualmente equivalente* a `malloc + memcpy + free` sull'altro puntatore.

Se la dimensione dello spazio richiesto è zero, il comportamento di `realloc` è definito dall'implementazione. Questo è simile per tutte le funzioni di allocazione della memoria che ricevono un parametro di `size` del valore `0` . Tali funzioni possono infatti restituire un puntatore non nullo, ma non deve mai essere dereferenziato.

Quindi, `realloc(ptr,0)` non è equivalente a `free(ptr)` . Esso può

- essere un'implementazione "pigra" e solo tornare `ptr`
- `free(ptr)` , assegna un elemento fittizio e restituiscilo
- `free(ptr)` e restituisce `0`
- basta restituire `0` per errore e non fare nient'altro.

Quindi, in particolare, questi ultimi due casi non sono distinguibili dal codice dell'applicazione.

Ciò significa che `realloc(ptr,0)` potrebbe non essere realmente libero / deallocare la memoria, e quindi non dovrebbe mai essere usato come sostituto `free` .

Gestione della memoria definita dall'utente

`malloc()` chiama spesso funzioni del sistema operativo sottostante per ottenere pagine di memoria. Ma non c'è nulla di speciale nella funzione e può essere implementato in C diritta dichiarando una grande matrice statica e assegnandole da essa (c'è una leggera difficoltà nell'assicurare un corretto allineamento, in pratica l'allineamento a 8 byte è quasi sempre adeguato).

Per implementare uno schema semplice, un blocco di controllo viene memorizzato nell'area della memoria immediatamente prima del puntatore da restituire dalla chiamata. Ciò significa che `free()` può essere implementato sottraendo dal puntatore restituito e leggendo le informazioni di controllo, che è tipicamente la dimensione del blocco più alcune informazioni che gli consentono di essere rimesso nella lista libera - un elenco collegato di blocchi non allocati.

Quando l'utente richiede un'allocazione, la ricerca della lista libera viene eseguita fino a quando non viene trovato un blocco di dimensioni identiche o maggiori dell'importo richiesto, quindi se necessario viene diviso. Questo può portare alla frammentazione della memoria se l'utente sta facendo continuamente molte allocazioni e libera da dimensioni imprevedibili e ad intervalli imprevedibili (non tutti i programmi reali si comportano in questo modo, lo schema semplice è spesso adeguato per i piccoli programmi).

```
/* 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;
```

Molti programmi richiedono un numero elevato di allocazioni di piccoli oggetti della stessa dimensione. Questo è molto facile da implementare. Basta usare un blocco con un puntatore successivo. Quindi se è richiesto un blocco di 32 byte:

```
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;
}
```

Questo schema è estremamente veloce ed efficiente e può essere reso generico con una certa perdita di chiarezza.

alloca: alloca la memoria sullo stack

`alloca` **: l'** `alloca` è menzionata solo qui per motivi di completezza. È interamente non portatile (non coperto da nessuno degli standard comuni) e ha un numero di funzioni potenzialmente pericolose che lo rendono non sicuro per l'inconsapevole. Il codice C moderno dovrebbe sostituirlo con *Array a lunghezza variabile* (VLA).

Pagina manuale

```
#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
}
```

Assegna memoria al frame dello stack del chiamante, lo spazio a cui fa riferimento il puntatore restituito viene liberato automaticamente al termine della funzione chiamante.

Sebbene questa funzione sia utile per la gestione automatica della memoria, tenere presente che la richiesta di allocazioni elevate potrebbe causare un sovraccarico dello stack e che non è possibile utilizzare `free` con la memoria allocata con `alloca` (che potrebbe causare più problemi con l'overflow dello stack).

Per questo motivo non è raccomandato l'uso di `alloca` all'interno di un ciclo né una funzione ricorsiva.

E poiché la memoria è `free` 'su funzione restituita non è possibile restituire il puntatore come risultato di una funzione (il comportamento sarebbe indefinito).

Sommario

- chiamare identico a `malloc`
- automaticamente liberato al ritorno della funzione
- incompatibile con le funzioni `free` , `realloc` (comportamento non definito)
- il puntatore non può essere restituito come risultato di una funzione (comportamento non definito)
- dimensione di allocazione limitata dallo spazio di stack, che (sulla maggior parte delle macchine) è molto più piccola dello spazio heap disponibile per l'uso da `malloc()`
- evitare l'uso di `alloca()` e VLA (array di lunghezza variabile) in una singola funzione
- `alloca()` non è così portatile come `malloc()` et al

Raccomandazione

- Non utilizzare `alloca()` nel nuovo codice

C99

Alternativa moderna

```
void foo(int size) {
    char data[size];
    /*
      function body;
    */
    // data is automatically freed
}
```

Funziona dove `alloca()` fa e funziona in luoghi in cui non è `alloca()` (loop interni, ad esempio). Presuppone un'implementazione C99 o un'implementazione C11 che non definisce `__STDC_NO_VLA__` .

Leggi Gestione della memoria online: https://riptutorial.com/it/c/topic/4726/gestione-della-memoria

Capitolo 30: I bit-field

introduzione

La maggior parte delle variabili in C ha una dimensione che è un numero intero di byte. I campi di bit sono una parte di una struttura che non occupa necessariamente un numero intero di byte; possono un numero qualsiasi di bit. Più campi di bit possono essere raggruppati in un'unica unità di memoria. Fanno parte dello standard C, ma ci sono molti aspetti che sono definiti dall'implementazione. Sono una delle parti meno trasportabili di C.

Sintassi

- identificatore del tipo identificativo: dimensione;

Parametri

Parametro	Descrizione
tipo-specificatore	`signed` , `unsigned` , `int` o `_Bool`
identificatore	Il nome per questo campo nella struttura
taglia	Il numero di bit da utilizzare per questo campo

Osservazioni

Gli unici tipi portatili per campi bit sono `signed` , `unsigned` o `_Bool` . Il tipo plain `int` può essere usato, ma lo standard dice (§6.7.2¶5) *... per i campi bit, è definito dall'implementazione se lo specificatore `int` designa lo stesso tipo di `signed int` o lo stesso tipo di `unsigned int` .*

Altri tipi di interi possono essere consentiti da un'implementazione specifica, ma il loro utilizzo non è portabile.

Examples

I bit-field

Un semplice campo di bit può essere usato per descrivere cose che possono avere un numero specifico di bit coinvolti.

```
struct encoderPosition {
   unsigned int encoderCounts : 23;
   unsigned int encoderTurns  : 4;
   unsigned int _reserved     : 5;
```

```
};
```

In questo esempio consideriamo un codificatore con 23 bit di precisione singola e 4 bit per descrivere il multigiro. I bit-field vengono spesso utilizzati quando si interfaccia con l'hardware che emette dati associati a un numero specifico di bit. Un altro esempio potrebbe essere la comunicazione con un FPGA, in cui l'FPGA scrive i dati nella memoria in sezioni a 32 bit consentendo letture hardware:

```
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;
   };
};
```

Per questo esempio abbiamo mostrato un costrutto comunemente usato per poter accedere ai dati nei suoi singoli bit, o per scrivere il pacchetto di dati nel suo complesso (emulare quello che potrebbe fare l'FPGA). Potremmo quindi accedere ai bit in questo modo:

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

Questo è valido, ma secondo lo standard C99 6.7.2.1, elemento 10:

> L'ordine di assegnazione dei campi di bit all'interno di un'unità (ordine alto a basso ordine o basso ordine a alto ordine) è definito dall'implementazione.

È necessario essere consapevoli di endianness quando si definiscono campi di bit in questo modo. Come tale può essere necessario utilizzare una direttiva preprocessore per verificare la endianità della macchina. Un esempio di questo segue:

```
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;
```

Utilizzo di campi bit come piccoli numeri interi

```
#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;
}
```

Allineamento del campo di bit

I campi di bit danno la possibilità di dichiarare campi di struttura che sono più piccoli della larghezza del carattere. I campi di bit vengono implementati con maschera a livello di byte o parola. L'esempio seguente produce una struttura di 8 byte.

```
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. */
};
```

I commenti descrivono un possibile layout, ma poiché lo standard dice che *l'allineamento dell'unità di memoria indirizzabile non è specificato* , sono possibili anche altri layout.

Un campo di bit senza nome può essere di qualsiasi dimensione, ma non può essere inizializzato o referenziato.

A un campo di bit a larghezza zero non può essere assegnato un nome e allinea il campo successivo al limite definito dal tipo di dati del campo di bit. Questo si ottiene tamponando i bit tra i campi di bit.

La dimensione della struttura 'A' è 1 byte.

```
struct A
```

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

Nella struttura B, il primo campo di bit senza nome salta 2 bit; il campo di bit a larghezza zero dopo `c2` fa sì che `c3` inizi dal limite del char (quindi vengono saltati 3 bit tra `c2` e `c3` . Dopo il `c4` ci sono 3 bit di padding, quindi la dimensione della struttura è 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;
};
```

Quando sono utili i campi bit?

Un campo di bit viene utilizzato per raggruppare più variabili in un unico oggetto, simile a una struttura. Ciò consente un utilizzo ridotto della memoria ed è particolarmente utile in un ambiente embedded.

```
 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
```

Se dichiariamo queste variabili separatamente, allora ognuno deve essere almeno un numero intero a 8 bit e lo spazio totale richiesto sarà 5 byte. Inoltre le variabili non utilizzeranno l'intero intervallo di un intero senza segno a 8 bit (0-255). Qui possiamo usare campi di bit.

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

I campi di bit nella struttura sono accessibili come qualsiasi altra struttura. Il programmatore deve fare attenzione che le variabili siano scritte nell'intervallo. Se fuori dal range il comportamento non è definito.

```
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;
}
```

Spesso il programmatore vuole azzerare il set di campi di bit. Questo può essere fatto elemento per elemento, ma esiste un secondo metodo. Crea semplicemente un'unione della struttura sopra con un tipo senza segno che è maggiore o uguale alla dimensione della struttura. Quindi l'intero set di campi di bit può essere azzerato azzerando questo intero senza segno.

```
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;
```

L'utilizzo è il seguente

```
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;
}
```

In conclusione, i campi di bit sono comunemente usati in situazioni con vincoli di memoria in cui si hanno molte variabili che possono assumere intervalli limitati.

Non fare per i campi di bit

1. Matrici di campi di bit, puntatori a campi di bit e funzioni che restituiscono campi di bit non sono consentiti.
2. L'operatore dell'indirizzo (&) non può essere applicato ai membri del campo di bit.
3. Il tipo di dati di un campo di bit deve essere sufficientemente ampio da contenere la dimensione del campo.
4. L'operatore `sizeof()` non può essere applicato a un campo di bit.
5. Non è possibile creare un `typedef` per un campo di bit in isolamento (sebbene sia possibile creare un `typedef` per una struttura contenente campi di bit).

```
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 */
}
```

Leggi I bit-field online: https://riptutorial.com/it/c/topic/1930/i-bit-field

Capitolo 31: Idiomi di programmazione C comuni e pratiche di sviluppo

Examples

Confronto letterale e variabile

Supponiamo che tu stia confrontando il valore con alcune variabili

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

Supponiamo ora di aver sbagliato `==` con `=` . Allora ci vorrà del tuo dolce momento per capirlo.

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

Quindi, se un segno di uguale viene accidentalmente omesso, il compilatore lamenterà un "tentativo di assegnazione al letterale". Questo non ti proteggerà quando confronti due variabili, ma ogni piccolo aiuto è utile.

Vedi qui per maggiori informazioni.

Non lasciare vuoto l'elenco dei parametri di una funzione - usa nullo

Supponiamo che tu stia creando una funzione che non richiede argomenti quando viene chiamata e ti trovi di fronte al dilemma su come dovresti definire la lista dei parametri nel prototipo della funzione e nella definizione della funzione.

- Hai la possibilità di mantenere l'elenco dei parametri vuoto sia per il prototipo che per la definizione. In tal modo, assomigliano all'istruzione di chiamata della funzione di cui avrete bisogno.
- Si legge da qualche parte che uno degli usi della parola chiave ***void*** (ce ne sono solo alcuni), consiste nel definire l'elenco dei parametri delle funzioni che non accettano alcun argomento nella propria chiamata. Quindi, questa è anche una scelta.

Quindi, qual è la scelta giusta?

RISPOSTA: utilizzando la parola chiave ***void***

CONSIGLI GENERALI: Se una lingua fornisce determinate funzionalità da utilizzare per uno

scopo speciale, è meglio utilizzarlo nel codice. Ad esempio, utilizzando `enum` s anziché `#define` macro (che è per un altro esempio).

C11 sezione 6.7.6.3 "Dichiaratori di funzione", paragrafo 10, recita:

> Il caso speciale di un parametro senza nome di tipo void come unica voce nell'elenco specifica che la funzione non ha parametri.

Il paragrafo 14 di quella stessa sezione mostra l'unica differenza:

> ... Una lista vuota in un dichiaratore di funzione che fa parte di una definizione di quella funzione specifica che la funzione non ha parametri. L'elenco vuoto in un dichiaratore di funzioni che non fa parte di una definizione di tale funzione specifica che non viene fornita alcuna informazione sul numero o sui tipi dei parametri.

Una spiegazione semplificata fornita da K & R (pagg. 72-73) per le cose di cui sopra:

> Inoltre, se una dichiarazione di funzione non include argomenti, come in `double atof();` , anche questo è inteso nel senso che nulla deve essere assunto riguardo agli argomenti di `atof` ; tutto il controllo dei parametri è disattivato. Questo speciale significato dell'elenco di argomenti vuoti ha lo scopo di consentire ai vecchi programmi in C di compilare con nuovi compilatori. Ma è una cattiva idea usarlo con nuovi programmi. Se la funzione accetta argomenti, dichiarali; se non accetta argomenti, usa `void` .

Ecco come dovrebbe apparire il prototipo della tua funzione:

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

Ed ecco come dovrebbe essere la definizione della funzione:

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

Un vantaggio nell'usare il suddetto tipo di dichiarazione `int foo()` (cioè senza utilizzare la parola chiave ***void***), è che il compilatore può rilevare l'errore se si chiama la propria funzione usando un'istruzione errata come `foo(42)` . Questo tipo di istruzione di chiamata a una funzione non causerebbe errori se si lascia vuota l'elenco dei parametri. L'errore passerebbe silenziosamente, inosservato e il codice continuerebbe a essere eseguito.

Questo significa anche che dovresti definire la funzione `main()` questo modo:

```
int main(void)
{
    ...
```

```
    <statements>
    ...
    return 0;
}
```

Si noti che anche se una funzione definita con un elenco di parametri vuoto non accetta argomenti, non fornisce un prototipo per la funzione, quindi il compilatore non si lamenterà se la funzione viene successivamente chiamata con argomenti. Per esempio:

```
#include <stdio.h>

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

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

Se quel codice è salvato nel file `proto79.c` , può essere compilato su Unix con GCC (versione 7.1.0 su macOS Sierra 10.12.5 usato per la dimostrazione) in questo modo:

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

Se compili con opzioni più stringenti, ottieni degli errori:

```
$ 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
$
```

Se assegni alla funzione il prototipo formale `static void parameterless(void)` , la compilazione restituisce errori:

```
$ 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)
             ^~~~~~~~~~~~
$
```

Morale: assicurati sempre di avere dei prototipi e assicurati che il compilatore ti dica quando non stai obbedendo alle regole.

Leggi Idiomi di programmazione C comuni e pratiche di sviluppo online:
https://riptutorial.com/it/c/topic/10543/idiomi-di-programmazione-c-comuni-e-pratiche-di-sviluppo

Capitolo 32: Imbottitura e imballaggio della struttura

introduzione

Per impostazione predefinita, i compilatori C dispongono di strutture in modo che sia possibile accedere rapidamente a ciascun membro, senza incorrere in penalità per l'accesso non allineato, un problema con le macchine RISC come DEC Alpha e alcune CPU ARM.

A seconda dell'architettura della CPU e del compilatore, una struttura può occupare più spazio in memoria della somma delle dimensioni dei membri del componente. Il compilatore può aggiungere padding tra membri o alla fine della struttura, ma non all'inizio.

L'imballaggio sovrascrive il riempimento predefinito.

Osservazioni

Eric Raymond ha un articolo su The Lost Art of C Structure Packing che è una lettura utile.

Examples

Strutture di imballaggio

Di default le strutture sono riempite in C. Se vuoi evitare questo comportamento, devi richiederlo esplicitamente. Sotto GCC è `__attribute__((__packed__))` . Considera questo esempio su una macchina a 64 bit:

```
struct foo {
    char *p;  /* 8 bytes */
    char c;   /* 1 byte  */
    long x;   /* 8 bytes */
};
```

La struttura verrà automaticamente riempita per avere `8-byte` allineamento di `8-byte` e avrà il seguente aspetto:

```
struct foo {
    char *p;     /* 8 bytes */
    char c;      /* 1 byte  */

    char pad[7]; /* 7 bytes added by compiler */

    long x;      /* 8 bytes */
};
```

Quindi `sizeof(struct foo)` ci darà `24` invece di `17` . Questo è accaduto a causa di un compilatore a

64 bit di lettura / scrittura da / a memoria in 8 byte di parola in ogni passaggio e ovvio quando si tenta di scrivere `char c;` un byte in memoria un intero 8 byte (cioè una parola) recuperato e consuma solo il primo byte di esso e i suoi sette successivi di byte rimane vuoto e non accessibile per qualsiasi operazione di lettura e scrittura per il riempimento della struttura.

Imballaggio della struttura

Ma se aggiungi l'attributo `packed` , il compilatore non aggiungerà il padding:

```
struct __attribute__((__packed__)) foo {
    char *p;  /* 8 bytes */
    char c;   /* 1 byte  */
    long x;   /* 8 bytes */
};
```

Ora `sizeof(struct foo)` restituirà `17` .

Vengono utilizzate strutture generalmente imballate:

- Per risparmiare spazio
- Formattare una struttura dati per trasmettere su rete senza dipendere da ciascun allineamento dell'architettura di ciascun nodo della rete.

È necessario tenere presente che alcuni processori come ARM Cortex-M0 non consentono l'accesso alla memoria non allineato; in questi casi, l'impacchettamento della struttura può comportare un *comportamento indefinito* e causare il blocco della CPU.

Imbottitura della struttura

Supponiamo che questa `struct` sia definita e compilata con un compilatore a 32 bit:

```
struct test_32 {
    int a;      // 4 byte
    short b;    // 2 byte
    int c;      // 4 byte
} str_32;
```

Potremmo aspettarci che questa `struct` occupi solo 10 byte di memoria, ma stampando `sizeof(str_32)` vediamo che usa 12 byte.

Ciò è accaduto perché il compilatore allinea le variabili per l'accesso rapido. Un modello comune è che quando il tipo di base occupa N byte (dove N è una potenza di 2 come 1, 2, 4, 8, 16 e raramente più grande), la variabile deve essere allineata su un limite di N byte (un multiplo di N byte).

Per la struttura mostrata con `sizeof(int) == 4` e `sizeof(short) == 2` , un layout comune è:

- `int a;` memorizzato all'offset 0; taglia 4.
- `short b;` memorizzato all'offset 4; taglia 2.

- imbottitura senza nome all'offset 6; taglia 2.
- `int c;` memorizzato all'offset 8; taglia 4.

Quindi `struct test_32` occupa 12 byte di memoria. In questo esempio, non vi è alcun riempimento finale.

Il compilatore garantirà che tutte le variabili `struct test_32` siano memorizzate a partire da un limite di 4 byte, in modo che i membri all'interno della struttura siano correttamente allineati per l'accesso rapido. Le funzioni di allocazione della memoria come `malloc()` , `calloc()` e `realloc()` sono necessarie per garantire che il puntatore restituito sia sufficientemente allineato per l'uso con qualsiasi tipo di dati, quindi anche le strutture allocate dinamicamente saranno allineate correttamente.

È possibile ritrovarsi con situazioni strane come su un processore Intel x86_64 a 64 bit (ad es. Intel Core i7 - un Mac con MacOS Sierra o Mac OS X), dove quando si compila in modalità a 32 bit, i compilatori vengono posizionati in modo `double` su un Limite di 4 byte; ma, sullo stesso hardware, durante la compilazione in modalità a 64 bit, i compilatori vengono posizionati in modo `double` su un limite di 8 byte.

Leggi Imbottitura e imballaggio della struttura online:
https://riptutorial.com/it/c/topic/4590/imbottitura-e-imballaggio-della-struttura

Capitolo 33: Inizializzazione

Examples

Inizializzazione di variabili in C

In assenza di inizializzazione esplicita, le variabili esterne e `static` sono garantite per essere inizializzate a zero; le variabili automatiche (incluse `register` variabili di `register`) hanno valori iniziali *indeterminati* [1] (cioè, garbage).

Le variabili scalari possono essere inizializzate quando vengono definite seguendo il nome con un segno di uguale e un'espressione:

```
int x = 1;
char squota = '\'';
long day = 1000L * 60L * 60L * 24L; /* milliseconds/day */
```

Per le variabili esterne e `static` , l'inizializzatore deve essere *un'espressione costante* [2] ; l'inizializzazione viene eseguita una volta, concettualmente prima che il programma inizi l'esecuzione.

Per `register` variabili automatiche e di `register` , l'inizializzatore non si limita ad essere una costante: può essere qualsiasi espressione che coinvolge valori precedentemente definiti, anche chiamate di funzione.

Ad esempio, vedi lo snippet di codice qui sotto

```
int binsearch(int x, int v[], int n)
{
   int low = 0;
   int high = n - 1;
   int mid;
   ...
}
```

invece di

```
   int low, high, mid;

   low = 0;
   high = n - 1;
```

In effetti, l'inizializzazione delle variabili automatiche è solo una scorciatoia per le istruzioni di assegnazione. Quale forma preferire è in gran parte una questione di gusti. Generalmente utilizziamo assegnazioni esplicite, perché gli inizializzatori nelle dichiarazioni sono più difficili da vedere e più lontano dal punto di utilizzo. D'altra parte, le variabili dovrebbero essere dichiarate solo quando stanno per essere utilizzate, quando possibile.

Inizializzazione di un array:

Un array può essere inizializzato seguendo la sua dichiarazione con un elenco di inizializzatori racchiusi tra parentesi e separati da virgole.

Ad esempio, per inizializzare i giorni di un array con il numero di giorni in ciascun mese:

```
int days_of_month[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
```

(Si noti che gennaio è codificato come mese zero in questa struttura.)

Quando la dimensione dell'array viene omessa, il compilatore calcolerà la lunghezza contando gli inizializzatori, di cui ci sono 12 in questo caso.

Se ci sono meno inizializzatori per una matrice rispetto alla dimensione specificata, gli altri saranno zero per tutti i tipi di variabili.

È un errore avere troppi inizializzatori. Non esiste un modo standard per specificare la ripetizione di un inizializzatore, ma GCC ha un'estensione per farlo.

C99

In C89 / C90 o versioni precedenti di C, non era possibile inizializzare un elemento nel mezzo di un array senza fornire tutti i valori precedenti.

C99

Con C99 e versioni successive, gli inizializzatori designati consentono di inizializzare elementi arbitrari di un array, lasciando come valori zero tutti i valori non inizializzati.

Inizializzazione degli array di caratteri:

Gli array di caratteri sono un caso speciale di inizializzazione; una stringa può essere usata al posto della notazione di parentesi graffe e virgole:

```
char chr_array[] = "hello";
```

è una scorciatoia per il più lungo ma equivalente:

```
char chr_array[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

In questo caso, la dimensione dell'array è sei (cinque caratteri più il terminante `'\0'`).

[1] Cosa succede a una variabile dichiarata non inizializzata in C? Ha un valore?

[2] Si noti che *un'espressione costante* è definita come qualcosa che può essere valutata in fase di compilazione. Quindi, `int global_var = f();` è invalido. Un altro equivoco comune è pensare a una variabile qualificata `const` come *espressione costante* . In C, `const` significa "sola lettura", non "costante di tempo di compilazione". Quindi, definizioni globali come `const int SIZE = 10; int`

`global_arr[SIZE];` e `const int SIZE = 10; int global_var = SIZE;` non sono legali in C.

Inizializzazione di strutture e matrici di strutture

Le strutture e le matrici di strutture possono essere inizializzate da una serie di valori racchiusi tra parentesi graffe, un valore per membro della struttura.

```
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
};
```

Si noti che l'inizializzazione dell'array potrebbe essere scritta senza le parentesi interne e in passato (prima del 1990, diciamo) spesso sarebbe stata scritta senza di essi:

```
struct Date uk_battles[] =
{
    1066, 10, 14,    // Battle of Hastings
    1815,  6, 18,    // Battle of Waterloo
    1805, 10, 21,    // Battle of Trafalgar
};
```

Anche se questo funziona, non è un buon stile moderno - non dovresti provare a usare questa notazione nel nuovo codice e dovresti correggere gli avvertimenti del compilatore che di solito produce.

Vedi anche gli inizializzatori designati .

Utilizzando inizializzatori designati

C99

C99 ha introdotto il concetto di *inizializzatori designati.* Questi ti permettono di specificare quali elementi di una matrice, struttura o unione devono essere inizializzati dai valori seguenti.

Inizializzatori designati per elementi array

Per un tipo semplice come plain `int` :

```
int array[] = { [4] = 29, [5] = 31, [17] = 101, [18] = 103, [19] = 107, [20] = 109 };
```

Il termine tra parentesi quadre, che può essere qualsiasi espressione intera costante, specifica quale elemento della matrice deve essere inizializzato dal valore del termine dopo il segno `=` . Gli elementi non specificati sono inizializzati di default, il che significa che gli zeri sono definiti. L'esempio mostra gli inizializzatori designati in ordine; non devono essere in ordine. L'esempio mostra lacune; quelli sono legittimi. L'esempio non mostra due diverse inizializzazioni per lo stesso elemento; anche questo è permesso (ISO / IEC 9899: 2011, §6.7.9 Inizializzazione, ¶19 *L'inizializzazione deve avvenire in ordine di lista inizializzatore, ogni inizializzatore fornito per un particolare subobject che sovrascrive qualsiasi inizializzatore precedentemente elencato per lo stesso subobject*).

In questo esempio, la dimensione dell'array non è definita esplicitamente, quindi l'indice massimo specificato negli inizializzatori designati determina la dimensione dell'array, che sarebbe 21 elementi nell'esempio. Se la dimensione è stata definita, l'inizializzazione di una voce oltre la fine dell'array sarebbe un errore, come al solito.

Inizializzatori designati per strutture

È possibile specificare quali elementi di una struttura vengono inizializzati utilizzando `.` notazione `element` :

```
struct Date
{
    int year;
    int month;
    int day;
};

struct Date us_independence_day = { .day = 4, .month = 7, .year = 1776 };
```

Se gli elementi non sono elencati, vengono inizializzati automaticamente (azzerati).

Inizializzatore designato per i sindacati

È possibile specificare quale elemento di unione viene inizializzato con un inizializzatore designato.

C89

Prima dello standard C, non c'era modo di inizializzare un `union` . Lo standard C89 / C90 consente di inizializzare il primo membro di un `union` , quindi la scelta di quale membro è elencato per prima cosa è importante.

```
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

Tieni presente che C11 ti consente di utilizzare membri di un sindacato anonimo all'interno di una struttura, in modo da non aver bisogno del nome `du` nell'esempio precedente:

```
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 };
```

Inizializzatori designati per array di strutture, ecc

Questi costrutti possono essere combinati per matrici di strutture contenenti elementi che sono matrici, ecc. L'uso di serie complete di parentesi assicura che la notazione sia non ambigua.

```
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",
          }
 };
```

Specifica degli intervalli negli inizializzatori di array

GCC fornisce un'estensione che consente di specificare un intervallo di elementi in una matrice a cui deve essere assegnato lo stesso inizializzatore:

```
int array[] = { [3 ... 7] = 29, 19 = 107 };
```

I punti tripli devono essere separati dai numeri per timore che uno dei punti sia interpretato come parte di un numero in virgola mobile (regola *massima di munch*).

Leggi Inizializzazione online: https://riptutorial.com/it/c/topic/4547/inizializzazione

Capitolo 34: inlining

Examples

Funzioni di allineamento utilizzate in più di un file sorgente

Per le piccole funzioni che vengono richiamate spesso, l'overhead associato alla chiamata di funzione può essere una frazione significativa del tempo di esecuzione totale di tale funzione. Un modo per migliorare le prestazioni, quindi, è eliminare il sovraccarico.

In questo esempio utilizziamo quattro funzioni (più `main()`) in tre file sorgente. Due di questi (`plusfive()` e `timestwo()`) vengono richiamati dagli altri due situati in "source1.c" e "source2.c". Il `main()` è incluso quindi abbiamo un esempio funzionante.

main.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;
}
```

headerfile.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
```

Le funzioni `timestwo` e `plusfive` vengono richiamate sia da `complicated1` sia da `complicated2` , che si trovano in "unità di traduzione" o file sorgente diversi. Per utilizzarli in questo modo, dobbiamo definirli nell'intestazione.

Compilare in questo modo, assumendo 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
```

Utilizziamo l'opzione di ottimizzazione -O2 perché alcuni compilatori non sono in linea senza l'ottimizzazione attivata.

L'effetto della parola chiave `inline` è che il simbolo della funzione in questione non viene emesso nel file oggetto. Altrimenti si verificherà un errore nell'ultima riga, dove stiamo collegando i file oggetto per formare l'eseguibile finale. Se non avessimo in `inline` , lo stesso simbolo verrebbe definito in entrambi i file `.o` e si verificherebbe un errore "simbolo definito in più punti".

In situazioni in cui il simbolo è effettivamente necessario, questo ha lo svantaggio che il simbolo non viene prodotto affatto. Ci sono due possibilità per affrontarlo. Il primo è aggiungere una dichiarazione `extern` extra delle funzioni inline esattamente in uno dei file `.c` . Quindi aggiungi quanto segue a `source1.c` :

```
extern int timestwo(int input);
extern int plusfive(int input);
```

L'altra possibilità è definire la funzione con `static inline` anziché `inline` . Questo metodo ha l'inconveniente che alla fine una copia della funzione in questione può essere prodotta in **ogni** file oggetto prodotto con questa intestazione.

Leggi inlining online: https://riptutorial.com/it/c/topic/7427/inlining

Capitolo 35: Input / Output formattato

Examples

Stampa del valore di un puntatore su un oggetto

Per stampare il valore di un puntatore su un oggetto (al contrario di un puntatore di funzione), utilizzare lo specificatore di conversione `p` . È stato definito per stampare solo i segnapunti `void` , quindi per stampare il valore di un punto non `void` deve essere convertito in modo esplicito ("castato *") per `void*` .

```
#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

Usando `<inttypes.h>` e `uintptr_t`

Un altro modo per stampare i puntatori in C99 o `uintptr_t` successive utilizza il tipo `uintptr_t` e le macro da `<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;
}
```

In teoria, potrebbe non esserci un tipo intero che può contenere qualsiasi puntatore convertito in un numero intero (quindi il tipo `uintptr_t` potrebbe non esistere). In pratica, esiste. I puntatori alle funzioni non devono essere convertibili nel tipo `uintptr_t` , anche se spesso sono più convertibili.

Se il tipo `uintptr_t` esiste, lo stesso `intptr_t` tipo `intptr_t` . Non è chiaro il motivo per cui si vorrebbe mai trattare gli indirizzi come interi con segno, però.

Storia pre-standard:

Prima di C89 durante i tempi di K & R-C non c'era nessun tipo `void*` (né header `<stdlib.h>` , né prototipi, e quindi nessuna notazione `int main(void)`), quindi il puntatore è stato castato su `long unsigned int` e stampato usando il modificatore di lunghezza / identificatore di conversione `lx` .

L'esempio sotto è solo a scopo informativo. Oggigiorno questo è un codice non valido, che molto bene potrebbe provocare il famigerato comportamento indefinito .

```
#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;
}
```

Stampa della differenza dei valori di due puntatori su un oggetto

Sottraendo i valori di due puntatori a un oggetto si ottiene un numero intero con segno * 1 . Quindi verrebbe stampato utilizzando *almeno* lo specifier di conversione `d` .

Per assicurarsi che ci sia un tipo abbastanza largo da contenere una "differenza puntatore", dato che C99 `<stddef.h>` definisce il tipo `ptrdiff_t` . Per stampare un `ptrdiff_t` usa il modificatore di lunghezza `t` .

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;
}
```

Il risultato potrebbe essere simile a questo:

```
p1 = 0x7fff6679f430
p2 = 0x7fff6679f434
p2 - p1 = 1
```

Si noti che il valore risultante della differenza viene ridimensionato in base alla dimensione del puntatore sottratto dal puntatore, un `int` qui. La dimensione di un `int` per questo esempio è 4.

---

* 1 Se i due puntatori da sottrarre non puntano allo stesso oggetto, il comportamento non è definito.

Specifiers di conversione per la stampa

Identificatore di conversione	Tipo di argomento	Descrizione
`i` , `d`	int	stampa decimale
`u`	int non firmato	stampa decimale
`o`	int non firmato	stampa ottale
`x`	int non firmato	stampa esadecimale, minuscolo
`X`	int non firmato	stampa esadecimale, maiuscolo
`f`	Doppio	stampa fluttuante con una precisione predefinita di 6, se non viene fornita alcuna precisione (minuscole utilizzate per numeri speciali `nan` e `inf` o `infinity`)
`F`	Doppio	stampa fluttuante con una precisione predefinita di 6, se non viene fornita alcuna precisione (maiuscole utilizzate per numeri speciali `NAN` e `INF` o `INFINITY`)
`e`	Doppio	le stampe fluttuano con una precisione predefinita di 6, se non viene data alcuna precisione, usando la notazione scientifica usando mantissa / esponente; esponente minuscolo e numeri speciali
`E`	Doppio	le stampe fluttuano con una precisione predefinita di 6, se non viene data alcuna precisione, usando la notazione scientifica usando mantissa / esponente; esponente maiuscolo e numeri speciali
`g`	Doppio	utilizza o `f` o `e` *[vedi sotto]*
`G`	Doppio	usa `F` o `E` *[vedi sotto]*
`a`	Doppio	stampa esadecimale, minuscolo

Identificatore di conversione	Tipo di argomento	Descrizione
`A`	Doppio	stampa esadecimale, maiuscolo
`c`	carbonizzare	stampa carattere singolo
`s`	char *	stampa una stringa di caratteri fino a un terminatore `NUL` o troncato a lunghezza data dalla precisione, se specificato
`p`	void *	stampa `void` -pointer value; un `void` non `void` dovrebbe essere convertito esplicitamente ("cast") a `void*` ; puntatore all'oggetto solo, **non a** un puntatore di funzione
`%`	n / A	stampa il carattere `%`
`n`	int *	***scrivere*** il numero di byte stampati finora nel `int` indicò.

Si noti che i modificatori di lunghezza possono essere applicati a `%n` (ad esempio `%hhn` indica che *un* `%hhn` *conversione* `n` *successivo si applica a un puntatore a un argomento* `signed char` , in base alla ISO / IEC 9899: 2011 §7.21.6.1 ¶7).

Si noti che le conversioni in virgola mobile si applicano ai tipi `float` e `double` causa delle regole di promozione predefinite - §6.5.2.2 Chiamate di funzione, ¶ 7 *La notazione dei puntini di sospensione in un dichiaratore di prototipo di funzione causa l'interruzione della conversione del tipo di argomento dopo l'ultimo parametro dichiarato. Le promozioni degli argomenti predefiniti vengono eseguite sugli argomenti finali.* Quindi, funzioni come `printf()` sono sempre passate solo `double` valori, anche se la variabile di riferimento è di tipo `float` .

Con i formati `g` e `G` , la scelta tra la notazione `e` ed `f` (o `E` e `F`) è documentata nello standard C e nelle specifiche POSIX per `printf()` :

> Il doppio argomento che rappresenta un numero in virgola mobile deve essere convertito nello stile `f` o `e` (o nello stile `F` o `E` nel caso di un `G` conversione `G`), a seconda del valore convertito e della precisione. Sia `P` la precisione se non zero, 6 se la precisione è omessa, o 1 se la precisione è zero. Quindi, se una conversione con lo stile `E` avrebbe un esponente di `X` :
>
> - Se P> X> = -4, la conversione deve essere con lo stile `f` (o `F`) e con precisione `P - (X+1)` .
> - Altrimenti, la conversione deve essere con stile `e` (o `E`) e precisione `P - 1` .
>
> Infine, a meno che non sia usato il contrassegno "#", tutti gli zeri finali devono essere rimossi dalla parte frazionaria del risultato e il carattere del punto decimale deve essere rimosso se non è rimasta alcuna parte frazionaria.

La funzione printf ()

Acceduto tramite l'inclusione di `<stdio.h>` , la funzione `printf()` è lo strumento principale utilizzato per stampare il testo sulla console in C.

```
printf("Hello world!");
// Hello world!
```

Gli array di caratteri normali e non formattati possono essere stampati da soli posizionandoli direttamente tra le parentesi.

```
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
```

In alternativa, è possibile stampare numeri interi, numeri in virgola mobile, caratteri e altro utilizzando il carattere di escape `%` , seguito da un carattere o sequenza di caratteri che denotano il formato, noto come *specificatore di formato* .

Tutti gli argomenti aggiuntivi alla funzione `printf()` sono separati da virgole e questi argomenti dovrebbero essere nello stesso ordine degli specificatori di formato. Argomenti aggiuntivi vengono ignorati, mentre gli argomenti digitati in modo errato o la mancanza di argomenti causano errori o comportamenti non definiti. Ogni argomento può essere un valore letterale o una variabile.

Al termine dell'esecuzione, il numero di caratteri stampati viene restituito con tipo `int` . In caso contrario, un errore restituisce un valore negativo.

Modificatori di lunghezza

Gli standard C99 e C11 specificano i seguenti modificatori di lunghezza per `printf()` ; i loro significati sono:

Modificatore	Modifica	Si applica a
hh	d, i, o, u, x o X	`char` , `char signed char` o `unsigned char`
h	d, i, o, u, x o X	`short int` o `unsigned short int`
l	d, i, o, u, x o X	`long int` o `unsigned long int`
l	a, A, e, E, f, F, g o G	`double` (per compatibilità con `scanf()` ; indefinito in C90)
ll	d, i, o, u, x o X	`long long int` o `unsigned long long int`
j	d, i, o, u, x o X	`intmax_t` o `uintmax_t`

Modificatore	Modifica	Si applica a
z	d, i, o, u, x o X	`size_t` o il tipo firmato corrispondente (`ssize_t` in POSIX)
t	d, i, o, u, x o X	`ptrdiff_t` o il corrispondente numero intero senza segno
L	a, A, e, E, f, F, g o G	`long double`

Se un modificatore di lunghezza viene visualizzato con un identificatore di conversione diverso da quello specificato sopra, il comportamento non è definito.

Microsoft specifica alcuni modificatori di lunghezza diversi e in modo esplicito non supporta `hh` , `j` , `z` o `t` .

Modificatore	Modifica	Si applica a
l32	d, i, o, x o X	`__int32`
l32	o, u, x o X	`unsigned __int32`
l64	d, i, o, x o X	`__int64`
l64	o, u, x o X	`unsigned __int64`
io	d, i, o, x o X	`ptrdiff_t` (cioè `__int32` su `__int32` a 32 bit, `__int64` su piattaforme a 64 bit)
io	o, u, x o X	`size_t` (ovvero, `unsigned __int32` su `unsigned __int32` a 32 bit, `unsigned __int64` su piattaforme a 64 bit)
io L	a, A, e, E, f, g o G	`long double` (In Visual C ++, anche se `long double` è un tipo distinto, ha la stessa rappresentazione interna del `double`).
io w	c o C	Ampio personaggio con funzioni `printf` e `wprintf` . (Un `wC` tipo `lc` , `lC` , `wc` o `wC` è sinonimo di `C` nelle funzioni `printf` e con `c` nelle funzioni `wprintf` .)
io w	s, S o Z	Stringa di caratteri `wprintf` funzioni `printf` e `wprintf` . (Un `wS` tipo `ls` , `lS` , `ws` o `wS` è sinonimo di `S` nelle funzioni `printf` e con `s` nelle funzioni `wprintf` .)

Notare che gli `Z` conversione `C` , `S` e `Z` ei modificatori di lunghezza `I` , `I32` , `I64` e `w` sono estensioni Microsoft. Trattare `l` come modificatore per il `long double` anziché il `double` è diverso dallo standard, anche se sarà difficile individuare la differenza a meno che il `long double` non abbia una rappresentazione diversa dal `double` .

Flag di formato di stampa

Lo standard C (C11 e C99) definisce i seguenti flag per `printf()` :

Bandiera	conversioni	Senso
`-`	tutti	Il risultato della conversione deve essere giustificato a sinistra all'interno del campo. La conversione è giustificata a destra se questo flag non è specificato.
`+`	numerico firmato	Il risultato di una conversione firmata inizia sempre con un segno ('+' o '-' '). La conversione inizia con un segno solo quando un valore negativo viene convertito se questo flag non è specificato.
`<space>`	numerico firmato	Se il primo carattere di una conversione firmata non è un segno o se una conversione firmata non produce alcun carattere, uno `<space>` deve essere preceduto dal risultato. Ciò significa che se appaiono entrambi i flag `<space>` e " `+` ", il flag `<space>` deve essere ignorato.
`#`	tutti	Specifica che il valore deve essere convertito in un modulo alternativo. Per `o` conversione, essa aumenta la precisione, se e solo se necessario, di forzare la prima cifra del risultato da uno zero (se il valore e la precisione sono entrambi 0, un singolo 0 è stampato). Per gli specificatori di conversione `x` o `X` , un risultato diverso da zero deve avere `0x` (o `0X`) come prefisso. Per gli specificatori di conversione `a` , `A` , `e` , `E` , `f` , `F` , `g` e `G` , il risultato deve sempre contenere un carattere di radice, anche se nessuna cifra segue il carattere di radice. Senza questo flag, un carattere di radice appare nel risultato di queste conversioni solo se una cifra lo segue. Per gli specificatori di conversione `g` e `G` , gli zeri finali non devono essere rimossi dal risultato come normalmente. Per altri specificatori di conversione, il comportamento non è definito.
`0`	numerico	Per gli identificatori di conversione d, i, o, u, x, X, a, A, e, E, f, G e G, gli zeri iniziali (a seguito di qualsiasi indicazione di segno o base) vengono utilizzati per eseguire il rilievo sul campo larghezza invece di eseguire il riempimento dello spazio, tranne quando si converte un infinito o NaN. Se appaiono entrambi i flag "0" e "-", il flag "0" viene ignorato. Per gli identificatori di conversione d, i, o, u, xe X, se viene specificata una precisione, il flag '0' deve essere ignorato. Se appaiono entrambi i flag "0" e `<apostrophe>` , i caratteri di raggruppamento vengono inseriti prima dello zero padding. Per le altre conversioni, il comportamento non è definito.

Questi flag sono supportati anche da Microsoft con lo stesso significato.

La specifica POSIX per `printf()` aggiunge:

Bandiera	conversioni	Senso
'	i, d, u, f, F, g, G	La parte intera del risultato di una conversione decimale deve essere formattata con migliaia di caratteri di raggruppamento. Per altre conversioni il comportamento non è definito. Viene utilizzato il carattere di raggruppamento non monetario.

Leggi Input / Output formattato online: https://riptutorial.com/it/c/topic/3750/input---output-formattato

Capitolo 36: Insidie comuni

introduzione

Questa sezione discute alcuni degli errori comuni che un programmatore C dovrebbe conoscere e dovrebbe evitare di fare. Per ulteriori informazioni su alcuni problemi imprevisti e le loro cause, vedere comportamento non definito

Examples

Mescolando interi con segno e senza segno nelle operazioni aritmetiche

Di solito non è una buona idea di mescolare `signed` e `unsigned` interi nelle operazioni aritmetiche. Ad esempio, quale sarà l'output del seguente esempio?

```
#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;
}
```

Dal momento che 1000 è superiore a -1, ci si aspetterebbe che l'output fosse `a is more than b` , tuttavia non sarà così.

Le operazioni aritmetiche tra diversi tipi di integrale sono eseguite all'interno di un tipo comune definito dalle cosiddette conversioni aritmetiche usuali (vedere le specifiche del linguaggio, 6.3.1.8).

In questo caso il "tipo comune" è `unsigned int` , perché, come indicato nelle conversioni aritmetiche usuali ,

> 714 Altrimenti, se l'operando che ha un numero intero senza segno ha rank superiore o uguale al rank del tipo di un altro operando, allora l'operando con tipo intero con segno viene convertito nel tipo dell'operando con tipo intero senza segno.

Ciò significa che `int` operando `b` verrà convertito in `unsigned int` prima del confronto.

Quando -1 viene convertito in un `unsigned int` il risultato è il massimo valore `unsigned int` , che è maggiore di 1000, il che significa che `a > b` è falso.

Scrittura errata = invece di == durante il confronto

L'operatore `=` è utilizzato per l'assegnazione.

L'operatore `==` è usato per il confronto.

Si dovrebbe fare attenzione a non mescolare i due. A volte uno scrive per errore

```
/* assign y to x */
if (x = y) {
     /* logic */
}
```

quando ciò che era veramente voluto è:

```
/* compare if x is equal to y */
if (x == y) {
    /* logic */
}
```

Il primo assegna il valore di y a x e controlla se quel valore è diverso da zero, invece di fare il confronto, che è equivalente a:

```
if ((x = y) != 0) {
    /* logic */
}
```

Ci sono momenti in cui il test del risultato di un compito è destinato ed è comunemente usato, perché evita di dover duplicare il codice e di doverlo trattare per la prima volta appositamente. Confrontare

```
while ((c = getopt_long(argc, argv, short_options, long_options, &option_index)) != -1) {
        switch (c) {
        ...
        }
}
```

contro

```
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);
}
```

I compilatori moderni riconosceranno questo modello e non avviseranno quando l'assegnazione è racchiusa tra parentesi come sopra, ma potrebbero mettere in guardia altri usi. Per esempio:

```
if (x = y)         /* warning */

if ((x = y))       /* no warning */
```

```
if ((x = y) != 0)  /* no warning; explicit */
```

Alcuni programmatori usano la strategia di mettere la costante alla sinistra dell'operatore (comunemente chiamata condizioni Yoda). Poiché le costanti sono rvalue, questo tipo di condizione causerà un errore nel compilatore se è stato utilizzato l'operatore sbagliato.

```
if (5 = y) /* Error */

if (5 == y) /* No error */
```

Tuttavia, questo riduce drasticamente la leggibilità del codice e non è considerato necessario se il programmatore segue buone pratiche di codifica in C e non aiuta nel confronto di due variabili, quindi non è una soluzione universale. Inoltre, molti compilatori moderni possono dare avvertimenti quando il codice è scritto con condizioni Yoda.

Uso non corretto del punto e virgola

Fai attenzione ai punti e virgola. Seguendo l'esempio

```
if (x > a);
   a = x;
```

in realtà significa:

```
if (x > a) {}
a = x;
```

il che significa che `x` sarà assegnato ad `a` in ogni caso, il che potrebbe non essere quello che volevi in origine.

A volte, la mancanza di un punto e virgola causerà anche un problema non percepibile:

```
if (i < 0)
    return
day = date[0];
hour = date[1];
minute = date[2];
```

Il punto e virgola dietro il ritorno viene perso, quindi verrà restituito il giorno = data [0].

Una tecnica per evitare questo e problemi simili è quella di usare sempre le parentesi su condizionali e loop multi-linea. Per esempio:

```
if (x > a) {
    a = x;
}
```

Dimenticando di allocare un byte in più per \ 0

Quando copi una stringa in un buffer `malloc` ed, ricorda sempre di aggiungere 1 a `strlen` .

```
char *dest = malloc(strlen(src)); /* WRONG */
char *dest = malloc(strlen(src) + 1); /* RIGHT */

strcpy(dest, src);
```

Questo perché `strlen` non include il trailing `\0` nella lunghezza. Se prendi l'approccio `WRONG` (come mostrato sopra), richiamando `strcpy` , il tuo programma invocherà un comportamento indefinito.

Si applica anche alle situazioni in cui stai leggendo una stringa di lunghezza massima nota da `stdin` o da qualche altra fonte. Per esempio

```
#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 */
```

Dimenticare di liberare memoria (perdite di memoria)

Una buona pratica di programmazione è quella di liberare tutta la memoria che è stata allocata direttamente dal proprio codice, o implicitamente chiamando una funzione interna o esterna, come un'API di libreria come `strdup()` . Non riuscendo a liberare memoria è possibile introdurre una perdita di memoria, che potrebbe accumularsi in una notevole quantità di memoria sprecata che non è disponibile per il programma (o il sistema), causando probabilmente arresti anomali o comportamenti non definiti. È più probabile che si verifichino problemi se la perdita si verifica ripetutamente in un ciclo o in una funzione ricorsiva. Il rischio di errori del programma aumenta più a lungo si verifica un programma che perde. A volte i problemi appaiono all'istante; altre volte i problemi non si vedranno per ore o addirittura anni di funzionamento costante. I fallimenti dell'esaurimento della memoria possono essere catastrofici, a seconda delle circostanze.

Il seguente ciclo infinito è un esempio di perdita che alla fine esaurirà la perdita di memoria disponibile chiamando `getline()` , una funzione che assegna implicitamente nuova memoria, senza liberare quella memoria.

```
#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;
}
```

Al contrario, il codice seguente utilizza anche la funzione `getline()` , ma questa volta, la memoria allocata viene liberata correttamente, evitando una perdita.

```
#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;
}
```

La perdita di memoria non ha sempre conseguenze tangibili e non è necessariamente un problema funzionale. Mentre "best practice" impone rigorosamente la liberazione della memoria in punti strategici e condizioni, per ridurre l'impronta di memoria e ridurre il rischio di esaurimento della memoria, possono esserci delle eccezioni. Ad esempio, se un programma è limitato in termini di durata e portata, il rischio di errore di allocazione potrebbe essere considerato troppo piccolo per preoccuparsi. In tal caso, l'esclusione della deallocazione esplicita potrebbe essere considerata accettabile. Ad esempio, la maggior parte dei sistemi operativi moderni libera automaticamente tutta la memoria consumata da un programma quando termina, sia a causa di un errore del programma, una chiamata di sistema a `exit()` , la chiusura del processo, o raggiungendo la fine di `main()` . Liberare esplicitamente la memoria al momento dell'imminente chiusura del programma potrebbe essere ridondante o introdurre una penalità di prestazioni.

L'allocazione può fallire se è disponibile memoria insufficiente e la gestione dei guasti deve essere tenuta in considerazione ai livelli appropriati dello stack di chiamate. `getline()` , mostrato sopra è un caso d'uso interessante perché è una funzione di libreria che non solo alloca la memoria che lascia al chiamante per liberare, ma può fallire per una serie di ragioni, che devono essere tutte prese in considerazione. Pertanto, quando si utilizza un'API C, è essenziale leggere la *documentazione (pagina man)* e prestare particolare attenzione alle condizioni di errore e

all'utilizzo della memoria e tenere presente quale livello del software ha il peso di liberare memoria restituita.

Un'altra pratica di gestione della memoria comune consiste nell'impostare costantemente i puntatori di memoria su NULL immediatamente dopo che la memoria referenziata da quei puntatori è stata liberata, in modo che quei puntatori possano essere verificati per la validità in qualsiasi momento (ad esempio, verificata per NULL / non NULL), perché l'accesso alla memoria liberata può portare a problemi gravi come ottenere dati inutili (operazione di lettura) o danneggiamento dei dati (operazione di scrittura) e / o arresto anomalo del programma. Nella maggior parte dei sistemi operativi moderni, liberare la posizione di memoria 0 (`NULL`) è un NOP (ad es. È innocuo), come richiesto dallo standard C - quindi impostando un puntatore su NULL, non c'è il rischio di liberare la memoria se il puntatore viene passato a `free()` . Tieni presente che la memoria a doppio rilascio può portare a guasti molto lunghi, confusi e *difficili da diagnosticare* .

Copiando troppo

```
char buf[8]; /* tiny buffer, easy to overflow */

printf("What is your name?\n");
scanf("%s", buf); /* WRONG */
scanf("%7s", buf); /* RIGHT */
```

Se l'utente immette una stringa più lunga di 7 caratteri (- 1 per il terminatore null), la memoria dietro il buffer `buf` verrà sovrascritta. Ciò si traduce in un comportamento indefinito. Gli hacker malintenzionati spesso lo sfruttano per sovrascrivere l'indirizzo di ritorno e cambiarlo all'indirizzo del codice dannoso dell'hacker.

Dimenticando di copiare il valore di ritorno di realloc in un temporaneo

Se `realloc` fallisce, restituisce `NULL` . Se si assegna il valore del buffer originale al valore di ritorno di `realloc` , e se restituisce `NULL` , allora il buffer originale (il vecchio puntatore) viene perso, causando una *perdita di memoria* . La soluzione è copiare in un puntatore temporaneo, e se questo temporaneo non è nullo, **quindi** copiare nel buffer reale.

```
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");
```

Confronto tra numeri in virgola mobile

I tipi di virgola mobile (`float` , `double` e `long double`) non possono rappresentare con precisione alcuni numeri perché hanno una precisione finita e rappresentano i valori in un formato binario. Proprio come abbiamo ripetuto i decimali in base 10 per le frazioni come 1/3, ci sono anche frazioni che non possono essere rappresentate finitamente in binario (come 1/3, ma anche, più importante, 1/10). Non confrontare direttamente i valori in virgola mobile; usa invece un 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;
}
```

Un altro esempio:

```
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;
}
```

Produzione:

```
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)
```

Effettuare il ridimensionamento extra nell'aritmetica del puntatore

Nell'aritmetica del puntatore, il numero intero da aggiungere o sottrarre al puntatore viene interpretato non come cambio di *indirizzo* ma come numero di *elementi* da spostare.

```
#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;
}
```

Questo codice `ptr2` ridimensionamento nel calcolo del puntatore assegnato a `ptr2` . Se `sizeof(int)` è 4, che è tipico nei moderni ambienti a 32 bit, l'espressione sta per "8 elementi dopo `array[0]` ", che è fuori intervallo e richiama il *comportamento non definito* .

Per avere punto `ptr2` corrispondenza di ciò che è 2 elementi dopo l' `array[0]` , devi semplicemente aggiungere 2.

```
#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;
}
```

L'aritmetica con puntatore esplicito che utilizza operatori additivi può essere fonte di confusione, quindi l'utilizzo di un indice di sottoscrizione può essere migliore.

```
#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]` è identico a `(*((E1)+(E2)))` (N1570 6.5.2.1, paragrafo 2), e `&(E1[E2])` è equivalente a `((E1)+(E2))` (N1570 6.5.3.2, nota 102).

In alternativa, se si preferisce l'aritmetica del puntatore, il cast del puntatore per indirizzare un diverso tipo di dati può consentire l'indirizzamento dei byte. Attenzione però: l' endianità può diventare un problema e il casting per tipi diversi da "puntatore al carattere" porta a problemi di aliasing rigorosi .

```
#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;
}
```

Le macro sono semplici sostituzioni di stringhe

Le macro sono semplici sostituzioni di stringhe. (A rigor di termini, funzionano con i token di preelaborazione, non con le stringhe arbitrarie.)

```
#include <stdio.h>

#define SQUARE(x) x*x

int main(void) {
    printf("%d\n", SQUARE(1+2));
    return 0;
}
```

È probabile che questo codice stampi `9` (`3*3`), ma in realtà `5` verrà stampato perché la macro verrà espansa su `1+2*1+2` .

Dovresti avvolgere gli argomenti e l'intera macro espressione tra parentesi per evitare questo problema.

```
#include <stdio.h>

#define SQUARE(x) ((x)*(x))

int main(void) {
    printf("%d\n", SQUARE(1+2));
    return 0;
```

```
}
```

Un altro problema è che gli argomenti di una macro non possono essere valutati una sola volta; potrebbero non essere affatto valutati o potrebbero essere valutati più volte.

```
#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 questo codice, la macro verrà estesa a `((a++) <= (10) ? (a++) : (10))` . Poiché `a++` (`0`) è minore di `10` , `a++` verrà valutato due volte e il valore di `a` e ciò che viene restituito da `MIN` differirà da quanto ci si potrebbe aspettare.

Questo può essere evitato usando le funzioni, ma si noti che i tipi verranno risolti dalla definizione della funzione, mentre i macro possono essere (troppo) flessibili con i tipi.

```
#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;
}
```

Ora il problema della doppia valutazione è fisso, ma questa funzione `min` non può trattare con i `double` dati senza troncare, per esempio.

Le direttive macro possono essere di due tipi:

```
#define OBJECT_LIKE_MACRO     followed by a "replacement list" of preprocessor tokens
#define FUNCTION_LIKE_MACRO(with, arguments) followed by a replacement list
```

Ciò che distingue questi due tipi di macro è il carattere che segue l'identificatore dopo `#define` : se è un *lparen* , è una macro simile alla funzione; altrimenti, è una macro simile ad un oggetto. Se l'intenzione è quella di scrivere una macro funzione di simile, non ci deve essere alcun spazio bianco tra la fine del nome della macro e `(` . Controllare questo per una spiegazione dettagliata.

C99

In C99 o versioni successive, è possibile utilizzare `static inline int min(int x, int y) { … }` .

C11

In C11, potresti scrivere un'espressione 'type-generic' per `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;
}
```

L'espressione generica potrebbe essere estesa con più tipi come `double` , `float` , `long long` , `unsigned long` , `long` , `unsigned` - e appropriato `gen_min` macro invocazioni scritte.

Errori di riferimento non definiti durante il collegamento

Uno degli errori più comuni nella compilazione si verifica durante la fase di collegamento. L'errore è simile a questo:

```
$ 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
$
```

Diamo un'occhiata al codice che ha generato questo errore:

```
int foo(void);

int main(int argc, char **argv)
{
    int foo_val;
    foo_val = foo();
    return foo_val;
}
```

Vediamo qui una *dichiarazione* di foo (`int foo();`) ma nessuna *definizione* di esso (funzione effettiva). Abbiamo quindi fornito al compilatore l'intestazione della funzione, ma non è stata definita alcuna funzione, quindi la fase di compilazione passa ma il linker viene chiuso con un errore di `Undefined reference` .
Per correggere questo errore nel nostro piccolo programma dovremmo solo aggiungere una *definizione* per foo:

```
/* 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;
}
```

Ora questo codice verrà compilato. Si presenta una situazione alternativa in cui il source per `foo()` trova in un file sorgente separato `foo.c` (e c'è un'intestazione `foo.h` per dichiarare `foo()` che è incluso in `foo.c` e `undefined_reference.c`). Quindi la correzione è di collegare sia il file oggetto da `foo.c` e `undefined_reference.c` , o di compilare entrambi i file sorgente:

```
$ gcc -c undefined_reference.c
$ gcc -c foo.c
$ gcc -o working_program undefined_reference.o foo.o
$
```

O:

```
$ gcc -o working_program undefined_reference.c foo.c
$
```

Un caso più complesso è dove sono coinvolte le librerie, come nel codice:

```
#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;
}
```

Il codice è sintatticamente corretto, la dichiarazione per `pow()` esiste da `#include <math.h>` , quindi proviamo a compilare e collegare ma otteniamo un errore come questo:

```
$ 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
$
```

Ciò accade perché la *definizione* di `pow()` non è stata trovata durante la fase di collegamento. Per risolvere questo problema, dobbiamo specificare che vogliamo collegarci alla libreria matematica chiamata `libm` specificando il flag `-lm` . (Si noti che esistono piattaforme come macOS dove `-lm` non è necessario, ma quando si ottiene il riferimento non definito, la libreria è necessaria).

Quindi eseguiamo nuovamente la fase di compilazione, questa volta specificando la libreria (dopo i file di origine o di oggetto):

```
$ 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
$
```

E funziona!

Degrado dell'array malinteso

Un problema comune nel codice che utilizza matrici multidimensionali, matrici di puntatori, ecc. È il fatto che `Type**` e `Type[M][N]` sono tipi fondamentalmente diversi:

```
#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;
}
```

Esempio di output del compilatore:

```
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)
```

L'errore indica che l'array `s` nella funzione `main` viene passato alla funzione `print_strings` , che si aspetta un tipo di puntatore diverso da quello ricevuto. Include anche una nota che esprime il tipo che ci si aspetta da `print_strings` e il tipo che è stato trasmesso dal `main` .

Il problema è dovuto a qualcosa chiamato *decadimento dell'array* . Quello che succede quando `s` con il suo tipo `char[4][20]` (array di 4 matrici di 20 caratteri) viene passato alla funzione è che si trasforma in un puntatore al suo primo elemento come se tu avessi scritto `&s[0]` , che ha il tipo `char (*)[20]` (puntatore a 1 array di 20 caratteri). Ciò si verifica per qualsiasi array, inclusi un array di puntatori, una matrice di matrici di array (matrici 3D) e una matrice di puntatori a un array. Di seguito è riportata una tabella che illustra cosa succede quando un array decade. Le modifiche nella descrizione del tipo sono evidenziate per illustrare cosa succede:

Prima della decomposizione		Dopo il decadimento	
`char [20]`	**serie di (20 caratteri)**	`char *`	**puntatore a (1 carattere)**
`char [4][20]`	**array di (4 array** di 20 caratteri)	`char (*)[20]`	**puntatore a (1 array** di 20 caratteri)
`char *[4]`	**array di (4 puntatori** a 1 carattere)	`char **`	**puntatore a (1 puntatore** a 1 carattere)
`char [3][4][20]`	**array di (3 matrici** di 4 matrici di 20 caratteri)	`char (*)[4][20]`	**puntatore a (1 array** di 4 matrici di 20 caratteri)
`char (*[4])[20]`	**array di (4 puntatori** a 1 array di 20 caratteri)	`char (**)[20]`	**puntatore a (1 puntatore** a 1 array di 20 caratteri)

Se una matrice può decadere su un puntatore, allora si può affermare che un puntatore può

essere considerato un array di almeno 1 elemento. Un'eccezione a questo è un puntatore nullo, che punta a nulla e di conseguenza non è un array.

Il decadimento delle matrici avviene solo una volta. Se una matrice è decaduta su un puntatore, ora è un puntatore, non una matrice. Anche se hai un puntatore a un array, ricorda che il puntatore potrebbe essere considerato come un array di almeno un elemento, quindi il decadimento dell'array si è già verificato.

In altre parole, un puntatore a un array (`char (*)[20]`) non diventerà mai un puntatore a un puntatore (`char **`). Per correggere la funzione `print_strings` , è sufficiente farla ricevere il tipo corretto:

```
void print_strings(char (*strings)[20], size_t n)
/* OR */
void print_strings(char strings[][20], size_t n)
```

Un problema sorge quando vuoi che la funzione `print_strings` sia generica per qualsiasi array di caratteri: cosa succede se ci sono 30 caratteri invece di 20? O 50? La risposta è aggiungere un altro parametro prima del parametro array:

```
#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;
}
```

La compilazione non produce errori e produce l'output atteso:

```
Example 1
Example 2
Example 3
Example 4
```

Passaggio di array non adiacenti a funzioni che prevedono array "reali" multidimensionali

Quando si assegnano array multidimensionali con `malloc` , `calloc` e `realloc` , uno schema comune è quello di allocare gli array interni con più chiamate (anche se la chiamata viene visualizzata solo una volta, potrebbe essere in un ciclo):

```
/* 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]));
```

La differenza di byte tra l'ultimo elemento di uno degli array interni e il primo elemento dell'array interno successivo potrebbe non essere 0 come sarebbe con un array multidimensionale "reale" (ad es. `int array[4][16];`) :

```
/* 0x40003c, 0x402000 */
printf("%p, %p\n", (void *)(array[0] + 15), (void *)array[1]);
```

Tenendo conto della dimensione di `int` , si ottiene una differenza di 8128 byte (8132-4), che è 2032 elementi di array `int` -size, e questo è il problema: un array multidimensionale "reale" non ha spazi tra gli elementi.

Se è necessario utilizzare un array allocato dinamicamente con una funzione che si aspetta un array "reale" multidimensionale, è necessario allocare un oggetto di tipo `int *` e utilizzare l'aritmetica per eseguire calcoli:

```
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);
```

Se `N` è una macro o un intero letterale piuttosto che una variabile, il codice può semplicemente utilizzare la notazione di array 2-D più naturale dopo aver assegnato un puntatore a un array:

```
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

Se `N` non è una macro o un intero letterale, la `array` punta a un array a lunghezza variabile (VLA). Questo può ancora essere usato con `func` `func_vla` a `int *` e una nuova funzione `func_vla` sostituirà `func_N` :

```
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

Nota : i VLA sono opzionali a partire da C11. Se la tua implementazione supporta C11 e definisce la macro `__STDC_NO_VLA__` su 1, sei bloccato con i metodi pre-C99.

Utilizzare le costanti di carattere anziché i valori letterali stringa e viceversa

In C, le costanti dei caratteri e le stringhe letterali sono cose diverse.

Un personaggio circondato da virgolette singole come `'a'` è una *costante di carattere* . Una costante di carattere è un numero intero il cui valore è il codice carattere che rappresenta il carattere. Come interpretare le costanti dei caratteri con più caratteri come `'abc'` è definito dall'implementazione.

Zero o più caratteri circondati da virgolette come `"abc"` è una *stringa letterale* . Una stringa letterale è una matrice non modificabile i cui elementi sono di tipo `char` . La stringa tra virgolette più terminazione null-character è il contenuto, quindi `"abc"` ha 4 elementi (`{'a', 'b', 'c', '\0'}`)

In questo esempio, viene utilizzata una costante di carattere in cui deve essere utilizzato un valore letterale stringa. Questa costante di carattere verrà convertita in un puntatore in un modo definito dall'implementazione e ci sono poche possibilità che il puntatore convertito sia valido, quindi questo esempio invocherà un *comportamento non definito* .

```
#include <stdio.h>

int main(void) {
    const char *hello = 'hello, world'; /* bad */
    puts(hello);
    return 0;
}
```

In questo esempio, viene utilizzata una stringa letterale in cui deve essere utilizzata una costante di carattere. Il puntatore convertito dal letterale stringa verrà convertito in un numero intero in un modo definito dall'implementazione e verrà convertito in `char` in un modo definito dall'implementazione. (Come convertire un intero in un tipo firmato che non può rappresentare il valore da convertire è definito dall'implementazione, e anche se il `char` è firmato è anche definito dall'implementazione.) L'output sarà una cosa priva di significato.

```
#include <stdio.h>

int main(void) {
    char c = "a"; /* bad */
    printf("%c\n", c);
    return 0;
}
```

In quasi tutti i casi, il compilatore si lamenterà di questi errori. In caso contrario, è necessario utilizzare più opzioni di avviso del compilatore o si consiglia di utilizzare un compilatore migliore.

Ignorare i valori di ritorno delle funzioni della libreria

Quasi tutte le funzioni della libreria standard C restituiscono qualcosa in caso di successo e qualcos'altro in caso di errore. Ad esempio, `malloc` restituirà un puntatore al blocco di memoria assegnato dalla funzione in caso di successo e, se la funzione non è riuscita ad allocare il blocco di memoria richiesto, un puntatore nullo. Quindi dovresti sempre controllare il valore di ritorno per facilitare il debugging.

Questo non va bene:

```
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 */
```

Questo è buono:

```
#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;
}
```

In questo modo sai subito la causa dell'errore, altrimenti potresti passare ore a cercare un bug in un posto completamente sbagliato.

Il carattere di nuova riga non viene utilizzato nella tipica chiamata scanf ()

Quando questo programma

```
#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;
}
```

viene eseguito con questo input

```
42
life
```

l'uscita sarà `42 ""` invece di `42 "life"` prevista.

Questo perché un carattere di nuova riga dopo `42` non viene consumato nella chiamata di `scanf()` e viene consumato da `fgets()` prima che legga la `life` . Quindi, `fgets()` smette di leggere prima di leggere la `life` .

Per evitare questo problema, un modo che è utile quando la lunghezza massima di una linea è nota - quando si risolvono i problemi nel sistema di giudice online, ad esempio - è evitare l'uso di `scanf()` direttamente e la lettura di tutte le linee tramite `fgets()` . È possibile utilizzare `sscanf()` per analizzare le righe lette.

```
#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;
}
```

Un altro modo è leggere fino a quando non si preme un carattere di nuova riga dopo aver usato `scanf()` e prima di usare `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;
}
```

Aggiungere un punto e virgola a un #define

È facile confondersi nel preprocessore C e trattarlo come parte della C stessa, ma è un errore perché il preprocessore è solo un meccanismo di sostituzione del testo. Ad esempio, se scrivi

```
/* WRONG */
#define MAX 100;
int arr[MAX];
```

il codice si espande in

```
int arr[100;];
```

che è un errore di sintassi. Il rimedio è rimuovere il punto e virgola dalla riga `#define` . È quasi sempre un errore terminare un `#define` con un punto e virgola.

I commenti a più righe non possono essere nidificati

In C, i commenti su più righe, / * e * /, non annidano.

Se annoti un blocco di codice o funzione usando questo stile di commento:

```
/*
 * 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;
}
```

Non sarai in grado di commentarlo facilmente:

```
//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...
*/
```

Una soluzione è usare i commenti in stile C99:

```
// 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;
}
```

Ora l'intero blocco può essere commentato facilmente:

```
/*

// 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;
}

*/
```

Un'altra soluzione è quella di evitare il codice disabilitazione usando la sintassi commento, utilizzando `#ifdef` o `#ifndef` direttive del preprocessore invece. Queste direttive *fanno* nido, lasciandovi liberi di commentare il codice nello stile che si preferisce.

```
#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
```

Alcune guide arrivano al punto di raccomandare che le sezioni di codice *non* debbano *mai* essere commentate e che se il codice deve essere temporaneamente disattivato si potrebbe ricorrere all'utilizzo di una direttiva `#if 0` .

Vedi #if 0 per bloccare le sezioni di codice .

Superamento dei limiti dell'array

Le matrici sono basate su zero, ovvero l'indice inizia sempre da 0 e termina con la lunghezza dell'array dell'indice meno 1, pertanto il codice seguente non emetterà il primo elemento dell'array e restituirà il garbage per il valore finale che stampa.

```
#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;
}
```

Uscita: `2 3 4 5 GarbageValue`

Di seguito viene illustrato il modo corretto per ottenere l'output desiderato:

```
#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;
}
```

Uscita: `1 2 3 4 5`

È importante conoscere la lunghezza di un array prima di utilizzarlo, altrimenti si potrebbe danneggiare il buffer o causare un errore di segmentazione accedendo a posizioni di memoria fuori limite.

Funzione ricorsiva: manca la condizione di base

Calcolare il fattoriale di un numero è un classico esempio di funzione ricorsiva.

Manca la condizione di base:

```
#include <stdio.h>

int factorial(int n)
{
       return n * factorial(n - 1);
}

int main()
{
    printf("Factorial %d = %d\n", 3, factorial(3));
    return 0;
}
```

Uscita tipica: `Segmentation fault: 11`

Il problema con questa funzione è il loop infinito, che causa un errore di segmentazione: è necessaria una condizione di base per interrompere la ricorsione.

Condizione di base dichiarata:

```
#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;
}
```

Uscita di esempio

```
Factorial 3 = 6
```

Questa funzione termina non appena colpisce la condizione `n` è uguale a 1 (a condizione che il valore iniziale di `n` sia abbastanza piccolo - il limite superiore è `12` quando `int` è una quantità di 32 bit).

Regole da seguire:

1. Inizializza l'algoritmo. I programmi ricorsivi spesso hanno bisogno di un valore iniziale da cui partire. Ciò può essere ottenuto utilizzando un parametro passato alla funzione o fornendo una funzione gateway che non è ricorsiva ma che imposta i valori seme per il calcolo ricorsivo.
2. Verificare se i valori correnti in fase di elaborazione corrispondono al caso base. In tal caso, elaborare e restituire il valore.
3. Ridefinisci la risposta in termini di un sotto-problema o sotto-problema più piccolo o più semplice.
4. Esegui l'algoritmo sul sotto-problema.
5. Combina i risultati nella formulazione della risposta.
6. Restituire i risultati.

Fonte: funzione ricorsiva

Controllo dell'espressione logica contro 'vero'

Lo standard C originale non aveva alcun tipo booleano intrinseco, quindi `bool` , `true` e `false` non avevano alcun significato intrinseco e venivano spesso definiti dai programmatori. Tipicamente `true` sarebbe definito come 1 e `false` sarebbe definito come 0.

C99

C99 aggiunge il tipo built-in `_Bool` e l'intestazione `<stdbool.h>` che definisce `bool` (espandibile in `_Bool`), `false` e `true` . Permette anche di ridefinire `bool` , `true` e `false` , ma osserva che questa è una caratteristica obsoleta.

Ancora più importante, le espressioni logiche trattano tutto ciò che viene valutato a zero come falso e qualsiasi valutazione diversa da zero come vera. Per esempio:

```
/* 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;
    }
}
```

Nell'esempio sopra, la funzione sta cercando di verificare se il bit superiore è impostato e restituisce `true` se lo è. Tuttavia, controllando esplicitamente contro `true` , l'istruzione `if` avrà successo solo se `(bitfield & 0x80)` valutata a qualsiasi cosa sia definita `true` , che è in genere `1` e molto raramente `0x80` . O esplicitamente controllare il caso che ti aspetti:

```
/* 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;
    }
}
```

O valuta qualsiasi valore diverso da zero come vero.

```
/* 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;
    }
}
```

I valori letterali in virgola mobile sono di tipo double per impostazione predefinita

Bisogna fare attenzione quando si inizializzano le variabili del tipo `float` su valori letterali o confrontandole con valori letterali, perché i letterali a virgola mobile regolari come `0.1` sono di tipo `double` . Questo può portare a sorprese:

```
#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
```

Qui, `n` viene inizializzato e arrotondato alla precisione singola, risultando in valore 0.10000000149011612. Quindi, `n` viene riconvertito in doppia precisione per essere confrontato con `0.1` letterale (che equivale a 0.10000000000000001), risultando in una mancata corrispondenza.

Oltre agli errori di arrotondamento, la miscelazione di variabili `float` con valori letterali `double` comporterà scarse prestazioni su piattaforme che non dispongono di supporto hardware per la doppia precisione.

Leggi Insidie comuni online: https://riptutorial.com/it/c/topic/2006/insidie---comuni

Capitolo 37: Interprocess Communication (IPC)

introduzione

I meccanismi di comunicazione tra processi (IPC) consentono a diversi processi indipendenti di comunicare tra loro. Lo standard C non fornisce alcun meccanismo IPC. Pertanto, tutti questi meccanismi sono definiti dal sistema operativo host. POSIX definisce un ampio insieme di meccanismi IPC; Windows definisce un altro set; e altri sistemi definiscono le proprie varianti.

Examples

semafori

I semafori vengono utilizzati per sincronizzare le operazioni tra due o più processi. POSIX definisce due diversi set di funzioni semaforo:

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()` .

Questa sezione descrive i semafori IPC System V, così chiamati perché originati da Unix System V.

Innanzitutto, devi includere le intestazioni richieste. Le vecchie versioni di POSIX richiedevano `#include <sys/types.h>` ; POSIX moderno e la maggior parte dei sistemi non lo richiedono.

```
#include <sys/sem.h>
```

Quindi, dovrai definire una chiave sia nel genitore che nel bambino.

```
#define KEY 0x1111
```

Questa chiave deve essere la stessa in entrambi i programmi o non si riferiscono alla stessa struttura IPC. Ci sono modi per generare una chiave concordata senza codificare il suo valore.

Successivamente, a seconda del compilatore, potrebbe essere necessario o meno eseguire questo passaggio: dichiarare un'unione ai fini delle operazioni del semaforo.

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short  *array;
};
```

Quindi, definisci le strutture *try* (`semwait`) e *raise* (`semsignal`). I nomi P e V provengono dall'olandese

```
struct sembuf p = { 0, -1, SEM_UNDO}; # semwait
struct sembuf v = { 0, +1, SEM_UNDO}; # semsignal
```

Ora, inizia prendendo l'id per il tuo semaforo IPC.

```
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 */
}
```

Nel genitore, inizializza il semaforo per avere un contatore di 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 */
}
```

Ora puoi decrementare o incrementare il semaforo di cui hai bisogno. All'inizio della tua sezione critica, decrementi il contatore usando la funzione `semop()` :

```
if (semop(id, &p, 1) < 0) {
    /* error handling code */
}
```

Per incrementare il semaforo, usi `&v` invece di `&p` :

```
if (semop(id, &v, 1) < 0) {
    /* error handling code */
}
```

Nota che ogni funzione restituisce `0` in caso di successo e `-1` in caso di fallimento. Non controllare questi stati di ritorno può causare problemi devastanti.

Esempio 1.1: Corse con discussioni

Il programma seguente avrà un `fork` processo e sia padre che figlio tenteranno di stampare caratteri sul terminale senza alcuna sincronizzazione.

```
#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);
        }
    }
}
```

Uscita (1a corsa):

```
aAABaBCbCbDDcEEcddeFFGGHHeffgghh
```

(2a corsa):

```
aabbccAABddBCeeCffgDDghEEhFFGGHH
```

Compilare ed eseguire questo programma dovrebbe darti un output diverso ogni volta.

Esempio 1.2: Evita le corse con i semafori

Modificando l' *Esempio 1.1* per usare i semafori, abbiamo:

```
#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);
        }
    }
}
```

Produzione:

```
aabbAABBCCccddeeDDffEEFFGGHHgghh
```

Compilare ed eseguire questo programma ti darà la stessa uscita ogni volta.

Leggi Interprocess Communication (IPC) online:
https://riptutorial.com/it/c/topic/10564/interprocess-communication--ipc-

Capitolo 38: Iterazioni / loop di iterazione: per, while, do-while

Sintassi

- / * tutte le versioni * /
- per ([espressione]; [espressione]; [espressione]) one_statement
- for ([espressione]; [espressione]; [espressione]) {zero o più affermazioni}
- while (espressione) one_statement
- while (espressione) {zero o più istruzioni}
- fai uno_statoment while (espressione);
- fare {una o più affermazioni} mentre (espressione);
- // dal C99 in aggiunta al modulo sopra
- per (dichiarazione; [espressione]; [espressione]) one_statement;
- for (declaration; [expression]; [expression]) {zero o più istruzioni}

Osservazioni

Iteration Statement / Loops si dividono in due categorie:

- istruzione / loop di iterazione controllata dalla testa
- istruzione / loop di iterazione comandata a pedale

Dichiarazione / cicli di iterazione controllati dalla testa

```
for ([<expression>]; [<expression>]; [<expression>]) <statement>
while (<expression>) <statement>
```

C99

```
for ([declaration expression]; [expression] [; [expression]]) statement
```

Iterazione / loop di iterazione a pedale

```
do <statement> while (<expression>);
```

Examples

Per ciclo

Per eseguire un blocco di codice su un altro, i loop entrano in scena. Il ciclo `for` deve essere utilizzato quando un blocco di codice deve essere eseguito un numero fisso di volte. Ad esempio,

per riempire una matrice di dimensione `n` con gli input dell'utente, è necessario eseguire `scanf()` per `n` volte.

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]);
}
```

In questo modo la chiamata della funzione `scanf()` viene eseguita `n` volte (10 volte nel nostro esempio), ma viene scritta una sola volta.

Qui, la variabile `i` è l'indice del ciclo e viene dichiarata come presentata. Il tipo `size_t` (*tipo di dimensione*) deve essere utilizzato per tutto ciò che conta o esegue il ciclo tra gli oggetti dati.

Questo modo di dichiarare le variabili all'interno di `for` è disponibile solo per i compilatori che sono stati aggiornati allo standard C99. Se per qualche motivo sei ancora bloccato con un compilatore più vecchio, puoi dichiarare l'indice del ciclo prima del ciclo `for` :

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]);
}
```

Mentre loop

Un ciclo `while` viene utilizzato per eseguire un pezzo di codice mentre una condizione è vera. Il ciclo `while` deve essere usato quando un blocco di codice deve essere eseguito un numero variabile di volte. Ad esempio, il codice mostrato ottiene l'input dell'utente, a condizione che l'utente inserisca numeri che non sono `0` . Se l'utente inserisce `0` , la condizione while non è più true, quindi l'esecuzione uscirà dal ciclo e continuerà su qualsiasi codice successivo:

```
int num = 1;

while (num != 0)
{
    scanf("%d", &num);
}
```

Ciclo Do-While

A differenza `for` cicli `while` e `while` , i loop `do-while` verificano la verità della condizione alla fine del ciclo, il che significa che il blocco `do` verrà eseguito una volta, quindi verificherà la condizione del `while` nella parte inferiore del blocco. Il che significa che un `do-while` ciclo sarà *sempre* eseguito almeno una volta.

Ad esempio questo ciclo `do-while` otterrà i numeri dall'utente, fino a quando la somma di questi valori è maggiore o uguale a `50` :

```
int num, sum;
num = sum = 0;

do
{
  scanf("%d", &num);
  sum += num;

} while (sum < 50);
```

`do-while` cicli di `do-while` sono relativamente rari nella maggior parte degli stili di programmazione.

Struttura e flusso di controllo in un ciclo for

```
for ([declaration-or-expression]; [expression2]; [expression3])
{
    /* body of the loop */
}
```

In un ciclo `for` , la condizione del ciclo ha tre espressioni, tutte opzionali.

- La prima espressione, `declaration-or-expression` , *inizializza* il ciclo. Viene eseguito esattamente una volta all'inizio del ciclo.

C99

Può essere una dichiarazione e l'inizializzazione di una variabile di loop o un'espressione generale. Se si tratta di una dichiarazione, l'ambito della variabile dichiarata è limitato dall'istruzione `for` .

C99

Le versioni storiche di C consentivano solo un'espressione, qui, e la dichiarazione di una variabile di loop doveva essere posizionata prima del `for` .

- La seconda espressione, `expression2` , è la *condizione di test* . Viene prima eseguito dopo l'inizializzazione. Se la condizione è `true` , il controllo entra nel corpo del ciclo. In caso contrario, si sposta all'esterno del corpo del ciclo alla fine del ciclo. Successivamente, questa condizione viene verificata dopo ogni esecuzione del corpo, nonché la dichiarazione di aggiornamento. Quando è `true` , il controllo torna all'inizio del corpo del loop. La condizione è solitamente intesa come un controllo sul numero di volte in cui il corpo del ciclo viene eseguito. Questo è il modo principale di uscire da un ciclo, l'altro modo è usare le istruzioni di salto .

- La terza espressione, `expression3` , è l' *istruzione di aggiornamento* . Viene eseguito dopo ogni esecuzione del corpo del loop. Viene spesso usato per incrementare una variabile tenendo conto del numero di volte che il corpo del ciclo ha eseguito, e questa variabile è chiamata *iteratore* .

Ogni istanza di esecuzione del corpo del ciclo viene chiamata *iterazione* .

Esempio:

C99

```
for(int i = 0; i < 10 ; i++)
{
    printf("%d", i);
}
```

L'output è:

```
0123456789
```

Nell'esempio sopra, viene eseguito first `i = 0` , inizializzando `i` . Quindi, viene controllata la condizione `i < 10` , che risulta essere `true` . Il controllo entra nel corpo del ciclo e il valore di `i` viene stampato. Quindi, il controllo passa a `i++` , aggiornando il valore di `i` da 0 a 1. Quindi, la condizione viene nuovamente controllata e il processo continua. Questo va avanti finché il valore di `i` diventa 10. Quindi, la condizione `i < 10` valutata `false` , dopo di che il controllo si sposta fuori dal ciclo.

Cicli infiniti

Si dice che un ciclo è un *ciclo infinito* se il controllo entra ma non lascia mai il corpo del ciclo. Questo accade quando la condizione di test del loop non è mai `false` .

Esempio:

C99

```
for (int i = 0; i >= 0; )
{
    /* body of the loop where i is not changed*/
}
```

Nell'esempio sopra, la variabile `i` , l'iteratore, viene inizializzata su 0. La condizione di test è inizialmente `true` . Tuttavia, `i` non viene modificato in qualsiasi parte del corpo e l'espressione aggiornamento è vuoto. Quindi, `i` rimarrà 0, e la condizione di prova sarà mai valutata come `false` , portando ad un ciclo infinito.

Supponendo che non ci siano istruzioni di salto, un altro modo in cui un loop infinito potrebbe essere formato è mantenendo esplicitamente la condizione vera:

```
while (true)
```

```
{
    /* body of the loop */
}
```

In un ciclo `for` , l'istruzione condizione facoltativa. In questo caso, la condizione è sempre `true` vacuo, portando a un ciclo infinito.

```
for (;;)
{
    /* body of the loop */
}
```

Tuttavia, in alcuni casi, la condizione potrebbe essere mantenuta `true` intenzionalmente, con l'intenzione di uscire dal ciclo usando un'istruzione jump come `break` .

```
while (true)
{
    /* statements */
    if (condition)
    {
         /* more statements */
         break;
    }
}
```

Loop srotolamento e dispositivo di Duff

A volte, il ciclo diretto non può essere interamente contenuto all'interno del corpo del loop. Questo perché, il ciclo deve essere innescato da alcune affermazioni ***B.*** Quindi, l'iterazione inizia con alcune istruzioni ***A*** , che sono poi seguite da ***B di*** nuovo prima del ciclo.

```
do_B();
while (condition) {
    do_A();
    do_B();
}
```

Per evitare potenziali problemi di taglia / incolla ripetendo ***B*** due volte nel codice, è possibile applicare il dispositivo di Duff per avviare il loop dal centro del corpo `while` usa l' istruzione switch e il comportamento di caduta.

```
switch (true) while (condition) {
case false: do_A(); /* FALL THROUGH */
default:    do_B(); /* FALL THROUGH */
}
```

Il dispositivo di Duff è stato in realtà inventato per implementare lo srotolamento del loop. Immagina di applicare una maschera a un blocco di memoria, dove `n` è un tipo integrale firmato con un valore positivo.

```
do {
```

```
    *ptr++ ^= mask;
} while (--n > 0);
```

Se `n` fosse sempre divisibile per 4, potresti srotolarlo facilmente come:

```
do {
    *ptr++ ^= mask;
    *ptr++ ^= mask;
    *ptr++ ^= mask;
    *ptr++ ^= mask;
} while ((n -= 4) > 0);
```

Ma con Duff's Device, il codice può seguire questo idioma di svolgimento che salta nel punto giusto nel mezzo del ciclo se `n` non è divisibile per 4.

```
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);
```

Questo tipo di srotolamento manuale è raramente richiesto con i moderni compilatori, poiché il motore di ottimizzazione del compilatore può srotolare loop per conto del programmatore.

Leggi Iterazioni / loop di iterazione: per, while, do-while online:
https://riptutorial.com/it/c/topic/5151/iterazioni---loop-di-iterazione--per--while--do-while

Capitolo 39: Jump Statements

Sintassi

- return val; / * Restituisce dalla funzione corrente. val può essere un valore di qualsiasi tipo che viene convertito nel tipo di ritorno della funzione. * /
- ritorno; / * Restituisce dalla funzione void corrente. * /
- rompere; / * Salta incondizionatamente oltre la fine ("break out") di un'istruzione di iterazione (loop) o fuori dall'istruzione switch più interna. * /
- Continua; / * Salta incondizionatamente all'inizio di un'istruzione di iterazione (loop). * /
- goto LBL; / * Salta all'etichetta LBL. * /
- LBL: *statement* / * qualsiasi istruzione nella stessa funzione. * /

Osservazioni

Questi sono i salti che sono integrati in C per mezzo di parole chiave.

C ha anche un altro costrutto salto, *salto in lungo,* che è specificato con un tipo di dati, `jmp_buf` , e chiamate alle librerie C, `setjmp` e `longjmp` .

Guarda anche

Iterazioni / loop di iterazione: per, while, do-while

Examples

Usando goto per saltare fuori da loop annidati

Saltare fuori da cicli annidati di solito richiede l'uso di una variabile booleana con un controllo di questa variabile nei loop. Supponiamo che stiamo iterando su `i` e `j` , potrebbe assomigliare a questo

```
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 */
     }
}
```

Ma il linguaggio C offre la clausola `goto` , che può essere utile in questo caso. Usandolo con un'etichetta dichiarata dopo i loop, possiamo facilmente uscire dai loop.

```
size_t i,j;
for (i = 0; i < myValue; ++i) {
    for (j = 0; j < mySecondValue; ++j) {
```

```
        ...
        if(breakout_condition)
          goto final;
    }
}
final:
```

Tuttavia, spesso quando si presenta questa necessità, è preferibile utilizzare un `return` . Questo costrutto è anche considerato "non strutturato" nella teoria della programmazione strutturale.

Un'altra situazione in cui `goto` potrebbe essere utile è passare a un gestore di errori:

```
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;
```

L'uso di `goto` mantiene il flusso degli errori separato dal normale flusso di controllo del programma. Tuttavia è anche considerato "non strutturato" in senso tecnico.

Usando il ritorno

Restituzione di un valore

Un caso comunemente usato: ritorno da `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;
}
```

Note aggiuntive:

1. Per una funzione che ha un tipo restituito come `void` (non includendo `void *` o tipi correlati), l'istruzione `return` non dovrebbe avere alcuna espressione associata; cioè, l'unica dichiarazione di reso ammessa sarebbe di `return;` .

2. Per una funzione che ha un tipo di `void` non `void` , la dichiarazione di `return` non deve apparire senza un'espressione.

3. Per `main()` (e solo per `main()`), non è richiesta un'istruzione `return` *esplicita* (in C99 o successive). Se l'esecuzione raggiunge il termine `}` , viene restituito un valore implicito di `0` . Alcune persone pensano che omettere questo `return` sia una cattiva pratica; altri suggeriscono attivamente di lasciarlo fuori.

Non restituire nulla

Di ritorno da una funzione di `void`

```
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. */
}
```

Usando la pausa e continua

`continue` immediatamente a leggere su input non validi o `break` la richiesta dell'utente o la fine del file:

```
#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);
  }
}
```

Leggi Jump Statements online: https://riptutorial.com/it/c/topic/5568/jump-statements

Capitolo 40: Letterali composti

Sintassi

- (tipo) {inizializzatore-elenco}

Osservazioni

Lo standard C dice in C11-§6.5.2.5 / 3:

> Un'espressione postfissa che consiste in un nome di tipo parentesi seguito da un elenco di inizializzatori racchiuso tra parentesi è un *letterale composto* . Fornisce un oggetto senza nome il cui valore è dato dall'elenco di inizializzazione. [99)]

e la nota 99 dice:

> Si noti che questo differisce da un'espressione cast. Ad esempio, un cast specifica una conversione in tipi scalari o solo **void** , e il risultato di un'espressione cast non è un lvalue.

Nota che:

> I valori letterali delle stringhe e i letterali composti con tipi qualificati const non devono necessariamente designare oggetti distinti. [101)]
>
> 101) Ciò consente alle implementazioni di condividere lo spazio di archiviazione per valori letterali costanti e costanti composti costanti con rappresentazioni uguali o sovrapposte.

L'esempio è dato in standard:
C11-§6.5.2.5 / 13:

> Come i letterali delle stringhe, i letterali composti const-qualified possono essere inseriti in una memoria di sola lettura e possono anche essere condivisi. Per esempio,
>
> ```
> (const char []){"abc"} == "abc"
> ```
>
> potrebbe produrre 1 se la memoria dei letterali è condivisa.

Examples

Definizione / Inizializzazione di composti letterali

Un letterale composto è un oggetto senza nome che viene creato nell'ambito in cui è definito. Il concetto è stato introdotto per la prima volta nello standard C99. Un esempio per il letterale composto è

Esempi dallo standard C, C11-§6.5.2.5 / 9:

```
int *p = (int [2]){ 2, 4 };
```

`p` è inizializzato all'indirizzo del primo elemento di una matrice anonima di due interi.

Il composto letterale è un lvalue. La durata di memorizzazione dell'oggetto senza nome è statica (se il letterale viene visualizzato nell'ambito del file) o automatica (se il letterale viene visualizzato nell'ambito del blocco) e in quest'ultimo caso la durata dell'oggetto termina quando il controllo lascia il blocco che lo contiene.

```
void f(void)
{
    int *p;
    /*...*/
    p = (int [2]){ *p };
    /*...*/
}
```

> `p` è assegnato l'indirizzo del primo elemento di una matrice di due interi, il primo con il valore precedentemente indicato da `p` e il secondo, zero. [...]

Qui `p` rimane valido fino alla fine del blocco.

Letterale composto con designatori

(anche da 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});
```

Un `drawline` funzione fittizia riceve due argomenti di tipo `struct point` . Il primo ha valori di coordinate `x == 1 y == 1` , mentre il secondo ha `x == 3 y == 4`

Letterale composto senza specificare la lunghezza dell'array

```
int *p = (int []){ 1, 2, 3};
```

In questo caso la dimensione dell'array non è specificata, quindi sarà determinata dalla lunghezza dell'inizializzatore.

Letterale composto con lunghezza dell'inizializzatore inferiore alla dimensione dell'array specificata

```
int *p = (int [10]){1, 2, 3};
```

il resto degli elementi del letterale composto sarà inizializzato a `0` implicitamente.

Letterale composto di sola lettura

Si noti che un letterale composto è un lvalue e quindi gli elementi possono essere modificabili. È possibile specificare un valore letterale composto di *sola lettura* utilizzando il qualificatore `const` come `(const int[]){1,2}` .

Letterale composto contenente espressioni arbitrarie

All'interno di una funzione, un letterale composto, come per ogni inizializzazione da C99, può avere espressioni arbitrarie.

```
void foo()
{
    int *p;
    int i = 2; j = 5;
    /*...*/
    p = (int [2]){ i+j, i*j };
    /*...*/
}
```

Leggi Letterali composti online: https://riptutorial.com/it/c/topic/4135/letterali-composti

Capitolo 41: Letterali per numeri, caratteri e archi

Osservazioni

Il termine **letterale** è comunemente usato per descrivere una sequenza di caratteri in un codice C che designa un valore costante come un numero (es. `0`) o una stringa (es. `"C"`). Strettamente parlando, lo standard usa la **costante di** termine per le **costanti** intere, le costanti fluttuanti, le costanti di enumerazione e le costanti di carattere, riservando il termine 'letterale' per i letterali stringa, ma questo non è un uso comune.

I letterali possono avere **prefissi** o **suffissi** (ma non entrambi) che sono caratteri extra che possono iniziare o terminare un letterale per cambiare il suo tipo predefinito o la sua rappresentazione.

Examples

Letterali interi

I valori letterali interi vengono utilizzati per fornire valori interi. Sono supportate tre basi numeriche, indicate dai prefissi:

Base	Prefisso	Esempio
Decimale	Nessuna	`5`
Octal	`0`	`0345`
esadecimale	`0x` o `0X`	`0x12AB` , `0X12AB` , `0x12ab` , `0x12Ab`

Nota che questo scritto non include alcun segno, quindi i letterali interi sono sempre positivi. Qualcosa come `-1` è trattato come un'espressione che ha un intero letterale (`1`) che viene annullato con un `-`

Il tipo di un intero decimale letterale è il primo tipo di dati che può adattarsi al valore di `int` e `long` . Dal momento che C99, `long long` è supportato anche per letterali molto grandi.

Il tipo di un letterale intero ottale o esadecimale è il primo tipo di dati che può adattarsi al valore di `int` , `unsigned` , `long` e `unsigned long` . Dal momento che C99, `long long` e `unsigned long long` sono supportati anche per letterali molto grandi.

Utilizzando vari suffissi, il tipo predefinito di un valore letterale può essere modificato.

Suffisso	Spiegazione
`L` , `l`	`long int`
`LL` , `ll` (dal C99)	`long long int`
`U` , `u`	`unsigned`

I suffissi U e L / LL possono essere combinati in qualsiasi ordine e caso. È un errore duplicare i suffissi (ad esempio fornire due suffissi `U`) anche se hanno casi diversi.

Stringhe letterali

I valori letterali stringa vengono utilizzati per specificare matrici di caratteri. Sono sequenze di caratteri racchiusi tra virgolette (ad esempio `"abcd"` e hanno il tipo `char*`).

Il prefisso `L` rende letterale un ampio array di caratteri, di tipo `wchar_t*` . Ad esempio, `L"abcd"` .

Dal momento che C11, ci sono altri prefissi di codifica, simili a `L` :

prefisso	tipo di base	codifica
nessuna	`char`	dipendente dalla piattaforma
`L`	`wchar_t`	dipendente dalla piattaforma
`u8`	`char`	UTF-8
`u`	`char16_t`	di solito UTF-16
`U`	`char32_t`	di solito UTF-32

Per gli ultimi due, può essere interrogato con macro test di funzionalità se la codifica è effettivamente la codifica UTF corrispondente.

Letterali in virgola mobile

I letterali in virgola mobile vengono utilizzati per rappresentare i numeri reali firmati. I seguenti suffissi possono essere usati per specificare il tipo di un letterale:

Suffisso	genere	Esempi
nessuna	`double`	`3.1415926` `-3E6`
`f` , `F`	`float`	`3.1415926f` `2.1E-6F`
`l` , `L`	`long double`	`3.1415926L` `1E126L`

Per utilizzare questi suffissi, il letterale *deve* essere un letterale in virgola mobile. Ad esempio, `3f` è

un errore, poiché `3` è un valore letterale intero, mentre `3.f` o `3.0f` sono corretti. Per il `long double` , la raccomandazione è di usare sempre il capitale `L` per motivi di leggibilità.

Caratteri letterali

I caratteri letterali sono un tipo speciale di letterali interi che vengono utilizzati per rappresentare un carattere. Sono racchiusi tra virgolette singole, ad es. `'a'` e hanno il tipo `int` . Il valore del letterale è un valore intero in base al set di caratteri della macchina. Non consentono i suffissi.

Il prefisso `L` prima di un carattere letterale lo rende un ampio carattere di tipo `wchar_t` . Allo stesso modo, dal momento che i prefissi C11 `u` e `U` rendono ampi caratteri rispettivamente di tipo `char16_t` e `char32_t` .

Quando si intende rappresentare determinati caratteri speciali, come un carattere che non è in stampa, vengono utilizzate le sequenze di escape. Le sequenze di escape usano una sequenza di caratteri che vengono tradotti in un altro personaggio. Tutte le sequenze di escape consistono in due o più caratteri, il primo dei quali è una barra rovesciata `\` . I caratteri che seguono immediatamente il backslash determinano quale carattere letterale viene interpretata come la sequenza.

Sequenza di fuga	Carattere Rappresentato
`\b`	Backspace
`\f`	Modulo di alimentazione
`\n`	Avanzamento riga (nuova riga)
`\r`	Ritorno a capo
`\t`	Scheda orizzontale
`\v`	Scheda verticale
`\\`	Barra rovesciata
`\'`	Virgoletta singola
`\"`	Doppia virgoletta
`\?`	Punto interrogativo
`\nnn`	Valore ottale
`\xnn` ...	Valore esadecimale

C89

Sequenza di fuga	Carattere Rappresentato
`\a`	Avviso (segnale acustico, campanello)

C99

Sequenza di fuga	Carattere Rappresentato
`\unnnn`	Nome del personaggio universale
`\Unnnnnnnn`	Nome del personaggio universale

Un nome di carattere universale è un punto di codice Unicode. Un nome di carattere universale può essere associato a più di un carattere. Le cifre `n` sono interpretate come cifre esadecimali. A seconda della codifica UTF in uso, una sequenza di nomi carattere universale può comportare un punto di codice costituito da più caratteri, anziché un singolo `char` carattere normale.

Quando si utilizza la sequenza di escape dell'alimentazione di riga in I / O in modalità testo, viene convertita nella sequenza di byte o riga di byte di nuova riga specifica del sistema operativo.

La sequenza di escape del punto interrogativo viene utilizzata per evitare i trigraph . Ad esempio, `??/` è compilato come il trigraph che rappresenta un carattere backslash `'\'` , ma usando `?\?/` Risulterebbe nella *stringa* `"??/"` .

Ci possono essere uno, due o tre numeri ottali `n` nella sequenza di escape del valore ottale.

Leggi Letterali per numeri, caratteri e archi online: https://riptutorial.com/it/c/topic/3455/letterali-per-numeri--caratteri-e-archi

Capitolo 42: Liste collegate

Osservazioni

Il linguaggio C non definisce una struttura di dati dell'elenco collegato. Se si utilizza C e si necessita di un elenco collegato, è necessario utilizzare un elenco collegato da una libreria esistente (ad esempio GLib) o scrivere la propria interfaccia dell'elenco collegato. Questo argomento mostra esempi di elenchi collegati e doppi elenchi collegati che possono essere utilizzati come punto di partenza per scrivere i propri elenchi collegati.

Elenco collegato singolarmente

L'elenco contiene nodi che sono composti da un collegamento chiamato next.

Struttura dati

```
struct singly_node
{
  struct singly_node * next;
};
```

Lista doppiamente collegata

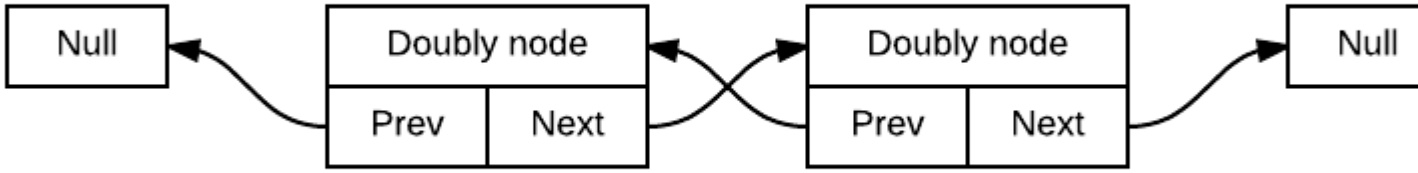
L'elenco contiene nodi composti da due collegamenti chiamati precedente e successivo. I collegamenti fanno normalmente riferimento a un nodo con la stessa struttura.

Struttura dati

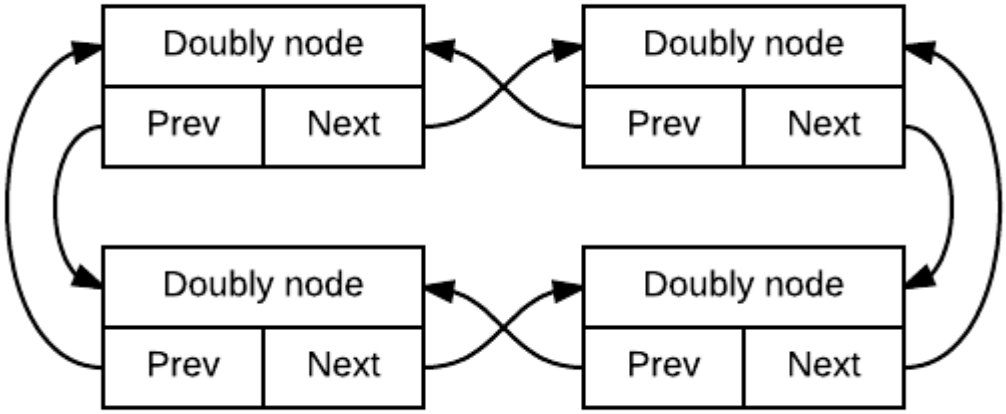
```
struct doubly_node
{
  struct doubly_node * prev;
  struct doubly_node * next;
};
```

Topoliges

Lineare o aperto



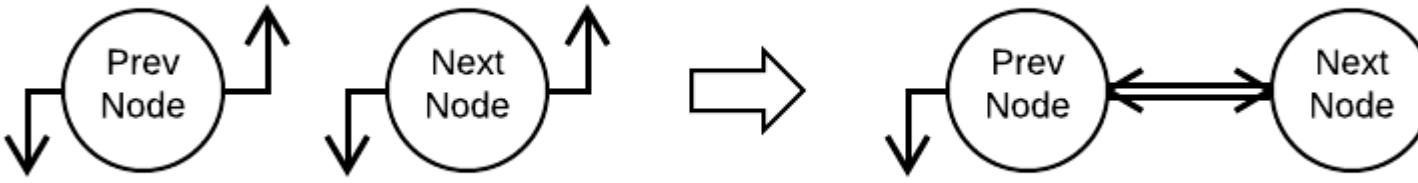
Circolare o anello



procedure

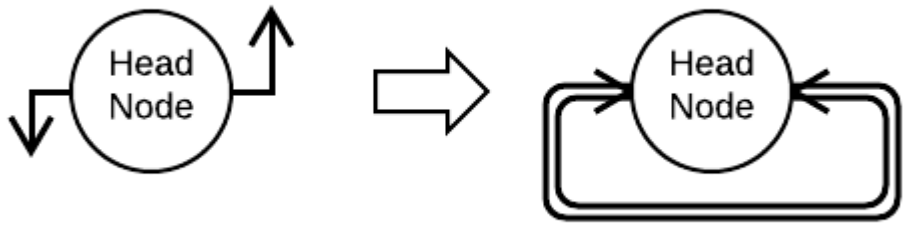
legare

Unisci due nodi insieme.



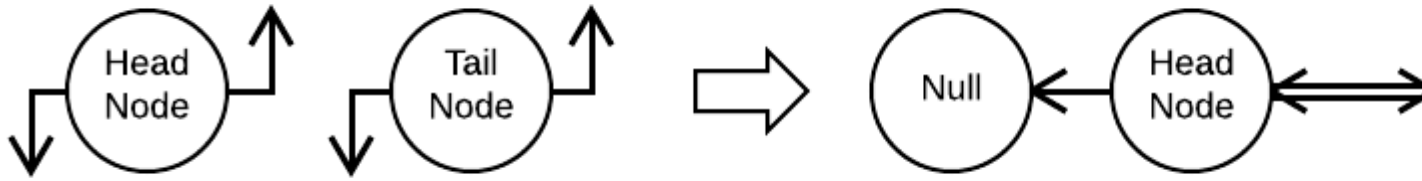

```
void doubly_node_bind (struct doubly_node * prev, struct doubly_node * next)
{
  prev->next = next;
  next->prev = prev;
}
```

Creare un elenco collegato in modo circolare




```
void doubly_node_make_empty_circularly_list (struct doubly_node * head)
{
  doubly_node_bind (head, head);
}
```

Creare un elenco linearmente collegato




```
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);
}
```

Inserimento

Supponiamo che una lista vuota contenga sempre un nodo invece di NULL. Quindi le procedure di inserimento non devono prendere in considerazione NULL.

```
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

Inserimento di un nodo all'inizio di una lista concatenata

Il codice seguente richiederà i numeri e continuerà ad aggiungerli all'inizio di un elenco collegato.

```
/* 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;
}
```

Spiegazione per l'inserimento di nodi

Per capire in che modo aggiungiamo i nodi all'inizio, diamo un'occhiata ai possibili scenari:

1. L'elenco è vuoto, quindi è necessario aggiungere un nuovo nodo. In questo caso, la nostra memoria appare come questa dove `HEAD` è un puntatore al primo nodo:

```
| HEAD | --> NULL
```

La riga `currentNode->next = *headNode;` assegnerà il valore di `currentNode->next` a essere `NULL` poiché `headNode` originariamente parte da un valore di `NULL` .

Ora, vogliamo impostare il nostro puntatore del nodo principale in modo che punti al nostro nodo corrente.

```
 -----      -------------
|HEAD | --> |CURRENTNODE| --> NULL /* The head node points to the current node */
 -----      -------------
```

Questo è fatto con `*headNode = currentNode;`

2. L'elenco è già compilato; dobbiamo aggiungere un nuovo nodo all'inizio. Per semplicità, iniziamo con 1 nodo:

```
-----    -----------
HEAD --> FIRST NODE --> NULL
-----    -----------
```

Con `currentNode->next = *headNode` , la struttura dati appare così:

```
---------        -----    ---------------------
currentNode -->  HEAD --> POINTER TO FIRST NODE --> NULL
---------        -----    ---------------------
```

Che, ovviamente, deve essere modificato poiché `*headNode` dovrebbe puntare a `currentNode` .

```
----    -----------    ---------------
HEAD -> currentNode -->     NODE       -> NULL
----    -----------    ---------------
```

Questo è fatto con `*headNode = currentNode;`

Inserimento di un nodo in corrispondenza dell'ennesimo posto

Finora, abbiamo esaminato l' inserimento di un nodo all'inizio di una lista concatenata . Tuttavia, la maggior parte delle volte vorrai essere in grado di inserire anche i nodi altrove. Il codice scritto qui sotto mostra come è possibile scrivere una funzione `insert()` per inserire nodi *ovunque* negli elenchi collegati.

```
#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;
  }
}
```

Inversione di una lista collegata

È anche possibile eseguire questa attività in modo ricorsivo, ma in questo esempio ho scelto di utilizzare un approccio iterativo. Questa attività è utile se si inseriscono tutti i nodi all'inizio di un elenco collegato . Ecco un esempio:

```
#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;
}
```

Spiegazione per il metodo dell'elenco inverso

Iniziamo il `previousNode` come `NULL` , dato che sappiamo sulla prima iterazione del ciclo, se stiamo cercando il nodo prima del primo nodo head, sarà `NULL` . Il primo nodo diventerà l'ultimo nodo nell'elenco, e la prossima variabile dovrebbe essere naturalmente `NULL` .

Fondamentalmente, il concetto di invertire la lista collegata qui è che in realtà invertiamo i collegamenti stessi. Il prossimo membro di ogni nodo diventerà il nodo prima di esso, in questo modo:

```
Head -> 1 -> 2 -> 3 -> 4 -> 5
```

Dove ogni numero rappresenta un nodo. Questa lista diventerebbe:

```
1 <- 2 <- 3 <- 4 <- 5 <- Head
```

Infine, la testa dovrebbe puntare al quinto nodo, e ogni nodo dovrebbe puntare al nodo precedente.

Il nodo 1 dovrebbe indicare `NULL` poiché non c'era niente prima. Il nodo 2 dovrebbe puntare al nodo 1, il nodo 3 dovrebbe puntare al nodo 2, eccetera.

Tuttavia, c'è *un piccolo problema* con questo metodo. Se interrompiamo il collegamento al nodo successivo e lo sostituiamo con il nodo precedente, non saremo in grado di passare al nodo successivo nell'elenco poiché il collegamento ad esso è scomparso.

La soluzione a questo problema è semplicemente memorizzare l'elemento successivo in una variabile (`nextNode`) prima di modificare il collegamento.

Una lista doppiamente collegata

Un esempio di codice che mostra come i nodi possono essere inseriti in una lista doppiamente collegata, come l'elenco può essere facilmente invertito e come può essere stampato al contrario.

```
#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;
  }
}
```

Si noti che a volte è utile memorizzare un puntatore sull'ultimo nodo (è più efficiente essere semplicemente in grado di saltare direttamente alla fine dell'elenco piuttosto che dover eseguire un'iterazione fino alla fine):

```
struct Node *lastNode = NULL;
```

In tal caso, è necessario aggiornarlo in caso di modifiche all'elenco.

A volte, una chiave viene anche utilizzata per identificare gli elementi. È semplicemente un membro della struttura del nodo:

```
struct Node {
  int data;
  int key;
  struct Node* next;
```

```
  struct Node* previous;
};
```

La chiave viene quindi utilizzata quando qualsiasi attività viene eseguita su un elemento specifico, ad esempio l'eliminazione di elementi.

Leggi Liste collegate online: https://riptutorial.com/it/c/topic/560/liste-collegate

Capitolo 43: Matematica standard

Sintassi

- #include <math.h>
- doppio pow (doppio x, doppio y);
- float powf (float x, float y);
- lungo doppio doppio (lungo doppio x, lungo doppio y);

Osservazioni

1. Per collegarsi con la libreria matematica usa `-lm` con i flag gcc.
2. Un programma portatile che deve verificare un errore da una funzione matematica dovrebbe impostare `errno` su zero e effettuare la seguente chiamata `feclearexcept(FE_ALL_EXCEPT);` prima di chiamare una funzione matematica. Al ritorno dalla funzione matematica, se `errno` è `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW);` zero, o la seguente chiamata restituisce nonzero `fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW);` quindi si è verificato un errore nella funzione matematica. Leggi la pagina di manuale di math_error per maggiori informazioni.

Examples

Resto di virgola mobile a doppia precisione: fmod ()

Questa funzione restituisce il resto in virgola mobile della divisione di `x/y` . Il valore restituito ha lo stesso segno di 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;
}
```

Produzione:

```
4.90000
```

Importante: utilizzare questa funzione con attenzione, poiché può restituire valori imprevisti a

causa dell'operazione dei valori in virgola mobile.

```
#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;
}
```

Produzione:

```
0.1
0.09999999999999995
```

Rifinitura a virgola mobile a precisione singola e lunga precisione doppia: fmodf (), fmodl ()

C99

Queste funzioni restituiscono il resto in virgola mobile della divisione di `x/y` . Il valore restituito ha lo stesso segno di x.

Singola precisione:

```
#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. */
}
```

Produzione:

```
4.90000
```

Doppia precisione doppia:

```
#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. */
}
```

Produzione:

```
4.90000
```

Funzioni di potenza: pow (), powf (), powl ()

Il seguente codice di esempio calcola la somma di *1 + 4 (3 + 3 ^ 2 + 3 ^ 3 + 3 ^ 4 + ... + 3 ^ N)* serie usando la famiglia pow () della libreria matematica standard.

```
#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;
}
```

Esempio di output:

```
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
```

Leggi Matematica standard online: https://riptutorial.com/it/c/topic/3170/matematica-standard

Capitolo 44: multithreading

introduzione

In C11 c'è una libreria di thread standard, `<threads.h>` , ma nessun compilatore conosciuto che lo implementa. Pertanto, per utilizzare il multithreading in C è necessario utilizzare implementazioni specifiche della piattaforma come la libreria di thread POSIX (spesso denominata pthreads) utilizzando l'intestazione `pthread.h` .

Sintassi

- thrd_t // Tipo di oggetto completo definito dall'implementazione che identifica una discussione
- int thrd_create (thrd_t * thr, thrd_start_t func, void * arg); // Crea una discussione
- int thrd_equal (thrd_t thr0, thrd_t thr1); // Controlla se gli argomenti si riferiscono allo stesso thread
- thr_t thrd_current (void); // Restituisce l'identificatore del thread che lo chiama
- int th_sleep (const struct timespec * duration, struct timespec * restante); // Sospende l'esecuzione del thread di chiamata per almeno un determinato periodo
- void thrd_yield (void); // Permette ad altri thread di essere eseguiti al posto del thread che lo chiama
- _Noreturn void thrd_exit (int res); // Termina il thread il thread che lo chiama
- int thrd_detatch (thrd_t thr; // Stacca un dato thread dall'ambiente corrente
- int thrd_join (thrd_t thr, int * res); // Blocca il thread corrente fino al termine del thread specificato

Osservazioni

L'utilizzo dei thread può introdurre comportamenti non definiti extra come http://www.Scriptutorial.com/c/example/2622/data-race . Per l'accesso senza `mtx_lock()` alle variabili condivise tra diversi thread C11 fornisce la funzionalità mutex `mtx_lock()` o i tipi di dati (opzionali) http://www.riptutorial.com/c/topic/4924/atomics e le funzioni associate in `stdatomic.h` .

Examples

Esempio di thread C11 semplice

```
#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);
}
```

Leggi multithreading online: https://riptutorial.com/it/c/topic/10489/multithreading

Capitolo 45: operatori

introduzione

Un operatore in un linguaggio di programmazione è un simbolo che indica al compilatore o all'interprete di eseguire un'operazione matematica, relazionale o logica specifica e di produrre un risultato finale.

C ha molti potenti operatori. Molti operatori C sono operatori binari, il che significa che hanno due operandi. Ad esempio, in `a / b` , `/` è un operatore binario che accetta due operandi (`a` , `b`). Esistono alcuni operatori unari che accettano un operando (ad esempio: `~` , `++`) e un solo operatore ternario `? :`

Sintassi

- operatore expr1
- operatore expr2
- expr1 operator expr2
- expr1? expr2: expr3

Osservazioni

Gli operatori hanno *un'arità* , una *precedenza* e *un'associatività* .

- *Arity* indica il numero di operandi. In C esistono tre diverse organizzazioni di operatori:
    - Unario (1 operando)
    - Binario (2 operandi)
    - Ternario (3 operandi)
- *Precedenza* indica quali operatori "eseguono il bind" prima dei loro operandi. Cioè, quale operatore ha la priorità di operare sui suoi operandi. Ad esempio, il linguaggio C obbedisce alla convenzione secondo cui la moltiplicazione e la divisione hanno la precedenza sull'addizione e sulla sottrazione:

    ```
    a * b + c
    ```

    Dà lo stesso risultato di

    ```
    (a * b) + c
    ```

    Se questo non è ciò che si voleva, la precedenza può essere forzata usando le parentesi, poiché hanno la precedenza *più alta* di tutti gli operatori.

```
a * (b + c)
```

Questa nuova espressione produrrà un risultato che differisce dalle precedenti due espressioni.

Il linguaggio C ha molti livelli di precedenza; Di seguito viene riportata una tabella di tutti gli operatori, in ordine decrescente di precedenza.

Tabella di precedenza

operatori	Associatività
`() [] -> .`	da sinistra a destra
`! ~ ++ -- + - *` (dereferenziazione) `(type) sizeof`	da destra a sinistra
`*` (moltiplicazione) `/ %`	da sinistra a destra
`+ -`	da sinistra a destra
`<< >>`	da sinistra a destra
`< <= > >=`	da sinistra a destra
`== !=`	da sinistra a destra
`&`	da sinistra a destra
`^`	da sinistra a destra
`|`	da sinistra a destra
`&&`	da sinistra a destra
`||`	da sinistra a destra
`?:`	da destra a sinistra
`= += -= *= /= %= &= ^= |= <<= >>=`	da destra a sinistra
`,`	da sinistra a destra

- *L'associatività* indica come gli operatori con precedenza uguale si legano per impostazione predefinita e sono disponibili due tipi: da *sinistra a destra* e da *destra a sinistra* . Un esempio di associazione da *sinistra a destra* è l'operatore di sottrazione (`-`). L'espressione

```
a - b - c - d
```

ha tre sottrazioni di precedenza identiche, ma dà lo stesso risultato di

```
((a - b) - c) - d
```

perché il più a sinistra `-` si lega prima ai suoi due operandi.

Un esempio di associatività da *destra a sinistra* sono gli operatori dereferenzia `*` e post-incremento `++` . Entrambi hanno la stessa precedenza, quindi se sono usati in un'espressione come

```
* ptr ++
```

, questo è equivalente a

```
* (ptr ++)
```

perché l'operatore all'estrema destra (`++`) si lega innanzitutto al suo unico operando.

Examples

Operatori relazionali

Gli operatori relazionali verificano se una relazione specifica tra due operandi è vera. Il risultato viene valutato su 1 (che significa *vero*) o 0 (che significa *falso*). Questo risultato viene spesso utilizzato per influenzare il flusso di controllo (tramite `if` , `while` , `for`), ma può anche essere memorizzato in variabili.

Uguale a "=="

Controlla se gli operandi forniti sono uguali.

```
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.
*/
```

Attenzione: questo operatore non deve essere confuso con l'operatore di assegnazione (`=`)!

Non uguale a "! ="

Controlla se gli operandi forniti non sono uguali.

```
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.
*/
```

Questo operatore restituisce in modo efficace il risultato opposto a quello dell'operatore di uguale (`==`).

Non "!"

Controlla se un oggetto è uguale a `0` .

Il `!` può anche essere usato direttamente con una variabile come segue:

```
!someVal
```

Questo ha lo stesso effetto di:

```
someVal == 0
```

Maggiore di ">"

Controlla se l'operando di sinistra ha un valore maggiore rispetto all'operando di destra

```
5 > 4      /* evaluates to 1. */
4 > 5      /* evaluates to 0. */
4 > 4      /* evaluates to 0. */
```

Meno di "<"

Controlla se l'operando di sinistra ha un valore inferiore rispetto all'operando di destra

```
5 < 4      /* evaluates to 0. */
4 < 5      /* evaluates to 1. */
4 < 4      /* evaluates to 0. */
```

Maggiore o uguale "> ="

Controlla se l'operando di sinistra ha un valore maggiore o uguale all'operando di destra.

```
5 >= 4      /* evaluates to 1. */
4 >= 5      /* evaluates to 0. */
4 >= 4      /* evaluates to 1. */
```

Minore o uguale "<="

Controlla se l'operando di sinistra ha un valore inferiore o uguale all'operando di destra.

```
5 <= 4      /* evaluates to 0. */
4 <= 5      /* evaluates to 1. */
4 <= 4      /* evaluates to 1. */
```

Operatori di assegnazione

Assegna il valore dell'operando di destra alla posizione di memoria indicata dall'operando di sinistra e restituisce il valore.

```
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'. */
```

Diverse operazioni aritmetiche hanno un operatore di *assegnazione composto* .

```
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 */
```

Una caratteristica importante di questi compiti composti è che l'espressione sul lato sinistro (`a`) viene valutata solo una volta. Ad esempio se `p` è un puntatore

```
*p += 27;
```

dereferenze `p` una sola volta, mentre il seguente lo fa due volte.

```
*p = *p + 27;
```

Va anche notato che il risultato di un compito come `a = b` è ciò che è noto come un *valore* . Pertanto, l'assegnazione ha effettivamente un valore che può quindi essere assegnato a un'altra variabile. Ciò consente il concatenamento di assegnazioni per impostare più variabili in una singola istruzione.

Questo *valore* può essere utilizzato nelle espressioni di controllo delle istruzioni `if` (o cicli o istruzioni `switch`) che proteggono il codice sul risultato di un'altra espressione o chiamata di

funzione. Per esempio:

```
char *buffer;
if ((buffer = malloc(1024)) != NULL)
{
    /* do something with buffer */
    free(buffer);
}
else
{
    /* report allocation failure */
}
```

Per questo motivo, è necessario prestare attenzione per evitare un errore di battitura comune che può portare a bug misteriosi.

```
int a = 2;
/* ... */
if (a = 1)
    /* Delete all files on my hard drive */
```

Ciò avrà risultati disastrosi, poiché `a = 1` valuterà sempre `1` e quindi l'espressione di controllo dell'istruzione `if` sarà sempre vera (leggi di più su questo errore comune qui). L'autore quasi certamente intendeva usare l'operatore di uguaglianza (`==`) come mostrato di seguito:

```
int a = 2;
/* ... */
if (a == 1)
    /* Delete all files on my hard drive */
```

Associatività dell'operatore

```
int a, b = 1, c = 2;
a = b = c;
```

Questo assegna `c` a `b` , che restituisce `b` , che è assegnato a `a` . Ciò accade perché tutti gli operatori di assegnazione hanno una associatività corretta, vale a dire che l'operazione più a destra nell'espressione viene valutata per prima e procede da destra a sinistra.

Operatori aritmetici

Aritmetica di base

Restituisce un valore che è il risultato dell'applicazione dell'operando della mano sinistra all'operando di destra, usando l'operazione matematica associata. Si applicano le normali regole matematiche di commutazione (cioè l'addizione e la moltiplicazione sono commutative, la sottrazione, la divisione e il modulo non lo sono).

Addition Operator

L'operatore di addizione (`+`) viene utilizzato per aggiungere due operandi insieme. Esempio:

```
#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;
}
```

Operatore di sottrazione

L'operatore di sottrazione (`-`) viene utilizzato per sottrarre il secondo operando dal primo. Esempio:

```
#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;
}
```

Operatore di moltiplicazione

L'operatore di moltiplicazione (`*`) viene utilizzato per moltiplicare entrambi gli operandi. Esempio:

```
#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;
}
```

Da non confondere con l'operatore `*` dereferenziazione.

Operatore di divisione

L'operatore di divisione (`/`) divide il primo operando del secondo. Se entrambi gli operandi della divisione sono numeri interi, restituirà un valore intero e scarterà il resto (utilizzare l'operatore modulo `%` per calcolare e acquisire il resto).

Se uno degli operandi è un valore in virgola mobile, il risultato è un'approssimazione della frazione.

Esempio:

```
#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;
}
```

Operatore di modulo

L'operatore modulo (`%`) riceve solo operandi interi e viene utilizzato per calcolare il resto dopo che il primo operando è stato diviso per il secondo. Esempio:

```
#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;
}
```

Operatori di incremento / decremento

Gli operatori di incremento (`a++`) e di decremento (`a--`) sono diversi in quanto modificano il valore della variabile a cui li si applica senza un operatore di assegnazione. È possibile utilizzare gli operatori di incremento e decremento prima o dopo la variabile. Il posizionamento dell'operatore modifica i tempi di incremento / decremento del valore prima o dopo l'assegnazione alla variabile. Esempio:

```
#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 */
    }
}
```

Come nell'esempio per `c` e `d` , entrambi gli operatori hanno due forme, come notazione prefisso e notazione postfissa. Entrambi hanno lo stesso effetto nell'incrementare (`++`) o decrementare (`--`) la variabile, ma differiscono in base al valore che restituiscono: le operazioni di prefisso eseguono prima l'operazione e quindi restituiscono il valore, mentre le operazioni postfissa determinano prima il valore che deve essere restituito, quindi eseguire l'operazione.

A causa di questo comportamento potenzialmente contro-intuitivo, l'uso di operatori di incremento / decremento all'interno di espressioni è controverso.

Operatori logici

AND logico

Esegue un AND logico booleano dei due operandi restituendo 1 se entrambi gli operandi sono

diversi da zero. L'operatore AND logico è di tipo `int` .

```
0 && 0  /* Returns 0. */
0 && 1  /* Returns 0. */
2 && 0  /* Returns 0. */
2 && 3  /* Returns 1. */
```

OR logico

Esegue un OR logico booleano dei due operandi restituendo 1 se uno degli operandi è diverso da zero. L'operatore logico OR è di tipo `int` .

```
0 || 0  /* Returns 0. */
0 || 1  /* Returns 1.  */
2 || 0  /* Returns 1.  */
2 || 3  /* Returns 1.  */
```

NOT logico

Esegue una negazione logica. L'operatore logico NOT è di tipo `int` . L'operatore NOT verifica se almeno un bit è uguale a 1, in tal caso restituisce 0. Altrimenti restituisce 1;

```
!1 /* Returns 0. */
!5 /* Returns 0. */
!0 /* Returns 1. */
```

Valutazione del cortocircuito

Ci sono alcune proprietà cruciali comuni a `&&` e `||` :

- l'operando di sinistra (LHS) è completamente valutato prima di valutare l'operando di destra (RHS),
- c'è un punto di sequenza tra la valutazione dell'operando di sinistra e l'operando di destra,
- e, soprattutto, l'operando di destra non viene valutato affatto se il risultato dell'operando di sinistra determina il risultato complessivo.

Ciò significa che:

- se l'LHS valuta "vero" (diverso da zero), l'RHS di `||` non sarà valutato (perché il risultato di 'true OR anything' è 'true'),
- se l'LHS restituisce 'false' (zero), l'RHS di `&&` non verrà valutato (perché il risultato di 'false AND anything' è 'false').

Questo è importante in quanto ti permette di scrivere codice come:

```
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";
}
```

Se un valore negativo viene passato alla funzione, il `value >= 0` termine `value < NUM_NAMES` false e il `value < NUM_NAMES` non viene valutato.

Incrementa / Decrementa

Gli operatori di incremento e decremento esistono in forma *prefissa* e *postfix* .

```
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 */
```

Nota che le operazioni aritmetiche non introducono punti di sequenza , quindi certe espressioni con `++` o `--` operatori possono introdurre un comportamento indefinito .

Operatore condizionale / Operatore ternario

Valuta il suo primo operando e, se il valore risultante non è uguale a zero, valuta il suo secondo operando. Altrimenti, valuta il suo terzo operando, come mostrato nell'esempio seguente:

```
a = b ? c : d;
```

è equivalente a:

```
if (b)
    a = c;
else
    a = d;
```

Questo pseudo-codice lo rappresenta: `condition ? value_if_true : value_if_false` . Ogni valore può essere il risultato di un'espressione valutata.

```
int x = 5;
int y = 42;
printf("%i, %i\n", 1 ? x : y, 0 ? x : y); /* Outputs "5, 42" */
```

L'operatore condizionale può essere annidato. Ad esempio, il seguente codice determina il più grande dei tre numeri:

```
big= a > b ? (a > c ? a : c)
           : (b > c ? b : c);
```

L'esempio seguente scrive anche interi in un file e interi dispari in un altro file:

```
#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;
}
```

L'operatore condizionale si associa da destra a sinistra. Considera quanto segue:

```
exp1 ? exp2 : exp3 ? exp4 : exp5
```

Poiché l'associazione va da destra a sinistra, l'espressione sopra è valutata come

```
exp1 ? exp2 : ( exp3 ? exp4 : exp5 )
```

Operatore di virgola

Valuta il suo operando di sinistra, scarta il valore risultante e quindi valuta l'operando dei diritti e il risultato restituisce il valore dell'operando più a destra.

```
int x = 42, y = 42;
printf("%i\n", (x *= 2, y)); /* Outputs "42". */
```

L'operatore virgola introduce un punto di sequenza tra i suoi operandi.

Si noti che la *virgola* utilizzata nelle funzioni chiama che gli argomenti separati NON è l' *operatore virgola* , piuttosto è chiamato un *separatore* che è diverso dall'operatore *virgola* . Quindi, non ha le proprietà *dell'operatore virgola* .

La precedente chiamata `printf()` contiene sia l' *operatore virgola che* il *separatore* .

```
printf("%i\n", (x *= 2, y)); /* Outputs "42". */
/*           ^        ^ this is a comma operator */
/*           this is a separator */
```

L'operatore virgola viene spesso utilizzato nella sezione di inizializzazione e nella sezione di aggiornamento di un ciclo `for` . Per esempio:

```
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 Operatore

Esegue una conversione *esplicita* nel tipo specificato dal valore risultante dalla valutazione dell'espressione specificata.

```
int x = 3;
int y = 4;
printf("%f\n", (double)x / y); /* Outputs "0.750000". */
```

Qui il valore di `x` viene convertito in un `double` , la divisione promuove anche il valore di `y` per `double` , e il risultato della divisione, un `double` viene passato a `printf` per la stampa.

sizeof Operatore

Con un tipo come operando

Valuta nella dimensione in byte, di tipo `size_t` , di oggetti del tipo specificato. Richiede parentesi attorno al tipo.

```
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! */
```

Con un'espressione come operando

Valuta nella dimensione in byte, di tipo `size_t` , di oggetti del tipo dell'espressione data. L'espressione stessa non viene valutata. Le parentesi non sono richieste; tuttavia, poiché l'espressione data deve essere unaria, è consigliabile utilizzarla sempre.

```
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. */
```

Puntatore aritmetico

Aggiunta puntatore

Dato un puntatore e un tipo scalare `N` , valuta in un puntatore il `N` elemento del tipo puntato che riesce direttamente nell'oggetto puntato in memoria.

```
int arr[] = {1, 2, 3, 4, 5};
printf("*(arr + 3) = %i\n", *(arr + 3)); /* Outputs "4", arr's fourth element. */
```

Non importa se il puntatore viene usato come valore dell'operando o come valore scalare. Ciò significa che cose come `3 + arr` sono valide. Se `arr[k]` è il membro `k+1` di un array, quindi `arr+k` è un puntatore a `arr[k]` . In altre parole, `arr` o `arr+0` è un puntatore a `arr[0]` , `arr+1` è un puntatore a `arr[2]` , e così via. In generale, `*(arr+k)` è uguale a `arr[k]` .

Diversamente dalla solita aritmetica, l'aggiunta di `1` a un puntatore a un `int` aggiungerà `4` byte al valore dell'indirizzo corrente. Poiché i nomi degli array sono puntatori costanti, `+` è l'unico operatore che possiamo usare per accedere ai membri di un array tramite la notazione del puntatore usando il nome dell'array. Tuttavia, definendo un puntatore a un array, possiamo ottenere una maggiore flessibilità per elaborare i dati in un array. Ad esempio, possiamo stampare i membri di un array come segue:

```
#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;
}
```

Definendo un puntatore all'array, il programma di cui sopra è equivalente al seguente:

```
#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;
}
```

Vedi che i membri dell'array `arr` sono accessibili usando gli operatori `+` e `++` . Gli altri operatori che possono essere utilizzati con il puntatore `ptr` sono `-` e `--` .

Sottrazione puntatore

Dato due puntatori allo stesso tipo, valuta in un oggetto di tipo `ptrdiff_t` che contiene il valore scalare che deve essere aggiunto al secondo puntatore per ottenere il valore del primo puntatore.

```
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". */
```

Operatori di accesso

Gli operatori di accesso utente (dot `.` E freccia `->`) vengono utilizzati per accedere a un membro di una `struct` .

Membro di oggetto

Valuta nel lvalue che indica l'oggetto che è un membro dell'oggetto accesso.

```
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". */
```

Membro di oggetto appuntito

Zucchero sintattico per il dereferenziamento seguito dall'accesso dei membri. In effetti, un'espressione della forma `x->y` è una scorciatoia per `(*x).y` - ma l'operatore della freccia è molto più chiaro, specialmente se i puntatori della struttura sono nidificati.

```
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". */
```

Indirizzo-di

L'unario `&` operatore è l'indirizzo dell'operatore. Valuta l'espressione data, in cui l'oggetto risultante deve essere un lvalue. Quindi, valuta in un oggetto il cui tipo è un puntatore al tipo di oggetto risultante e contiene l'indirizzo dell'oggetto risultante.

```
int x = 3;
int *p = &x;
printf("%p = %p\n", (void *)&x, (void *)p); /* Outputs "A = A", for some implementation-
defined A. */
```

dereference

L'operatore unario `*` denota un puntatore. Valuta nel lvalue risultante dal dereferenziamento del puntatore che risulta dalla valutazione dell'espressione data.

```
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". */
```

indicizzazione

L'indicizzazione è zucchero sintattico per l'aggiunta del puntatore seguita dal dereferenziamento. In effetti, un'espressione della forma `a[i]` è equivalente a `*(a + i)` - ma la notazione di indice esplicito è preferita.

```
int arr[] = { 1, 2, 3, 4, 5 };
printf("arr[2] = %i\n", arr[2]); /* Outputs "arr[2] = 3". */
```

Intercambiabilità dell'indicizzazione

L'aggiunta di un puntatore a un numero intero è un'operazione commutativa (ovvero l'ordine degli operandi non modifica il risultato) quindi `pointer + integer == integer + pointer` .

Una conseguenza di ciò è che `arr[3]` e `3[arr]` sono equivalenti.

```
printf("3[arr] = %i\n", 3[arr]); /* Outputs "3[arr] = 4". */
```

L'uso di un'espressione `3[arr]` invece di `arr[3]` è generalmente raccomandato, in quanto influisce sulla leggibilità del codice. Tende ad essere popolare nei concorsi di programmazione offuscati.

Operatore di chiamata di funzione

Il primo operando deve essere un puntatore di funzione (anche un designatore di funzione è accettabile perché verrà convertito in un puntatore alla funzione), identificando la funzione da chiamare e tutti gli altri operandi, se presenti, sono noti collettivamente come argomenti della chiamata di funzione . Valuta nel valore restituito risultante dal richiamo della funzione appropriata con i rispettivi argomenti.

```
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 */
```

Operatori bit a bit

Gli operatori bit a bit possono essere utilizzati per eseguire operazioni a livello di bit su variabili. Di seguito è riportato un elenco di tutti e sei gli operatori bit a bit supportati in C:

Simbolo	Operatore
&	bit a bit AND
|	bit a bit compreso OR
^	OR esclusivo bit a bit (XOR)
~	bit a bit no (il proprio complemento)
<<	spostamento logico sinistro
>>	spostamento logico giusto

Il seguente programma illustra l'uso di tutti gli operatori bit a bit:

```
#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;
}
```

Le operazioni bit a bit con i tipi firmati dovrebbero essere evitate perché il bit di segno di tale rappresentazione bit ha un significato particolare. Restrizioni particolari si applicano agli operatori di turno:

- Lo spostamento a sinistra di 1 bit nel bit firmato è errato e porta a un comportamento indefinito.
- Il corretto spostamento di un valore negativo (con il bit di segno 1) è definito dall'implementazione e quindi non è trasferibile.
- Se il valore dell'operando di destra di un operatore di spostamento è negativo o è maggiore o uguale alla larghezza dell'operando di sinistra promosso, il comportamento non è definito.

Masking:

Il mascheramento si riferisce al processo di estrazione dei bit desiderati (o alla trasformazione dei bit desiderati in) di una variabile utilizzando operazioni logiche bit a bit. L'operando (una costante o variabile) utilizzato per eseguire il mascheramento è chiamato *maschera* .

Il mascheramento è utilizzato in molti modi diversi:

- Per decidere il modello di bit di una variabile intera.
- Per copiare una parte di un dato pattern di bit in una nuova variabile, mentre il resto della nuova variabile viene riempita con 0s (usando bitwise AND)
- Per copiare una porzione di un dato pattern di bit in una nuova variabile, mentre il resto della nuova variabile viene riempita con 1s (usando OR bit a bit).
- Per copiare una porzione di un dato pattern di bit in una nuova variabile, mentre il resto del pattern di bit originale viene invertito all'interno della nuova variabile (usando OR esclusivo bit per bit).

La seguente funzione utilizza una maschera per visualizzare il modello di bit di una variabile:

```
#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

Esegue una query sul requisito di allineamento per il tipo specificato. Il requisito di allineamento è una potenza integrale positiva di 2 che rappresenta il numero di byte tra i quali possono essere allocati due oggetti del tipo. In C, il requisito di allineamento è misurato in `size_t` .

Il nome del tipo potrebbe non essere un tipo incompleto né un tipo di funzione. Se viene utilizzato un array come tipo, viene utilizzato il tipo di elemento dell'array.

Questo operatore è spesso accessibile tramite il pratico `alignof` macro di `<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;}));
}
```

Uscita possibile:

```
Alignment of char = 1
Alignment of max_align_t = 16
alignof(float[10]) = 4
alignof(struct{char c; int n;}) = 4
```

http://en.cppreference.com/w/c/language/_Alignof

Comportamento di cortocircuito degli operatori logici

Il cortocircuito è una funzionalità che salta la valutazione delle parti di una condizione (se / mentre / ...) quando possibile. Nel caso di un'operazione logica su due operandi, il primo operando viene valutato (vero o falso) e se c'è un verdetto (cioè il primo operando è falso quando si usa &&, il

primo operando è vero quando si usa ||) il secondo operando è non valutato.

Esempio:

```
#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;
}
```

Dai un'occhiata a te stesso:

```
#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;
}
```

Produzione:

```
$ ./a.out
print function 20
I will be printed!
```

Il cortocircuito è importante, quando si vuole evitare di valutare termini che sono (computazionalmente) costosi. Inoltre, può influire pesantemente sul flusso del tuo programma come in questo caso: Perché questo programma stampa "forked!" 4 volte?

Leggi operatori online: https://riptutorial.com/it/c/topic/256/operatori

Capitolo 46: Parametri di funzione

Osservazioni

In C, è comune utilizzare i valori di ritorno per indicare gli errori che si verificano; e per restituire i dati tramite l'uso dei puntatori passati. Questo può essere fatto per molteplici ragioni; incluso non dover allocare memoria sullo heap o utilizzare l'allocazione statica nel punto in cui viene chiamata la funzione.

Examples

Utilizzo dei parametri del puntatore per restituire più valori

Un modello comune in C, per imitare facilmente la restituzione di più valori da una funzione, è utilizzare i puntatori.

```
#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;
}
```

Passare da array a funzioni

```
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;
  }
}
```

Qui la `static` all'interno di `[]` del parametro function, richiede che la matrice di argomenti abbia almeno tanti elementi quanti sono specificati (es. Elementi di `size`). Per essere in grado di utilizzare tale funzione, dobbiamo assicurarci che il parametro `size` venga prima del parametro array nell'elenco.

Usa `getListOfFriends()` questo modo:

```
#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;
}
```

Guarda anche

Passaggio di array multidimensionali a una funzione

I parametri sono passati per valore

In C, tutti i parametri di funzione vengono passati per valore, pertanto la modifica di ciò che viene passato nelle funzioni di chiamata non influisce sulle variabili locali delle funzioni del chiamante.

```
#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;
}
```

È possibile utilizzare i puntatori per consentire alle funzioni callee di modificare le variabili locali delle funzioni del chiamante. Si noti che questo non è *passato per riferimento,* ma i *valori* del puntatore *che* puntano alle variabili locali vengono passati.

```
#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;
}
```

Tuttavia, la restituzione dell'indirizzo di una variabile locale al callee comporta un comportamento indefinito. Vedi Dereferenziare un puntatore alla variabile oltre la sua durata .

Ordine di esecuzione del parametro funzione

L'ordine di esecuzione dei parametri non è definito nella programmazione C. Qui può essere eseguito da sinistra a destra o da destra a sinistra. L'ordine dipende dall'implementazione.

```
#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;
}
```

Esempio di funzione che restituisce una struct contenente valori con codici di errore

La maggior parte degli esempi di una funzione che restituisce un valore implica la fornitura di un puntatore come uno degli argomenti per consentire alla funzione di modificare il valore puntato a, simile al seguente. Il valore di ritorno effettivo della funzione è solitamente un tipo, ad esempio un `int` per indicare lo stato del risultato, indipendentemente dal fatto che abbia funzionato o meno.

```
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 */
}
```

Tuttavia, è possibile utilizzare anche una `struct` come valore di ritorno che consente di restituire sia uno stato di errore insieme ad altri valori. Per esempio.

```
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;
}
```

Questa funzione potrebbe quindi essere utilizzata come il seguente esempio.

```
int usingFunc (int iValue)
{
    RetValue iRet = func (iValue);

    if (iRet.iStat == 1) {
        /* do things with iRet.iValue, the returned value */
    }
    return 0;
}
```

O potrebbe essere usato come il seguente.

```
int usingFunc (int iValue)
{
    RetValue iRet;

    if ( (iRet = func (iValue)).iStat == 1 ) {
        /* do things with iRet.iValue, the returned value */
    }
    return 0;
}
```

Leggi Parametri di funzione online: https://riptutorial.com/it/c/topic/1006/parametri-di-funzione

Capitolo 47: Passa le matrici 2D alle funzioni

Examples

Passa un array 2D a una funzione

Passare un array 2d a una funzione sembra semplice e ovvio e scriviamo felicemente:

```
#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]);
    }
  }
}
```

Ma il compilatore, qui GCC nella versione 4.9.4, non lo apprezza bene.

```
$ 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);
```

Le ragioni di ciò sono duplici: il problema principale è che gli array non sono puntatori e il secondo inconveniente è il cosiddetto *decadimento del puntatore* . Passare un array a una funzione decomporrà l'array a un puntatore al primo elemento dell'array - nel caso di un array 2d esso decadrà in un puntatore alla prima riga perché nei array C viene ordinato riga-prima.

```
#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]);
    }
  }
}
```

È necessario passare il numero di righe, non possono essere calcolate.

```
#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

Il numero di colonne è predefinito e quindi fissato in fase di compilazione, ma il predecessore dell'attuale standard C (che era ISO / IEC 9899: 1999, corrente ISO / IEC 9899: 2011) implementava VLA (TODO: collegalo) e sebbene lo standard attuale lo abbia reso opzionale, quasi tutti i moderni compilatori C lo supportano (TODO: controlla se MS Visual Studio lo supporta ora).

```
#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]);
    }
  }
}
```

Questo non funziona, il compilatore si lamenta:

```
$ 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]);
```

Diventa un po 'più chiaro se intenzionalmente `void fun1(int **a, int rows, int cols)` un errore nella chiamata della funzione cambiando la dichiarazione in `void fun1(int **a, int rows, int cols)` . Ciò fa sì che il compilatore si lamenti in un modo diverso, ma ugualmente nebuloso

```
$ 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);
```

Possiamo reagire in diversi modi, uno dei quali è ignorarlo e fare qualche giocoleria puntatore illeggibile:

```
#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)));
    }
  }
}
```

O lo facciamo bene e passiamo le informazioni necessarie a `fun1` . Per fare ciò, è necessario riorganizzare gli argomenti su `fun1` : la dimensione della colonna deve precedere la dichiarazione dell'array. Per tenerlo più leggibile, anche la variabile che contiene il numero di righe ha cambiato la sua posizione ed è la prima.

```
#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]);
    }
  }
}
```

Questo sembra imbarazzante per alcune persone, che ritengono che l'ordine delle variabili non dovrebbe avere importanza. Questo non è un grosso problema, basta dichiarare un puntatore e lasciarlo puntare alla matrice.

```
#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]);
    }
  }
}
```

Utilizzo di array flat come array 2D

Spesso la soluzione più semplice è semplicemente passare gli array 2D e superiori in giro come memoria piatta.

```
/* 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;
        }
```

```
    }
}
```

Leggi Passa le matrici 2D alle funzioni online: https://riptutorial.com/it/c/topic/6862/passa-le-matrici-2d-alle-funzioni

Capitolo 48: Preprocessore e macro

introduzione

Tutti i comandi del preprocessore iniziano con il simbolo `#` cancelletto (cancelletto). La macro AC è solo un comando del preprocessore che viene definito utilizzando la direttiva `#define` preprocessore. Durante la fase di pre-elaborazione, il preprocessore C (una parte del compilatore C) sostituisce semplicemente il corpo della macro ovunque appaia il suo nome.

Osservazioni

Quando un compilatore incontra una macro nel codice, esegue la sostituzione semplice della stringa, senza eseguire ulteriori operazioni. Per questo motivo, le modifiche apportate dal preprocessore non rispettano l'ambito dei programmi C: ad esempio, una definizione di macro non è limitata all'essere all'interno di un blocco, quindi non è influenzata da un `'}'` che termina un'istruzione di blocco.

Il preprocessore è, per impostazione, non completo - esistono diversi tipi di calcolo che non possono essere eseguiti dal solo preprocessore.

Di solito i compilatori hanno un flag a riga di comando (o impostazione di configurazione) che ci consente di interrompere la compilazione dopo la fase di pre-elaborazione e di ispezionare il risultato. Su piattaforme POSIX questo flag è `-E` . Quindi, l'esecuzione di gcc con questo flag stampa il codice sorgente espanso su stdout:

```
$ gcc -E cprog.c
```

Spesso il preprocessore viene implementato come un programma separato, che viene richiamato dal compilatore, il nome comune per quel programma è `cpp` . Un numero di preprocessori emette informazioni di supporto, come informazioni sui numeri di riga, che viene utilizzato dalle fasi successive della compilazione per generare informazioni di debug. Nel caso in cui il preprocessore sia basato su gcc, l'opzione -P sopprime tali informazioni.

```
$ cpp -P cprog.c
```

Examples

Inclusione condizionale e modifica della firma della funzione condizionale

Per includere condizionalmente un blocco di codice, il preprocessore ha diverse direttive (ad esempio `#if` , `#ifdef` , `#else` , `#endif` , ecc.).

```
/* 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
```

Gli operatori relazionali C normali possono essere utilizzati per la condizione `#if`

```
#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
```

Le direttive `#if` si comportano in modo analogo all'istruzione C `if` , devono contenere solo espressioni costanti integrali e nessun cast. Supporta un operatore unario aggiuntivo, `defined( identifier )` , che restituisce `1` se l'identificatore è definito e `0` altrimenti.

```
#if defined(DEBUG) && !defined(QUIET)
#define DLOG(x) (printf(x))
#else
#define DLOG(x)
#endif
```

Modifica firma funzione condizionale

Nella maggior parte dei casi, si prevede che il rilascio di build di un'applicazione abbia il minor overhead possibile. Tuttavia, durante il test di una build provvisoria, possono essere utili log aggiuntivi e informazioni sui problemi rilevati.

Ad esempio, supponiamo che ci sia una qualche funzione `SHORT SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd)` che quando si esegue una build di test è necessario generare un log relativo al suo utilizzo. Tuttavia questa funzione viene utilizzata in più punti e si desidera che quando si genera il registro, parte delle informazioni sia sapere da dove viene chiamata la funzione.

Pertanto, utilizzando la compilazione condizionale è possibile avere qualcosa di simile al seguente nel file di inclusione che dichiara la funzione. Sostituisce la versione standard della funzione con una versione di debug della funzione. Il preprocessore viene utilizzato per sostituire le chiamate alla funzione `SerOpPluAllRead()` con le chiamate alla funzione `SerOpPluAllRead_Debug()` con due argomenti aggiuntivi, il nome del file e il numero di riga di dove viene utilizzata la funzione.

La compilazione condizionale viene utilizzata per scegliere se sovrascrivere la funzione standard con una versione di debug o meno.

```
#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
```

Ciò consente di sostituire la versione standard della funzione `SerOpPluAllRead()` con una versione che fornirà il nome del file e il numero di riga nel file in cui viene chiamata la funzione.

C'è una considerazione importante: *qualsiasi file che utilizza questa funzione deve includere il file di intestazione in cui viene utilizzato questo approccio in modo che il preprocessore possa modificare la funzione. Altrimenti vedrai un errore del linker.*

La definizione della funzione sarebbe simile alla seguente. Ciò che fa questa fonte è richiedere che il preprocessore rinomina la funzione `SerOpPluAllRead()` in `SerOpPluAllRead_Debug()` e che modifichi l'elenco degli argomenti per includere due argomenti aggiuntivi, un puntatore al nome del file in cui è stata chiamata la funzione e il numero di riga nel file in cui viene utilizzata la funzione.

```
#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;
}
```

Inclusione del file di origine

Gli usi più comuni delle direttive di `#include` pre-elaborazione sono i seguenti:

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

`#include` sostituisce la dichiarazione con il contenuto del file a cui si fa riferimento. Le parentesi angolari (<>) si riferiscono ai file di intestazione installati sul sistema, mentre le virgolette ("") sono per i file forniti dall'utente.

Le macro stesse possono espandere altre macro una volta, come questo esempio illustra:

```
#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
```

Sostituzione macro

La forma più semplice di sostituzione macro è definire una `manifest constant` , come in

```
#define ARRSIZE 100
int array[ARRSIZE];
```

Definisce una macro *simile a* una *funzione* che moltiplica una variabile per `10` e memorizza il nuovo valore:

```
#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);
```

La sostituzione viene eseguita prima di qualsiasi altra interpretazione del testo del programma. Nella prima chiamata a `TIMES10` il nome `A` della definizione viene sostituito da `b` e il testo espanso viene quindi inserito al posto della chiamata. Si noti che questa definizione di `TIMES10` non è equivalente a

```
#define TIMES10(A) ((A) = (A) * 10)
```

perché questo potrebbe valutare la sostituzione di `A` , due volte, che può avere effetti collaterali

indesiderati.

Quanto segue definisce una macro simile a una funzione il cui valore è il massimo dei suoi argomenti. Ha i vantaggi di lavorare per qualsiasi tipo compatibile di argomenti e di generare codice in linea senza il sovraccarico delle chiamate di funzione. Ha gli svantaggi di valutare l'uno o l'altro dei suoi argomenti una seconda volta (compresi gli effetti collaterali) e di generare più codice di una funzione se invocato più volte.

```
#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 */
```

Per questo motivo, tali macro che valutano i loro argomenti più volte vengono solitamente evitate nel codice di produzione. Dal momento che C11 c'è la funzione `_Generic` che consente di evitare tali invocazioni multiple.

Le abbondanti parentesi nelle macro espansioni (lato destro della definizione) assicurano che gli argomenti e l'espressione risultante siano collegati correttamente e si adattino bene al contesto in cui viene chiamata la macro.

Direttiva di errore

Se il preprocessore incontra una direttiva `#error` , la compilazione viene interrotta e viene stampato il messaggio diagnostico.

```
#define DEBUG

#ifdef DEBUG
#error "Debug Builds Not Supported"
#endif

int main(void) {
    return 0;
}
```

Uscita possibile:

```
$ gcc error.c
error.c: error: #error "Debug Builds Not Supported"
```

#if 0 per bloccare le sezioni di codice

Se ci sono sezioni di codice che stai considerando di rimuovere o disabilitare temporaneamente, puoi commentare con un commento di blocco.

```
/* 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;
}
*/
```

Tuttavia, se il codice sorgente che hai circondato con un commento a blocchi ha commenti di stile bloccati nell'origine, il finale * / dei commenti di blocco esistenti può causare il tuo nuovo commento di blocco non valido e causare problemi di compilazione.

```
/* 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;
}
*/
```

Nell'esempio precedente, le ultime due righe della funzione e l'ultimo '* /' sono viste dal compilatore, quindi compilare con errori. Un metodo più sicuro è usare una direttiva `#if 0` attorno al codice che vuoi bloccare.

```
#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
```

Un vantaggio è che quando si desidera tornare indietro e trovare il codice, è molto più semplice eseguire una ricerca per "#if 0" rispetto alla ricerca di tutti i commenti.

Un altro vantaggio molto importante è che è possibile nidificare commentando il codice con `#if 0` . Questo non può essere fatto con i commenti.

Un'alternativa all'utilizzo di `#if 0` consiste nell'utilizzare un nome che non sarà `#defined` ma che è più descrittivo del motivo per cui il codice viene bloccato. Ad esempio, se esiste una funzione che sembra essere un codice morto inutile, è possibile utilizzare `#if defined(POSSIBLE_DEAD_CODE)` o `#if defined(FUTURE_CODE_REL_020201)` per il codice necessario una volta che sono state `#if defined(FUTURE_CODE_REL_020201)` altre funzionalità o qualcosa di simile. Quindi, quando si torna indietro per rimuovere o abilitare tale fonte, quelle sezioni di origine sono facili da trovare.

Incollare token

L'incollaggio di token consente di incollare insieme due argomenti macro. Ad esempio, il `front##back` restituisce il `frontback` . Un esempio famoso è l'intestazione `<TCHAR.H>` di Win32. Nella C standard, si può scrivere `L"string"` per dichiarare una stringa di caratteri ampia. Tuttavia, l'API di Windows consente di convertire tra stringhe di caratteri estesi e stringhe di caratteri strette semplicemente con `#define UNICODE` . Per implementare i letterali stringa, `TCHAR.H` utilizza questo

```
#ifdef UNICODE
#define TEXT(x) L##x
#endif
```

Ogni volta che un utente scrive `TEXT("hello, world")` e UNICODE è definito, il preprocessore C concatena `L` e l'argomento macro. `L` concatenato con `"hello, world"` dà `L"hello, world"` .

Macro predefinite

Una macro predefinita è una macro già compresa dal pre processore C senza che il programma debba definirla. Esempi inclusi

Macro predefinite obbligatorie

- `__FILE__` , che fornisce il nome file del file sorgente corrente (una stringa letterale),
- `__LINE__` per il numero di riga corrente (una costante intera),
- `__DATE__` per la data di compilazione (una stringa letterale),
- `__TIME__` per il tempo di compilazione (una stringa letterale).

C'è anche un identificatore predefinito correlato, `__func__` (ISO / IEC 9899: 2011 §6.4.2.2), che *non* è una macro:

> L'identificatore `__func__` deve essere implicitamente dichiarato dal traduttore come se, immediatamente dopo la parentesi di apertura di ogni definizione di funzione, la dichiarazione:
>
> ```
> static const char __func__[] = "function-name";
> ```
>
> è apparso, dove *nome-funzione* è il nome della funzione che include lessicamente.

`__FILE__` , `__LINE__` e `__func__` sono particolarmente utili per scopi di debug. Per esempio:

```
fprintf(stderr, "%s: %s: %d: Denominator is 0", __FILE__, __func__, __LINE__);
```

I compilatori Pre-C99, possono o non possono supportare `__func__` o possono avere una macro che agisce come denominata in modo diverso. Ad esempio, gcc ha utilizzato `__FUNCTION__` in modalità C89.

I seguenti macro consentono di chiedere dettagli sull'implementazione:

- `__STDC_VERSION__` La versione dello standard C implementata. Questo è un numero intero costante che utilizza il formato `yyyymmL` (il valore `201112L` per C11, il valore `199901L` per C99,

non definito per C89 / C90)
- `__STDC_HOSTED__` `1` se si tratta di un'implementazione ospitata, altrimenti `0` .
- `__STDC__` Se `1` , l'implementazione è conforme allo standard C.

Altre macro predefinite (non obbligatorie)

ISO / IEC 9899: 2011 §6.10.9.2 Macro ambiente:

- `__STDC_ISO_10646__` Una costante intera del formato `yyyymmL` (ad esempio, 199712L). Se questo simbolo è definito, ogni carattere nel set richiesto Unicode, se memorizzato in un oggetto di tipo `wchar_t` , ha lo stesso valore dell'identificatore breve di quel carattere. Il set Unicode richiesto è costituito da tutti i caratteri definiti da ISO / IEC 10646, insieme a tutti gli emendamenti e le rettifiche tecniche, a partire dall'anno e dal mese specificati. Se si utilizza un'altra codifica, la macro non deve essere definita e la codifica effettiva utilizzata è definita dall'implementazione.

- `__STDC_MB_MIGHT_NEQ_WC__` La costante intera 1, destinata a indicare che, nella codifica per `wchar_t` , un membro del set di caratteri di base non deve avere un valore di codice uguale al suo valore quando viene utilizzato come carattere solitario in una costante di carattere intero.

- `__STDC_UTF_16__` La costante intera 1, destinata a indicare che i valori di tipo `char16_t` sono codificati UTF-16. Se si utilizza un'altra codifica, la macro non deve essere definita e la codifica effettiva utilizzata è definita dall'implementazione.

- `__STDC_UTF_32__` La costante intera 1, destinata a indicare che i valori di tipo `char32_t` sono codificati UTF-32. Se si utilizza un'altra codifica, la macro non deve essere definita e la codifica effettiva utilizzata è definita dall'implementazione.

ISO / IEC 9899: 2011 §6.10.8.3 Macro delle funzioni condizionali

- `__STDC_ANALYZABLE__` La costante intera 1, destinata a indicare la conformità alle specifiche dell'allegato L (Analizzabilità).
- `__STDC_IEC_559__` La costante intera 1, destinata a indicare la conformità alle specifiche dell'allegato F (aritmetica in virgola mobile IEC 60559).
- `__STDC_IEC_559_COMPLEX__` La costante intera 1, destinata a indicare l'aderenza alle specifiche dell'allegato G (Aritmetica complessa compatibile con IEC 60559).
- `__STDC_LIB_EXT1__` La costante intera `201112L` , destinata a indicare il supporto per le estensioni definite nell'allegato K (Interfacce di controllo dei limiti).
- `__STDC_NO_ATOMICS__` La costante intera 1, destinata a indicare che l'implementazione non supporta i tipi atomici (incluso il qualificatore di tipo `_Atomic`) e l'intestazione `<stdatomic.h>` .
- `__STDC_NO_COMPLEX__` La costante intera 1, destinata a indicare che l'implementazione non supporta tipi complessi o l'intestazione `<complex.h>` .
- `__STDC_NO_THREADS__` La costante intera 1, destinata a indicare che l'implementazione non supporta l'intestazione `<threads.h>` .
- `__STDC_NO_VLA__`

La costante intera 1, intesa a indicare che l'implementazione non supporta matrici di lunghezza variabile o tipi modificati in modo variabile.

L'intestazione include guardie

Praticamente ogni file di intestazione dovrebbe seguire l'idioma di protezione incluso :

my-header-file.h

```
#ifndef MY_HEADER_FILE_H
#define MY_HEADER_FILE_H

// Code body for header file

#endif
```

Ciò garantisce che quando si `#include "my-header-file.h"` in più punti, non si ottengano dichiarazioni duplicate di funzioni, variabili, ecc. Immaginate la seguente gerarchia di file:

header-1.h

```
typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);
```

header-2.h

```
#include "header-1.h"

int myFunction2(MyStruct *value);
```

main.c

```
#include "header-1.h"
#include "header-2.h"

int main() {
    // do something
}
```

Questo codice ha un problema serio: il contenuto dettagliato di `MyStruct` è definito due volte, cosa che non è consentita. Ciò comporterebbe un errore di compilazione che può essere difficile da rintracciare, poiché un file di intestazione ne include un altro. Se invece lo facevi con le guardie di intestazione:

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
```

main.c

```
#include "header-1.h"
#include "header-2.h"

int main() {
    // do something
}
```

Questo si espanderebbe quindi a:

```
#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
}
```

Quando il compilatore raggiunge la seconda inclusione **dell'header-** `HEADER_1_H` , `HEADER_1_H` è già stato definito dall'inclusione precedente. Ergo, si riduce a quanto segue:

```
#define HEADER_1_H

typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#define HEADER_2_H

int myFunction2(MyStruct *value);

int main() {
    // do something
}
```

E quindi non c'è errore di compilazione.

Nota: esistono diverse convenzioni per nominare le protezioni dell'intestazione. Ad alcune persone piace `HEADER_2_H_` , alcuni includono il nome del progetto come `MY_PROJECT_HEADER_2_H` . L'importante è assicurarsi che la convenzione che segui faccia in modo che ogni file del tuo progetto abbia una protezione di intestazione unica.

Se i dettagli della struttura non sono stati inclusi nell'intestazione, il tipo dichiarato sarebbe incompleto o di tipo opaco . Tali tipi possono essere utili, nascondendo i dettagli di implementazione dagli utenti delle funzioni. Per molti scopi, il tipo `FILE` nella libreria C standard può essere considerato un tipo opaco (sebbene di solito non sia opaco in modo che le implementazioni macro delle funzioni di I / O standard possano fare uso degli interni della struttura). In tal caso, l' `header-1.h` potrebbe contenere:

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct MyStruct MyStruct;

int myFunction(MyStruct *value);

#endif
```

Si noti che la struttura deve avere un nome di tag (qui `MyStruct` - che si trova nello spazio dei nomi dei tag, separato dallo spazio dei nomi identificativo ordinario del nome typedef `MyStruct`) e che `{ … }` è omesso. Questo dice "c'è un tipo di struttura `struct MyStruct` e c'è un alias per esso `MyStruct`

".

Nel file di implementazione, è possibile definire i dettagli della struttura per rendere il tipo completo:

```
struct MyStruct {
    …
};
```

Se si utilizza C11, è possibile ripetere la `typedef struct MyStruct MyStruct;` dichiarazione senza causare un errore di compilazione, ma le versioni precedenti di C si lamentavano. Di conseguenza, è ancora meglio utilizzare l'idioma di include include, anche se in questo esempio, sarebbe facoltativo se il codice fosse stato compilato solo con compilatori che supportavano C11.

Molti compilatori supportano la direttiva `#pragma once` , che ha gli stessi risultati:

my-header-file.h

```
#pragma once

// Code for header file
```

Tuttavia, `#pragma once` non fa parte dello standard C, quindi il codice è meno portabile se lo si utilizza.

Alcune intestazioni non usano l'idioma di guardia incluso. Un esempio specifico è l'intestazione `<assert.h>` standard. Può essere incluso più volte in una singola unità di traduzione e l'effetto di tale operazione dipende dal fatto che la macro `NDEBUG` sia definita ogni volta che viene inclusa l'intestazione. Occasionalmente potresti avere un requisito analogo; questi casi saranno pochi e lontani tra loro. Normalmente, le intestazioni devono essere protette dall'idioma di protezione incluso.

ESEGUI l'implementazione

Possiamo anche usare macro per rendere il codice più facile da leggere e scrivere. Ad esempio, possiamo implementare macro per implementare il costrutto `foreach` in C per alcune strutture dati come elenchi, code, ecc.

Ecco un piccolo esempio.

```
#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);
    }
}
```

È possibile creare un'interfaccia standard per tali strutture di dati e scrivere un'implementazione generica di `FOREACH` come:

```
#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);
    }
}
```

Per utilizzare questa implementazione generica, è sufficiente implementare queste funzioni per la struttura dei dati.

```
1.  void* (*first)(void *coll);
2.  void* (*last) (void *coll);
3.  void* (*next) (void *coll, CollectionItem *currItem);
```

__cplusplus per l'uso di C esterni in codice C ++ compilato con C ++ - nome mangling

Ci sono momenti in cui un file di inclusione deve generare un output diverso dal preprocessore a seconda che il compilatore sia un compilatore C o un compilatore C ++ a causa delle differenze linguistiche.

Ad esempio una funzione o un altro esterno è definito in un file sorgente C ma è usato in un file sorgente C ++. Poiché C ++ utilizza il nome mangling (o name decoration) per generare nomi di funzioni univoci basati su tipi di argomenti di funzione, una dichiarazione di funzione C utilizzata in un file di origine C ++ causerà errori di collegamento. Il compilatore C ++ modificherà il nome esterno specificato per l'output del compilatore utilizzando le regole di mangling del nome per C ++. Il risultato sono errori di collegamento dovuti a elementi esterni non trovati quando l'output del compilatore C ++ è collegato all'output del compilatore C.

Dal momento che i compilatori C non nominano il mangling ma i compilatori C ++ fanno per tutte le etichette esterne (nomi di funzioni o nomi di variabili) generati dal compilatore C ++, è stata introdotta una macro preprocessore predefinita, `__cplusplus` , per consentire il rilevamento del compilatore.

Per aggirare questo problema di output del compilatore incompatibile per nomi esterni tra C e C ++, la macro `__cplusplus` è definita nel preprocessore C ++ e non è definita nel preprocessore C. Questo nome di macro può essere utilizzato con la direttiva `#ifdef` preprocessore condizionale o con l'operatore di `defined()` con `#if defined()` per indicare se un codice sorgente o un file di inclusione viene compilato come C ++ o C.

```
#ifdef __cplusplus
printf("C++\n");
#else
printf("C\n");
#endif
```

O potresti usare

```
#if defined(__cplusplus)
printf("C++\n");
#else
printf("C\n");
#endif
```

Per specificare il nome corretto della funzione di una funzione da un file sorgente C compilato con il compilatore C che viene utilizzato in un file sorgente C ++, è possibile verificare la costante definita `__cplusplus` per causare la `extern "C" { /* ... */ };` da utilizzare per dichiarare esterni C quando il file di intestazione è incluso in un file di origine C ++. Tuttavia, quando compilato con un compilatore C, l' `extern "C" { */ ... */ };` non è usato Questa compilazione condizionale è necessaria perché `extern "C" { /* ... */ };` è valido in C ++ ma non 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
```

Macro simili a funzioni

Le macro di tipo funzione sono simili alle funzioni `inline` , utili in alcuni casi, ad esempio il registro di debug temporaneo:

```
#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;
}
```

Qui in entrambi i casi (con `DEBUG` o meno) la chiamata si comporta allo stesso modo di una funzione con tipo di ritorno `void` . Ciò garantisce che i condizionali `if/else` siano interpretati come previsto.

Nel caso `DEBUG` questo è implementato attraverso un costrutto `do { ... } while(0)` . Nell'altro caso, `(void)0` è un'affermazione senza effetti collaterali che viene semplicemente ignorata.

Un'alternativa per quest'ultimo sarebbe

```
#define LOG(LINE) do { /* empty */ } while (0)
```

tale che sia in tutti i casi sintatticamente equivalente al primo.

Se si utilizza GCC, è anche possibile implementare una macro simile alla funzione che restituisce il risultato utilizzando un'estensione GNU non standard - espressioni di istruzioni . Per esempio:

```
#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);
}
```

Argomenti Variadici macro

C99

Macro con argomenti variadici:

Supponiamo che tu voglia creare una macro di stampa per il debug del tuo codice, prendiamo questa macro come esempio:

```
#define debug_print(msg) printf("%s:%d %s", __FILE__, __LINE__, msg)
```

Alcuni esempi di utilizzo:

La funzione `somefunc()` restituisce -1 se fallisce e 0 se ha successo, e viene chiamata da molti posti diversi all'interno del codice:

```
int retVal = somefunc();

if(retVal == -1)
{
    debug_printf("somefunc() has failed");
}

/* some other code */

 retVal = somefunc();
```

```
if(retVal == -1)
{
    debug_printf("somefunc() has failed");
}
```

Cosa succede se l'implementazione di `somefunc()` cambia e restituisce ora valori diversi che corrispondono a diversi possibili tipi di errore? Si desidera comunque utilizzare la macro di debug e stampare il valore dell'errore.

```
debug_printf(retVal);      /* this would obviously fail */
debug_printf("%d",retVal); /* this would also fail */
```

Per risolvere questo problema è stata introdotta la macro `__VA_ARGS__` . Questa macro consente più parametri X-macro:

Esempio:

```
 #define debug_print(msg, ...) printf(msg, __VA_ARGS__) \
                               printf("\nError occurred in file:line (%s:%d)\n", __FILE__,
__LINE)
```

Uso:

```
int retVal = somefunc();

debug_print("retVal of somefunc() is-> %d", retVal);
```

Questa macro ti consente di passare più parametri e stamparli, ma ora ti impedisce di inviare alcun parametro.

```
debug_print("Hey");
```

Ciò solleverebbe un errore di sintassi poiché la macro si aspetta almeno un altro argomento e il pre-processore non ignorerebbe la mancanza di virgola nella macro `debug_print()` . Anche `debug_print("Hey",);` solleverebbe un errore di sintassi poiché non si può mantenere l'argomento passato alla macro vuota.

Per risolvere questo problema, è stata introdotta la macro `##__VA_ARGS__` , questa macro indica che se non esistono argomenti variabili, la virgola viene cancellata dal pre-processore dal codice.

Esempio:

```
 #define debug_print(msg, ...) printf(msg, ##__VA_ARGS__) \
                               printf("\nError occured in file:line (%s:%d)\n", __FILE__,
__LINE)
```

Uso:

```
 debug_print("Ret val of somefunc()?");
```

```
debug_print("%d",somefunc());
```

Leggi Preprocessore e macro online: https://riptutorial.com/it/c/topic/447/preprocessore-e-macro

Capitolo 49: puntatori

introduzione

Un puntatore è un tipo di variabile che può memorizzare l'indirizzo di un altro oggetto o una funzione.

Sintassi

- <Tipo di dati> * <Nome variabile>;
- int * ptrToInt;
- void * ptrToVoid; / * C89 + * /
- struct someStruct * ptrToStruct;
- int ** ptrToPtrToInt;
- int arr [lunghezza]; int * ptrToFirstElem = arr; / * Per <C99 'length' deve essere una costante di tempo di compilazione, per> = C11 potrebbe essere necessario essere uno. * /
- int * arrayOfPtrsToInt [lunghezza]; / * Per <C99 'length' deve essere una costante di tempo di compilazione, per> = C11 potrebbe essere necessario essere uno. * /

Osservazioni

La posizione dell'asterisco non influisce sul significato della definizione:

```
/* The * operator binds to right and therefore these are all equivalent. */
int *i;
int * i;
int* i;
```

Tuttavia, quando si definiscono più puntatori contemporaneamente, ognuno richiede il proprio asterisco:

```
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 */
```

È anche possibile una serie di puntatori, dove viene indicato un asterisco prima del nome della variabile dell'array:

```
int *foo[2]; /* foo is a array of pointers, can be accessed as *foo[0] and *foo[1] */
```

Examples

Errori comuni

L'uso improprio dei puntatori è spesso una fonte di bug che può includere bug di sicurezza o crash

del programma, molto spesso a causa di errori di segmentazione.

Non verificare i problemi di allocazione

L'allocazione della memoria non è garantita e può invece restituire un puntatore `NULL` . L'utilizzo del valore restituito, senza verificare se l'allocazione ha esito positivo, richiama il comportamento non definito . Questo di solito porta a un arresto anomalo, ma non è garantito che si verifichi un arresto anomalo, quindi fare affidamento su questo può anche portare a problemi.

Ad esempio, modo pericoloso:

```
struct SomeStruct *s = malloc(sizeof *s);
s->someValue = 0; /* UNSAFE, because s might be a null pointer */
```

Modo sicuro:

```
struct SomeStruct *s = malloc(sizeof *s);
if (s)
{
    s->someValue = 0; /* This is safe, we have checked that s is valid */
}
```

Utilizzando numeri letterali invece di sizeof quando si richiede memoria

Per una determinata configurazione di compilatore / macchina, i tipi hanno una dimensione nota; tuttavia, non esiste uno standard che definisce che la dimensione di un dato tipo (diversa da `char`) sarà la stessa per tutte le configurazioni del compilatore / macchina. Se il codice utilizza 4 invece di `sizeof(int)` per l'allocazione di memoria, potrebbe funzionare sulla macchina originale, ma il codice non è necessariamente portabile ad altre macchine o compilatori. Le dimensioni fisse per i tipi devono essere sostituite da `sizeof(that_type)` o `sizeof(*var_ptr_to_that_type)` .

Assegnazione non portatile:

```
int *intPtr = malloc(4*1000);    /* allocating storage for 1000 int */
long *longPtr = malloc(8*1000);  /* allocating storage for 1000 long */
```

Assegnazione portatile:

```
int *intPtr = malloc(sizeof(int)*1000);     /* allocating storage for 1000 int */
long *longPtr = malloc(sizeof(long)*1000);  /* allocating storage for 1000 long */
```

O, meglio ancora:

```
int *intPtr = malloc(sizeof(*intPtr)*1000);     /* allocating storage for 1000 int */
long *longPtr = malloc(sizeof(*longPtr)*1000);  /* allocating storage for 1000 long */
```

Perdite di memoria

Mancata allocazione della memoria utilizzando lead `free` a un accumulo di memoria non riutilizzabile, che non è più utilizzato dal programma; questo è chiamato una perdita di memoria . Le perdite di memoria sprecano risorse di memoria e possono portare a problemi di allocazione.

Errori logici

Tutte le allocazioni devono seguire lo stesso schema:

1. Assegnazione tramite `malloc` (o `calloc`)
2. Utilizzo per memorizzare i dati
3. Ridistribuzione utilizzando `free`

La mancata aderenza a questo schema, come l'uso della memoria dopo una chiamata a `free` (puntatore penzolante) o prima di una chiamata a `malloc` (puntatore jolly), chiamata `free` due volte ("double free"), ecc., Solitamente causa un errore di segmentazione e provoca un arresto anomalo del programma.

Questi errori possono essere transitori e difficili da debugare: ad esempio, la memoria liberata di solito non viene immediatamente recuperata dal sistema operativo, e quindi i puntatori penzolanti possono persistere per un po 'e sembrano funzionare.

Sui sistemi in cui funziona, Valgrind è uno strumento prezioso per identificare quale memoria è trapelata e dove è stata originariamente allocata.

Creazione di puntatori per impilare le variabili

La creazione di un puntatore non prolunga la vita della variabile puntata. Per esempio:

```
int* myFunction()
{
    int x = 10;
    return &x;
}
```

Qui, `x` ha *durata di archiviazione automatica* (comunemente nota come allocazione dello *stack*). Poiché è allocata nello stack, la sua durata è limitata a quando `myFunction` è in esecuzione; dopo che `myFunction` è uscito, la variabile `x` viene distrutta. Questa funzione ottiene l'indirizzo di `x` (usando `&x`), e lo restituisce al chiamante, lasciando il chiamante con un puntatore a una variabile inesistente. Il tentativo di accedere a questa variabile invocherà quindi un comportamento non definito .

La maggior parte dei compilatori non cancella un frame di stack dopo che la funzione è stata chiusa, quindi il dereferenziamento del puntatore restituito spesso fornisce i dati previsti. Tuttavia, quando viene chiamata un'altra funzione, la memoria puntata potrebbe essere sovrascritta e sembra che i dati puntati siano stati danneggiati.

Per risolvere questo, o `malloc` lo spazio di archiviazione per la variabile da restituire e restituire un puntatore allo spazio di archiviazione appena creato oppure richiedere che venga passato un puntatore valido alla funzione anziché restituirlo, ad esempio:

```
#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;
}
```

Incremento / decremento e dereferenziazione

Se scrivi `*p++` per incrementare ciò che è indicato da `p` , hai torto.

Post incremento / decremento viene eseguito prima del dereferenziamento. Pertanto, questa espressione incrementerà il puntatore `p` stesso e restituirà ciò che è stato indicato da `p` prima di incrementarlo senza modificarlo.

Dovresti scrivere `(*p)++` per incrementare ciò che è indicato da `p` .

Questa regola si applica anche al post decremento: `*p--` diminuirà il puntatore `p` stesso, non quello che viene indicato da `p` .

Dereferenziare un puntatore

```
int a = 1;
int *a_pointer = &a;
```

Per dereferenziare `a_pointer` e modificare il valore di a, usiamo la seguente operazione

```
*a_pointer = 2;
```

Questo può essere verificato usando le seguenti dichiarazioni di stampa.

```
printf("%d\n", a); /* Prints 2 */
printf("%d\n", *a_pointer); /* Also prints 2 */
```

Tuttavia, si sarebbe errato dereferenziare un puntatore `NULL` o altrimenti non valido. Questo

```
int *p1, *p2;

p1 = (int *) 0xbad;
p2 = NULL;

*p1 = 42;
*p2 = *p1 + 1;
```

è di solito un comportamento indefinito . `p1` non può essere dereferenziato perché punta a un indirizzo `0xbad` che potrebbe non essere un indirizzo valido. Chissà cosa c'è? Potrebbe essere la memoria del sistema operativo o la memoria di un altro programma. L'unico codice temporale come questo è usato, è nello sviluppo embedded, che memorizza informazioni particolari su indirizzi hard-coded. `p2` non può essere dereferenziato perché è `NULL` , che non è valido.

Dereferenziare un puntatore a una struttura

Diciamo che abbiamo la seguente struttura:

```
struct MY_STRUCT
```

```
{
    int my_int;
    float my_float;
};
```

Possiamo definire `MY_STRUCT` per omettere la parola chiave `struct` modo da non dover scrivere `struct MY_STRUCT` ogni volta che la usiamo. Questo, tuttavia, è facoltativo.

```
typedef struct MY_STRUCT MY_STRUCT;
```

Se poi abbiamo un puntatore a un'istanza di questa struttura

```
MY_STRUCT *instance;
```

Se questa istruzione appare nell'ambito del file, l' `instance` verrà inizializzata con un puntatore nullo all'avvio del programma. Se questa istruzione appare all'interno di una funzione, il suo valore non è definito. La variabile deve essere inizializzata in modo che punti a una variabile `MY_STRUCT` valida o a spazio assegnato dinamicamente, prima che possa essere dereferenziata. Per esempio:

```
MY_STRUCT info = { 1, 3.141593F };
MY_STRUCT *instance = &info;
```

Quando il puntatore è valido, possiamo dereferenziarlo per accedere ai suoi membri usando una delle due diverse notazioni:

```
int a = (*instance).my_int;
float b = instance->my_float;
```

Mentre entrambi questi metodi funzionano, è preferibile utilizzare l'operatore freccia `->` anziché la combinazione di parentesi, l'operatore dereferenziazione `*` e il punto `.` operatore perché è più facile da leggere e capire, specialmente con gli usi annidati.

Un'altra importante differenza è mostrata di seguito:

```
MY_STRUCT copy = *instance;
copy.my_int    = 2;
```

In questo caso, la `copy` contiene una copia del contenuto `instance` . Cambiare `my_int` di `copy` non lo cambierà in `instance` .

```
MY_STRUCT *ref = instance;
ref->my_int    = 2;
```

In questo caso, `ref` è un riferimento `instance` . Cambiando `my_int` usando il riferimento lo cambieremo in `instance` .

È pratica comune utilizzare i puntatori alle strutture come parametri nelle funzioni, piuttosto che le

strutture stesse. L'utilizzo delle structs come parametri di funzione potrebbe causare un overflow dello stack se la struttura è grande. L'utilizzo di un puntatore su una struttura utilizza solo uno spazio sufficiente per il puntatore, ma può causare effetti secondari se la funzione modifica la struttura passata nella funzione.

Puntatori di funzione

I puntatori possono anche essere usati per indicare le funzioni.

Prendiamo una funzione di base:

```
int my_function(int a, int b)
{
    return 2 * a + 3 * b;
}
```

Ora definiamo un puntatore del tipo di quella funzione:

```
int (*my_pointer)(int, int);
```

Per crearne uno, usa questo modello:

```
return_type_of_func (*my_func_pointer)(type_arg1, type_arg2, ...)
```

Quindi dobbiamo assegnare questo puntatore alla funzione:

```
my_pointer = &my_function;
```

Questo puntatore può ora essere utilizzato per chiamare la funzione:

```
/* 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);
}
```

Sebbene questa sintassi sembri più naturale e coerente con i tipi di base, i puntatori alle funzioni di attribuzione e dereferenziazione non richiedono l'uso degli operatori `&` e `*` . Quindi il seguente snippet è ugualmente valido:

```
/* Attribution without the & operator */
my_pointer = my_function;
```

```
/* Dereferencing without the * operator */
int result = my_pointer(4, 2);
```

Per aumentare la leggibilità dei puntatori di funzione, è possibile utilizzare typedef.

```
typedef void (*Callback)(int a);

void some_function(Callback callback)
{
    int a = 4;
    callback(a);
}
```

Un altro trucco di leggibilità è che lo standard C consente di semplificare un puntatore a funzione in argomenti come sopra (ma non nella dichiarazione di variabili) a qualcosa che assomiglia a un prototipo di funzione; quindi il seguente può essere usato in modo equivalente per le definizioni e le dichiarazioni di funzioni:

```
void some_function(void callback(int))
{
    int a = 4;
    callback(a);
}
```

Guarda anche

Puntatori di funzione

Inizializzazione dei puntatori

L'inizializzazione del puntatore è un buon modo per evitare i puntatori selvaggi. L'inizializzazione è semplice e non è diversa dall'inizializzazione di una variabile.

```
#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 */

    ...
}
```

Nella maggior parte dei sistemi operativi, l'utilizzo involontario di un puntatore che è stato inizializzato su `NULL` spesso causa l'arresto immediato del programma immediatamente, facilitando l'identificazione della causa del problema. L'utilizzo di un puntatore non inizializzato può spesso causare errori di diagnostica.

Attenzione:

Il risultato del dereferenziamento di un puntatore `NULL` non è definito, pertanto *non causerà necessariamente un arresto anomalo* anche se questo è il comportamento naturale del sistema operativo su cui è in esecuzione il programma. Le ottimizzazioni del compilatore possono mascherare l'arresto anomalo, causare l'arresto anomalo prima o dopo il punto nel codice sorgente in cui si è verificata la deviazione del puntatore nullo o causare la rimozione inaspettata dal programma di parti del codice che contengono il riferimento del puntatore nullo. Generalmente, i build di debug non mostrano questi comportamenti, ma ciò non è garantito dallo standard di lingua. Sono ammessi anche altri comportamenti imprevisti e / o indesiderati.

Poiché `NULL` non punta mai a una variabile, a una memoria allocata oa una funzione, è sicuro da utilizzare come valore di guardia.

Attenzione:

Solitamente `NULL` è definito come `(void *)0` . Ma questo non implica che l'indirizzo di memoria assegnato sia `0x0` . Per ulteriori chiarimenti, fai riferimento a C-faq per i puntatori NULL

Si noti che è anche possibile inizializzare i puntatori per contenere valori diversi da NULL.

```
int i1;

int main()
{
   int *p1 = &i1;
   const char *p2 = "A constant string to point to";
   float *p3 = malloc(10 * sizeof(float));
}
```

Indirizzo-dell'operatore (&)

Per qualsiasi oggetto (cioè variabile, matrice, unione, struttura, puntatore o funzione) l'operatore di indirizzo unario può essere utilizzato per accedere all'indirizzo di quell'oggetto.

Supporre che

```
int i = 1;
int *p = NULL;
```

Quindi una dichiarazione `p = &i;` , copia l'indirizzo della variabile `i` nel puntatore `p` .

È espresso come `p` ***punti a*** `i` .

`printf("%d\n", *p);` stampa 1, che è il valore di `i` .

Puntatore aritmetico

Vedi qui: Pointer Arithmetic

void * puntatori come argomenti e valori di ritorno alle funzioni standard

K & R

`void*` è un tipo catch all per i puntatori ai tipi di oggetto. Un esempio di questo in uso è con la funzione `malloc` , che è dichiarata come

```
void* malloc(size_t);
```

Il tipo restituito da puntatore a vuoto indica che è possibile assegnare il valore restituito da `malloc` a un puntatore a qualsiasi altro tipo di oggetto:

```
int* vector = malloc(10 * sizeof *vector);
```

Generalmente è considerata una buona pratica *non* inserire esplicitamente i valori dentro e fuori i puntatori void. Nel caso specifico di `malloc()` questo è dovuto al fatto che con un cast esplicito, il compilatore può assumere, ma non avvisare, un tipo di ritorno errato per `malloc()` , se si dimentica di includere `stdlib.h` . È anche il caso di usare il comportamento corretto dei puntatori void per conformarsi meglio al principio DRY (non ripeterlo); confrontare il precedente con il seguente, in cui il seguente codice contiene diversi posti aggiuntivi inutili in cui un errore di battitura potrebbe causare problemi:

```
int* vector = (int*)malloc(10 * sizeof int*);
```

Allo stesso modo, funzioni come

```
void* memcpy(void *restrict target, void const *restrict source, size_t size);
```

avere i loro argomenti specificati come `void *` perché l'indirizzo di qualsiasi oggetto, indipendentemente dal tipo, può essere passato. Qui anche una chiamata non dovrebbe usare un cast

```
unsigned char buffer[sizeof(int)];
int b = 67;
memcpy(buffer, &b, sizeof buffer);
```

Puntatori di Const

Puntatori singoli

- Puntatore a un `int`

  Il puntatore può puntare a numeri interi diversi e gli `int` possono essere modificati tramite il puntatore. Questo esempio di codice assegna b a punto `int b` quindi cambia il valore di `b` a `100` .

```
int b;
int* p;
p = &b;    /* OK */
*p = 100;  /* OK */
```

- Puntatore a un `const int`

  Il puntatore può puntare a numeri interi diversi ma il valore di `int` non può essere modificato tramite il puntatore.

```
int b;
const int* p;
p = &b;    /* OK */
*p = 100;  /* Compiler Error */
```

- `const` puntatore a `int`

  Il puntatore può solo puntare a un `int` ma il valore di `int` può essere modificato tramite il puntatore.

```
int a, b;
int* const p = &b; /* OK as initialisation, no assignment */
*p = 100;  /* OK */
p = &a;    /* Compiler Error */
```

- `const` puntatore a `const int`

  Il puntatore può solo puntare a un `int` e l' `int` non può essere modificato tramite il puntatore.

```
int a, b;
const int* const p = &b; /* OK as initialisation, no assignment */
p = &a;   /* Compiler Error */
*p = 100; /* Compiler Error */
```

Puntatore al puntatore

- Puntatore a un puntatore a un `int`

  Questo codice assegna l'indirizzo di `p1` al doppio puntatore `p` (che punta quindi a `int* p1` (che punta a `int`)).

  Quindi cambia `p1` in modo che punti a `int a` . Quindi cambia il valore di a per essere 100.

```
void f1(void)
{
  int a, b;
  int *p1;
  int **p;
  p1 = &b; /* OK */
  p = &p1; /* OK */
  *p = &a; /* OK */
```

```
  **p = 100; /* OK */
}
```

- Puntatore per puntare a un `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’ */
}
```

- Puntatore a `const` puntatore a un `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` puntatore a puntatore a `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 */
}
```

- Puntatore a `const` puntatore a `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` puntatore a puntatore a `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` puntatore a `const` puntatore a `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 */
}
```

Same Asterisk, Different Meanings

Premessa

La cosa più confusa che circonda la sintassi del puntatore in C e C ++ è che ci sono in realtà due significati diversi che si applicano quando il simbolo del puntatore, l'asterisco (`*`), viene utilizzato con una variabile.

Esempio

In primo luogo, si usa `*` per **dichiarare** una variabile puntatore.

```
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' */
```

Quando non stai dichiarando (o moltiplicando), `*` è usato per **dereferenziare** una variabile puntatore:

```
*p = 123;
/* 'p' was pointing to 'i', so this changes value of 'i' to 123 */
```

Quando vuoi una variabile puntatore esistente per contenere l'indirizzo di un'altra variabile, **non** usi `*` , ma fallo in questo modo:

```
p = &another_variable;
```

Una confusione comune tra i neofiti di programmazione C si verifica quando dichiarano e inizializzano una variabile puntatore allo stesso tempo.

```
int *p = &i;
```

Poiché `int i = 5;` e `int i; i = 5;` dare lo stesso risultato, alcuni potrebbero pensare `int *p = &i;` e `int *p; *p = &i;` dare lo stesso risultato anche Il fatto è, no, `int *p; *p = &i;` tenterà di deferire un puntatore **non inizializzato** che risulterà in UB. Non usare mai `*` quando non stai dichiarando né dereferenziare un puntatore.

Conclusione

L'asterisco (`*`) ha due significati distinti all'interno di C in relazione ai puntatori, a seconda di dove viene utilizzato. Se utilizzato all'interno di una **dichiarazione di variabile** , il valore sul lato destro del lato uguale deve essere un ***valore puntatore*** a un ***indirizzo*** in memoria. Quando viene utilizzato con una **variabile** già **dichiarata** , l'asterisco ***dedurrà*** il riferimento al valore del puntatore, lo seguirà nel punto appuntito in memoria e consentirà di assegnare o recuperare il valore memorizzato.

Porta via

È importante tenere a mente le tue P e Q, per così dire, quando si tratta di puntatori. Fai attenzione quando usi l'asterisco e cosa significa quando lo usi lì. Affrontare questo piccolo dettaglio potrebbe portare a comportamenti buggy e / o indefiniti che in realtà non si vuole avere a che fare.

Puntatore al puntatore

In C, un puntatore può fare riferimento a un altro puntatore.

```
#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;
}
```

Ma, riferimento-e-riferimento direttamente non è permesso.

```
#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! */

  ...
```

introduzione

Un puntatore viene dichiarato molto simile a qualsiasi altra variabile, tranne che un asterisco (`*`) è posto tra il tipo e il nome della variabile per indicare che si tratta di un puntatore.

```
int *pointer; /* inside a function, pointer is uninitialized and doesn't point to any valid
object yet */
```

Per dichiarare due variabili puntatore dello stesso tipo, nella stessa dichiarazione, utilizzare il simbolo asterisco prima di ogni identificatore. Per esempio,

```
int *iptr1, *iptr2;
int *iptr3,  iptr4;  /* iptr3 is a pointer variable, whereas iptr4 is misnamed and is an int
*/
```

L'operatore di riferimento o di riferimento indicato da una e commerciale (`&`) fornisce l'indirizzo di una determinata variabile che può essere inserito in un puntatore di tipo appropriato.

```
int value = 1;
pointer = &value;
```

L'operatore di riferimento o dereferenza contrassegnato da un asterisco (`*`) ottiene il contenuto di un oggetto puntato da un puntatore.

```
printf("Value of pointed to integer: %d\n", *pointer);
/* Value of pointed to integer: 1 */
```

Se il puntatore punta a una struttura oa un tipo di unione, puoi dereferenziarlo e accedere direttamente ai suoi membri usando l'operatore `->` :

```
SomeStruct *s = &someObject;
s->someMember = 5; /* Equivalent to (*s).someMember = 5 */
```

In C, un puntatore è un tipo di valore distinto che può essere riassegnato e altrimenti viene trattato come una variabile a sé stante. Ad esempio il seguente esempio stampa il valore del puntatore (variabile) stesso.

```
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 */
```

Poiché un puntatore è una variabile mutabile, è possibile che non punti a un oggetto valido, sia impostandolo su null

```
pointer = 0;     /* or alternatively */
pointer = NULL;
```

o semplicemente contenendo un pattern di bit arbitrario che non è un indirizzo valido. Quest'ultima è una situazione molto brutta, perché non può essere testata prima che il puntatore venga sottoposto a dereferenziazione, c'è solo un test per il caso in cui un puntatore è nullo:

```
if (!pointer) exit(EXIT_FAILURE);
```

Un puntatore può essere sottoposto a dereferenziazione solo se punta a un oggetto *valido* , altrimenti il comportamento non è definito. Molte implementazioni moderne possono aiutarti a sollevare qualche tipo di errore come un errore di segmentazione e terminare l'esecuzione, ma altri potrebbero semplicemente lasciare il tuo programma in uno stato non valido.

Il valore restituito dall'operatore di dereferenziazione è un alias mutabile della variabile originale, quindi può essere modificato, modificando la variabile originale.

```
*pointer += 1;
printf("Value of pointed to variable after change: %d\n", *pointer);
/* Value of pointed to variable after change: 2 */
```

I puntatori sono anche riassegnabili. Ciò significa che un puntatore che punta a un oggetto può essere successivamente utilizzato per puntare a un altro oggetto dello stesso tipo.

```
int value2 = 10;
pointer = &value2;
printf("Value from pointer: %d\n", *pointer);
/* Value from pointer: 10 */
```

Come qualsiasi altra variabile, i puntatori hanno un tipo specifico. Ad esempio, non è possibile assegnare l'indirizzo di un `short int` a un puntatore a un `long int` . Tale comportamento è indicato come punire tipo ed è vietato in C, anche se ci sono alcune eccezioni.

Sebbene il puntatore debba essere di un tipo specifico, la memoria allocata per ciascun tipo di puntatore è uguale alla memoria utilizzata dall'ambiente per memorizzare gli indirizzi, piuttosto che la dimensione del tipo puntato.

```
#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;
}
```

(NB: se si utilizza Microsoft Visual Studio, che non supporta gli standard C99 o C11, è necessario utilizzare `%Iu` [1] anziché `%zu` nell'esempio precedente.)

Si noti che i risultati sopra riportati possono variare da ambiente a ambiente in numeri, ma tutti gli ambienti mostrerebbero uguali dimensioni per diversi tipi di puntatore.

Estratto basato sulle informazioni fornite dall'Università di Cardiff C Pointers Introduzione

Puntatori e matrici

Puntatori e array sono intimamente connessi in C. Le matrici in C si trovano sempre in posizioni contigue nella memoria. L'aritmetica del puntatore viene sempre ridimensionata in base alla dimensione dell'oggetto puntato. Quindi, se abbiamo una matrice di tre doppi e un puntatore alla base, `*ptr` riferisce al primo doppio, `*(ptr + 1)` al secondo, `*(ptr + 2)` al terzo. Una notazione più conveniente è usare la notazione array `[]` .

```
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]);
```

Quindi essenzialmente ptr e il nome dell'array sono intercambiabili. Questa regola significa anche che un array decade a un puntatore quando viene passato a una subroutine.

```
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])
}
```

Un puntatore può puntare a qualsiasi elemento in una matrice o all'elemento oltre l'ultimo elemento. È tuttavia un errore impostare un puntatore su qualsiasi altro valore, incluso l'elemento prima dell'array. (Il motivo è che su architetture segmentate l'indirizzo prima che il primo elemento possa attraversare un limite di segmento, il compilatore assicura che ciò non avvenga per l'ultimo elemento più uno).

Nota 1: informazioni sul formato Microsoft possono essere trovate tramite `printf()` e sintassi delle specifiche del formato .

Comportamento polimorfico con puntatori void

La `qsort()` libreria standard `qsort()` è un buon esempio di come si possano usare i puntatori void per far funzionare una singola funzione su una grande varietà di tipi diversi.

```
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 */
```

La matrice da ordinare viene passata come un puntatore vuoto, quindi è possibile utilizzare una matrice di qualsiasi tipo di elemento. I prossimi due argomenti dicono a `qsort()` quanti elementi deve aspettarsi nell'array e quanto è grande, in byte, ogni elemento.

L'ultimo argomento è un puntatore a funzione di una funzione di confronto che a sua volta prende due puntatori void. Facendo in modo che il chiamante fornisca questa funzione, `qsort()` può effettivamente ordinare elementi di qualsiasi tipo.

Ecco un esempio di tale funzione di confronto, per confrontare i float. Si noti che qualsiasi funzione di confronto passata a `qsort()` deve avere questa firma di tipo. Il modo in cui è reso polimorfico è colando gli argomenti del puntatore del vuoto ai puntatori del tipo di elemento che vogliamo confrontare.

```
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;
}
```

Dato che sappiamo che qsort userà questa funzione per confrontare i float, convertiamo gli argomenti del puntatore void indietro in virgola mobile prima di dereferenziarli.

Ora, l'uso della funzione polimorfica qsort su un array "array" con lunghezza "len" è molto semplice:

```
qsort(array, len, sizeof(array[0]), compare_floats);
```

Leggi puntatori online: https://riptutorial.com/it/c/topic/1108/puntatori

Capitolo 50: Puntatori di funzione

introduzione

I puntatori di funzione sono puntatori che puntano a funzioni invece di tipi di dati. Possono essere utilizzati per consentire la variabilità della funzione che deve essere chiamata, in fase di esecuzione.

Sintassi

- returnType (* nome) (parametri)
- typedef returnType (* nome) (parametri)
- typedef returnType Name (parametri);
  Nome * nome;
- typedef returnType Name (parametri);
  typedef Nome * NamePtr;

Examples

Assegnazione di un puntatore funzione

```
#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;
}
```

Restituzione dei puntatori funzione da una funzione

```
#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;
}
```

Migliori pratiche

Usando typedef

Potrebbe essere utile usare un `typedef` invece di dichiarare il puntatore funzione ogni volta a mano.

La sintassi per la dichiarazione di `typedef` per un puntatore a funzione è:

```
typedef returnType (*name)(parameters);
```

Esempio:

Positivo che abbiamo una funzione, `sort` , che si aspetta un puntatore a una funzione `compare` tale che:

> compare - Una funzione di confronto per due elementi che deve essere fornita ad una funzione di ordinamento.
>
> "compare" dovrebbe restituire 0 se i due elementi sono considerati uguali, un valore positivo se il primo elemento passato è "più grande" in un certo senso rispetto al secondo elemento e altrimenti la funzione restituisce un valore negativo (il che significa che il primo elemento è "minore" rispetto al secondo).

Senza un `typedef` passeremmo un puntatore a funzione come argomento a una funzione nel modo seguente:

```
void sort(int (*compare)(const void *elem1, const void *elem2)) {
    /* inside of this block, the function is named "compare" */
}
```

Con un `typedef` , scriveremmo:

```
typedef int (*compare_func)(const void *, const void *);
```

e quindi potremmo cambiare la firma della funzione di `sort` a:

```
void sort(compare_func func) {
    /* In this block the function is named "func" */
}
```

entrambe le definizioni di `sort` accettano qualsiasi funzione della forma

```
int compare(const void *arg1, const void *arg2) {
    /* Note that the variable names do not have to be "elem1" and "elem2" */
}
```

I puntatori di funzione sono l'unico posto in cui devi includere la proprietà del puntatore del tipo, ad

esempio non provare a definire tipi come `typedef struct something_struct *something_type` . Ciò vale anche per una struttura con membri a cui non si deve accedere direttamente dai chiamanti API, ad esempio il tipo `FILE` stdio.h (che come si vedrà ora non è un puntatore).

Prendendo indicatori di contesto.

Un puntatore a funzione dovrebbe quasi sempre prendere un vuoto fornito dall'utente * come puntatore del contesto.

Esempio

```
/* 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);
}
```

L'uso del puntatore del contesto significa che i parametri aggiuntivi non devono essere codificati nella funzione puntata o richiedono l'uso globale.

La funzione di libreria `qsort()` non segue questa regola e spesso si può andare via senza contesto per funzioni di confronto banali. Ma per qualcosa di più complicato, il puntatore del contesto diventa essenziale.

Guarda anche

Funzioni puntatori

introduzione

Proprio come `char` e `int` , una funzione è una caratteristica fondamentale di C. Come tale, è possibile dichiarare un puntatore a uno: il che significa che è possibile passare la *funzione da chiamare* a un'altra funzione per aiutarla a svolgere il proprio lavoro. Ad esempio, se disponi di una funzione `graph()` che visualizza un grafico, puoi passare la *funzione da rappresentare graficamente* in `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)
```

uso

Quindi il codice sopra riportato graficherà su qualunque funzione tu abbia passato in esso - purché tale funzione soddisfi determinati criteri: cioè, che tu superi un `double` e ottieni un `double` . Ci sono molte funzioni come questa: `sin()` , `cos()` , `tan()` , `exp()` ecc. - ma ce ne sono molte che non lo sono, come `graph()` stesso!

Sintassi

Quindi, come si specificano le funzioni che è possibile passare in `graph()` e quali non è possibile? Il modo convenzionale consiste nell'utilizzare una sintassi che potrebbe non essere facile da leggere o comprendere:

```
double (*fn)(double); // fn is a pointer-to-function that takes a double and returns one
```

Il problema sopra è che ci sono due cose che cercano di essere definite allo stesso tempo: la struttura della funzione e il fatto che si tratta di un puntatore. Quindi, dividi le due definizioni! Ma usando `typedef` , è possibile ottenere una sintassi migliore (più facile da leggere e capire).

Mnemonico per scrivere puntatori di funzioni

Tutte le funzioni C sono in effetti puntatori a un punto nella memoria del programma in cui esiste un codice. L'uso principale di un puntatore a funzione è quello di fornire un "callback" ad altre funzioni (o per simulare classi e oggetti).

La sintassi di una funzione, come definito più avanti in questa pagina è:

returnType (* nome) (parametri)

Un mnemonico per scrivere una definizione di puntatore a funzione è la seguente procedura:

1. Inizia scrivendo una dichiarazione di una funzione normale: `returnType name(parameters)`
2. Avvolgi il nome della funzione con la sintassi del puntatore: `returnType (*name)(parameters)`

Nozioni di base

Proprio come si può avere un puntatore a **int** , **char** , **float** , **array / string** , **struct** , ecc. - si può avere un puntatore a una funzione.

La dichiarazione del puntatore assume il *valore di ritorno della funzione* , il *nome della funzione* e il *tipo di argomenti / parametri che riceve* .

Supponi di avere la seguente funzione dichiarata e inizializzata:

```
int addInt(int n, int m){
    return n+m;
}
```

È possibile dichiarare e inizializzare un puntatore a questa funzione:

```
int (*functionPtrAdd)(int, int) = addInt; // or &addInt - the & is optional
```

Se disponi di una funzione di annullamento, potrebbe avere il seguente aspetto:

```
void Print(void){
    printf("look ma' - no hands, only pointers!\n");
}
```

Quindi dichiarare il puntatore ad esso sarebbe:

```
void (*functionPtrPrint)(void) = Print;
```

Accedere alla funzione stessa richiederebbe il dereferenziamento del puntatore:

```
sum = (*functionPtrAdd)(2, 3); //will assign 5 to sum
(*functionPtrPrint)(); //will print the text in Print function
```

Come visto in esempi più avanzati in questo documento, dichiarare un puntatore a una funzione

potrebbe diventare disordinato se la funzione è passata più di pochi parametri. Se hai alcune indicazioni su funzioni che hanno "struttura" identica (stesso tipo di valore di ritorno e lo stesso tipo di parametri) è meglio usare il comando **typedef** per risparmiare qualche digitazione e per rendere il codice più chiaro:

```
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;
}
```

È inoltre possibile creare una **matrice di puntatori di funzioni** . Se tutti i puntatori sono della stessa "struttura":

```
int (*array[2]) (int x, int y); // can hold 2 function pointers
array[0] = Add;
array[1] = Multiply;
```

Puoi saperne di più qui e qui .

È anche possibile definire una serie di puntatori di funzioni di tipi diversi, sebbene ciò richiederebbe la trasmissione ogni volta che si desidera accedere alla funzione specifica. Puoi saperne di più qui .

Leggi Puntatori di funzione online: https://riptutorial.com/it/c/topic/250/puntatori-di-funzione

Capitolo 51: Punti di sequenza

Osservazioni

Standard internazionale ISO / IEC 9899: 201x Linguaggi di programmazione - C

> L'accesso a un oggetto volatile, la modifica di un oggetto, la modifica di un file o il richiamo di una funzione che esegue una di queste operazioni sono tutti *effetti collaterali* , che sono cambiamenti nello stato dell'ambiente di esecuzione.
>
> La presenza di un *punto di sequenza* tra la valutazione delle espressioni A e B implica che ogni calcolo del valore ed effetto collaterale associato ad A viene sequenziato prima di ogni calcolo del valore ed effetto collaterale associato a B.

Ecco l'elenco completo dei punti di sequenza dell'allegato C della bozza di pre-pubblicazione online 2011 dello standard di lingua C:

> **Punti di sequenza**
>
> 1 I seguenti sono i punti di sequenza descritti in 5.1.2.3:
>
> - Tra le valutazioni del designatore di funzioni e gli argomenti effettivi in una chiamata di funzione e la chiamata effettiva. (6.5.2.2).
> - Tra le valutazioni del primo e del secondo operando dei seguenti operatori: AND logico `&&` (6.5.13); OR logico `||` (6.5.14); comma `,` (6.5.17).
> - Tra le valutazioni del primo operando del condizionale `? :` viene valutato l'operatore e il secondo e il terzo operando (6.5.15).
> - La fine di un dichiarante completo: dichiarators (6.7.6);
> - Tra la valutazione di un'espressione completa e la successiva espressione completa da valutare. Le seguenti sono espressioni complete: un inizializzatore che non fa parte di un letterale composto (6.7.9); l'espressione in un'espressione (6.8.3); l'espressione di controllo di una dichiarazione di selezione (`if` o `switch`) (6.8.4); l'espressione di controllo di un `while` o `do` statement (6.8.5); ciascuna delle espressioni (facoltative) di una dichiarazione `for` (6.8.5.3); l'espressione (facoltativa) in una dichiarazione di `return` (6.8.6.4).
> - Immediatamente prima che una funzione di libreria ritorni (7.1.4).
> - Dopo le azioni associate a ciascun identificatore di conversione della funzione di input / output formattato (7.21.6, 7.29.2).
> - Immediatamente prima e immediatamente dopo ogni chiamata a una funzione di confronto, e anche tra qualsiasi chiamata a una funzione di confronto e qualsiasi movimento degli oggetti passati come argomenti a quella chiamata (7.22.5).

Examples

Espressioni sequenziate

Le seguenti espressioni sono *sequenziate* :

```
a && b
a || b
a , b
a ? b : c
for ( a ; b ; c ) { ... }
```

In tutti i casi, l'espressione `a` viene valutata completamente e *tutti gli effetti collaterali vengono applicati* prima che sia valutato `b` o `c` . Nel quarto caso, verrà valutato solo uno di `b` o `c` . Nell'ultimo caso, `b` viene completamente valutato e tutti gli effetti collaterali vengono applicati prima che `c` venga valutata.

In tutti i casi, la valutazione dell'espressione `a` *sequenziato prima* le valutazioni di `b` o `c` (alternativamente, le valutazioni di `b` e `c` sono *in sequenza dopo* la valutazione di `a`).

Quindi, espressioni come

```
x++ && x++
x++ ? x++ : y++
(x = f()) && x != 0
for ( x = 0; x < 10; x++ ) { ... }
y = (x++, x++);
```

avere un comportamento ben definito

Espressioni senza conseguenze

C11

Le seguenti espressioni non sono state *seguite* :

```
a + b;
a - b;
a * b;
a / b;
a % b;
a & b;
a | b;
```

Negli esempi precedenti, l'espressione `a` può essere valutata prima o dopo l'espressione `b` , `b` può essere valutata prima di `a` , o possono anche essere mescolati se corrispondono a più istruzioni.

Una regola simile vale per le chiamate di funzione:

```
f(a, b);
```

Qui non solo `a` e `b` sono non in sequenza (cioè la `,` operatore in una chiamata di funzione *non* produce un punto sequenza), ma anche `f` , l'espressione che determina la funzione che deve

essere chiamato.

Gli effetti collaterali possono essere applicati immediatamente dopo la valutazione o posticipati fino a un momento successivo.

Espressioni come

```
x++ & x++;
f(x++, x++); /* the ',' in a function call is *not* the same as the comma operator */
x++ * x++;
a[i] = i++;
```

o

```
x++ & x;
f(x++, x);
x++ * x;
a[i++] = i;
```

produrrà un *comportamento indefinito* perché

- una modifica di un oggetto e qualsiasi altro accesso ad essa deve essere sequenziata
- l'ordine di valutazione e l'ordine in cui vengono applicati gli *effetti collaterali* [1] non è specificato.

---

1 Eventuali cambiamenti nello stato dell'ambiente di esecuzione.

Espressioni a sequenza indeterminata

Le chiamate di funzione come `f(a)` implicano sempre un punto di sequenza tra la valutazione degli argomenti e il designatore (qui `f` e `a`) e la chiamata effettiva. Se due di tali chiamate vengono annullate, le due chiamate di funzione vengono sequenzialmente indeterminate, ovvero, una viene eseguita prima dell'altro e l'ordine non è specificato.

```
unsigned counter = 0;

unsingned account(void) {
   return counter++;
}

int main(void) {
   printf("the order is %u %u\n", account(), account());
}
```

Questa modifica implicita del `counter` durante la valutazione degli argomenti di `printf` è valida, semplicemente non sappiamo quale delle chiamate viene prima. Poiché l'ordine non è specificato, può variare e non può dipendere da. Quindi la stampa potrebbe essere:

> l'ordine è 0 1

o

l'ordine è 1 0

L'affermazione analoga a quanto sopra senza chiamata di funzione intermedia

```
printf("the order is %u %u\n", counter++, counter++); // undefined behavior
```

ha un comportamento indefinito perché non esiste un punto di sequenza tra le due modifiche del `counter` .

Leggi Punti di sequenza online: https://riptutorial.com/it/c/topic/1275/punti-di-sequenza

Capitolo 52: Selezione generica

Sintassi

- _Generico (espressione-assegnazione, elenco-associati-generici)

Parametri

Parametro	Dettagli
generic-assoc-list	associazione generica **O** elenco generico-associato, associazione generica
generic-associazione	nome-tipo: espressione-assegnazione **O** predefinito: espressione-assegnazione

Osservazioni

1. Tutti i qualificatori di tipo verranno eliminati durante la valutazione `_Generic` primaria `_Generic` .
2. `_Generic` primaria generale viene valutata nella fase di traduzione 7 . Quindi fasi come la concatenazione di stringhe sono state completate prima della sua valutazione.

Examples

Controlla se una variabile è di un certo tipo qualificato

```
#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));
}
```

Produzione:

```
i is a const int
j is a non-const int
k is of other type
```

Tuttavia, se il tipo di macro generica è implementato in questo modo:

```
#define is_const_int(x) _Generic((x), \
        const int: "a const int",     \
        int:       "a non-const int", \
        default:   "of other type")
```

L'output è:

```
i is a non-const int
j is a non-const int
k is of other type
```

Questo perché tutti i qualificatori di tipo vengono rilasciati per la valutazione dell'espressione di controllo di un'espressione primaria `_Generic` .

Macro di stampa di tipo generico

```
#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");
}
```

Produzione:

```
int: 42
double: 3.14
unknown argument
```

Si noti che se il tipo non è né `int` né `double` , verrà generato un avviso. Per eliminare l'avviso, è possibile aggiungere quel tipo alla macro di `print(X)` .

Selezione generica basata su più argomenti

Se si desidera una selezione su più argomenti per un'espressione di tipo generico e tutti i tipi in

questione sono tipi aritmetici, un modo semplice per evitare espressioni `_Generic` nidificate `_Generic` nell'utilizzare l'aggiunta dei parametri nell'espressione di controllo:

```
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))
```

Qui, l'espressione di controllo `(X)+(Y)` viene ispezionata solo in base al suo tipo e non viene valutata. Le normali conversioni per gli operandi aritmetici vengono eseguite per determinare il tipo selezionato.

Per situazioni più complesse, è possibile effettuare una selezione in base a più argomenti per l'operatore, annidandoli insieme.

Questo esempio seleziona tra quattro funzioni implementate esternamente, che accettano combinazioni di due argomenti int e / o stringa e restituiscono la loro somma.

```
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 );
}
```

Anche se sembra che l'argomento `y` sia valutato più di una volta, non è [1] . Entrambi gli argomenti vengono valutati una sola volta, alla fine della macro Aggiungi: `( x , y )` , proprio come in una normale chiamata di funzione.

---

[1] (Citato da: ISO: IEC 9899: 201X 6.5.1.1 Selezione generica 3)
L'espressione di controllo di una selezione generica non viene valutata.

Leggi Selezione generica online: https://riptutorial.com/it/c/topic/571/selezione-generica

Capitolo 53: Sequenza di caratteri multi-carattere

Osservazioni

Non tutti i preprocessori supportano l'elaborazione della sequenza trigraph. Alcuni compilatori offrono un'opzione o un passaggio aggiuntivo per elaborarli. Altri usano un programma separato per convertire i trigraph.

Il compilatore GCC non li riconosce a meno che tu non lo richieda esplicitamente di farlo (usa `-trigraphs` per abilitarli; usa `-Wtrigraphs` , parte di `-Wall` , per ricevere avvisi sui trigraphs).

Poiché la maggior parte delle piattaforme in uso oggi supporta l'intera gamma di caratteri singoli utilizzati in C, i digrafi sono preferiti rispetto ai trigrafi, ma l'uso di sequenze di caratteri multi-carattere è generalmente scoraggiato.

Inoltre, fai attenzione all'uso accidentale del trigrafo (`puts("What happened??!!");` ad esempio).

Examples

trigraph

I simboli `[ ] { } ^ \ | ~ #` Sono spesso utilizzati nei programmi C, ma alla fine del 1980, ci sono stati set di codici in uso (ISO 646 varianti, per esempio, nei paesi scandinavi), dove le posizioni dei caratteri ASCII per questi sono stati utilizzati per i caratteri variante linguistica nazionale (ad esempio `£` per `#` nel Regno Unito; `Æ Å æ å ø Ø` per `{ } { } | \` in Danimarca; non ci sono stati `~` in EBCDIC). Ciò significava che era difficile scrivere codice C su macchine che usavano questi set.

Per risolvere questo problema, lo standard C suggeriva l'uso di combinazioni di tre caratteri per produrre un singolo carattere chiamato trigramma. Un trigramma è una sequenza di tre caratteri, i primi due dei quali sono punti interrogativi.

Quello che segue è un semplice esempio che usa sequenze trigraph invece di `#` , `{` e `}` :

```
??=include <stdio.h>

int main()
??<
    printf("Hello World!\n");
??>
```

Questo verrà modificato dal preprocessore C sostituendo i trigrafi con i loro equivalenti a carattere singolo come se il codice fosse stato scritto:

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

```
int main()
{
    printf("Hello World!\n");
}
```

trigraph	Equivalente
?? =	#
?? /	\
??'	^
?? (	[
??)	]
??!	|
?? <	{
??>	}
?? -	~

Si noti che i trigraph sono problematici perché, ad esempio, `??/` è una barra retroversa e può influenzare il significato delle linee di continuazione nei commenti e deve essere riconosciuto all'interno di stringhe e caratteri letterali (ad esempio `'??/??/'` è un singolo carattere, una barra rovesciata).

digrafi

C99

Nel 1994 furono fornite alternative più leggibili a cinque dei trigrafi. Questi usano solo due caratteri e sono conosciuti come digrafi. A differenza dei trigrafi, i digrafi sono token. Se un digraph si verifica in un altro token (ad es. Stringhe letterali o costanti di caratteri), non verrà trattato come un digrafo, ma rimarrà così com'è.

Quanto segue mostra la differenza prima e dopo l'elaborazione della sequenza di digrammi.

```
#include <stdio.h>

int main()
<%
    printf("Hello %> World!\n"); /* Note that the string contains a digraph */
%>
```

Che sarà trattato come:

```
#include <stdio.h>

int main()
{
    printf("Hello %> World!\n"); /* Note the unchanged digraph within the string. */
}
```

digramma	Equivalente
<:	[
:>	]
<%	{
%>	}
%:	#

Leggi Sequenza di caratteri multi-carattere online: https://riptutorial.com/it/c/topic/7111/sequenza-di-caratteri-multi-carattere

Capitolo 54: sindacati

Examples

Differenza tra struttura e unione

Questo dimostra che i membri del sindacato condividono la memoria e che i membri della struttura non condividono la memoria.

```
#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;
}
```

Utilizzare i sindacati per reinterpretare i valori

Alcune implementazioni C consentono al codice di scrivere su un membro di un tipo di unione e poi a leggerlo da un altro per eseguire una sorta di cast di reinterpretazione (analizzando il nuovo tipo come rappresentazione bit di quello vecchio).

È importante notare, tuttavia, che questo non è consentito dallo standard C corrente o passato e risulterà in un comportamento indefinito, tuttavia è un'estensione molto comune offerta dai compilatori (quindi controllare i documenti del compilatore se si prevede di farlo) .

Un vero esempio di questa tecnica è l'algoritmo "Fast Inverse Square Root" che si basa sui

dettagli di implementazione dei numeri in virgola mobile IEEE 754 per eseguire una radice quadrata inversa più rapidamente rispetto all'utilizzo di operazioni in virgola mobile, questo algoritmo può essere eseguito tramite puntatore (che è molto pericoloso e infrange la rigida regola dell'aliasing) o attraverso un'unione (che è ancora un comportamento indefinito ma funziona in molti compilatori):

```
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);
}
```

Questa tecnica è stata ampiamente utilizzata in computer grafica e giochi in passato a causa della sua maggiore velocità rispetto all'utilizzo di operazioni in virgola mobile, ed è molto compromessa, perdendo un po 'di precisione ed essendo molto non portatile in cambio di velocità.

Scrivendo a un membro del sindacato e leggendo da un altro

I membri di un'unione condividono lo stesso spazio in memoria. Ciò significa che la scrittura su un membro sovrascrive i dati in tutti gli altri membri e che la lettura da un membro produce gli stessi dati letti da tutti gli altri membri. Tuttavia, poiché i membri del sindacato possono avere tipi e dimensioni differenti, i dati letti possono essere interpretati in modo diverso, vedi http://www.Scriptutorial.com/c/example/9399/using-unions-to-reinterpret-values

Il semplice esempio qui sotto dimostra un'unione con due membri, entrambi dello stesso tipo. Mostra che scrivendo al membro `m_1` il valore scritto viene letto dal membro `m_2` e la scrittura nel membro `m_2` comporta la lettura del valore scritto dal membro `m_1` .

```
#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;
}
```

Risultato

```
u.m_2: 1
u.m_1: 2
```

Leggi sindacati online: https://riptutorial.com/it/c/topic/7645/sindacati

Capitolo 55: stringhe

introduzione

In C, una stringa non è un tipo intrinseco. Una C-string è la convenzione per avere una matrice unidimensionale di caratteri che viene terminata da un carattere null, da un `'\0'` .

Ciò significa che una stringa C con un contenuto di `"abc"` avrà quattro caratteri `'a'` , `'b'` , `'c'` e `'\0'` .

Vedere l' introduzione di base all'esempio di stringhe .

Sintassi

- char str1 [] = "Ciao, mondo!"; / * Modificabile * /
- char str2 [14] = "Ciao, mondo!"; / * Modificabile * /
- char * str3 = "Ciao, mondo!"; / * Non modificabile * /

Examples

Calcola la lunghezza: 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;
}
```

Questo programma calcola la lunghezza del suo secondo argomento di input e memorizza il risultato in `len` . Quindi stampa quella lunghezza sul terminale. Ad esempio, se eseguito con i parametri `program_name "Hello, world!"` , il programma uscirà `The length of the second argument is 13.` perché la stringa `Hello, world!` è lungo 13 caratteri.

`strlen` conta tutti i **byte** dall'inizio della stringa fino a, ma non incluso, il carattere NUL terminante, `'\0'` . Come tale, può essere utilizzato solo quando la stringa è *garantita* per essere terminata

NUL.

Inoltre, tieni presente che se la stringa contiene caratteri Unicode, `strlen` non ti dirà quanti caratteri ci sono nella stringa (dal momento che alcuni caratteri potrebbero essere più byte). In questi casi, è necessario contare i caratteri (*es* . Unità di codice) da soli. Considera l'output del seguente esempio:

```
#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));
}
```

Produzione:

```
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
```

Copia e concatenazione: 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;
}
```

Uscite:

```
foo
foobar
bar
```

Se si aggiunge o si copia da una stringa esistente, assicurarsi che sia NUL-terminato!

Le stringhe letterali (ad es. `"foo"`) saranno sempre terminate NUL dal compilatore.

Comparsa: strcmp (), strncmp (), strcasecmp (), strncasecmp ()

Le `strcase*` non sono Standard C, ma un'estensione POSIX.

La funzione `strcmp` confronta lessicograficamente due array di caratteri con terminazione null. Le funzioni restituiscono un valore negativo se il primo argomento compare prima del secondo in ordine lessicografico, zero se si confronta uguale o positivo se il primo argomento compare dopo il secondo in ordine lessicografico.

```
#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;
}
```

Uscite:

```
BBB equals BBB
BBB comes before CCCCC
```

```
BBB comes after AAAAAA
```

Come `strcmp` , la funzione `strcasecmp` confronta anche lessicograficamente i suoi argomenti dopo aver tradotto ciascun carattere nel suo corrispondente in lettere minuscole:

```
#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;
}
```

Uscite:

```
BBB equals bBB
BBB comes before ccCCC
BBB comes after aaaaaa
```

`strncmp` e `strncasecmp` confrontano al massimo n caratteri:

```
#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;
```

```
}
```

Uscite:

```
BBB equals Bb
BBB comes before Bb
BBB comes before Bb
```

Tokenizzazione: strtok (), strtok_r () e strtok_s ()

La funzione `strtok` spezza una stringa in stringhe più piccole, o token, usando un set di delimitatori.

```
#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" */
}
```

Produzione:

```
1: [Hello]
2: [world]
```

La stringa di delimitatori può contenere uno o più delimitatori e diverse stringhe delimitatori possono essere utilizzate con ogni chiamata a `strtok` .

Le chiamate a `strtok` per continuare a tokenizzare la stessa stringa sorgente non dovrebbero passare di nuovo la stringa sorgente, ma passare `NULL` come primo argomento. Se *viene* passata la stessa stringa sorgente, il primo token verrà invece ridenominato. Cioè, dati gli stessi delimitatori, `strtok` restituirebbe semplicemente il primo token di nuovo.

Notare che come `strtok` non alloca nuova memoria per i token, *modifica la stringa di origine* . Cioè, nell'esempio precedente, la stringa `src` verrà manipolata per produrre i token a cui fa riferimento il puntatore restituito dalle chiamate a `strtok` . Ciò significa che la stringa di origine non può essere `const` (quindi non può essere una stringa letterale). Significa anche che l'identità del byte di delimitazione viene persa (cioè nell'esempio il "," e "!" Vengono effettivamente eliminati dalla stringa di origine e non è possibile stabilire quale carattere delimitatore corrisponde).

Si noti inoltre che più delimitatori consecutivi nella stringa di origine vengono considerati come

uno; nell'esempio, la seconda virgola viene ignorata.

`strtok` non è né thread-safe né ri-entrant perché usa un buffer statico durante l'analisi. Ciò significa che se una funzione chiama `strtok` , nessuna funzione che chiama mentre sta usando `strtok` può anche usare `strtok` , e non può essere chiamata da alcuna funzione che sta usando `strtok` .

Un esempio che dimostra i problemi causati dal fatto che `strtok` non è rientrante è il seguente:

```
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);
```

Produzione:

```
[1.2]
 [1]
 [2]
```

L'operazione prevista è che il ciclo `do while` esterno crei tre token costituiti da ogni stringa decimale (`"1.2"` , `"3.5"` , `"4.2"`), per ognuno dei quali lo `strtok` chiama il ciclo interno dovrebbe dividerlo in un separato stringhe di cifre (`"1"` , `"2"` , `"3"` , `"5"` , `"4"` , `"2"`).

Tuttavia, poiché `strtok` non è rientranti, ciò non si verifica. Invece il primo `strtok` crea correttamente il token "1.2 \ 0" e il ciclo interno crea correttamente i token `"1"` e `"2"` . Ma poi lo `strtok` nel ciclo esterno è alla fine della stringa usata dal ciclo interno e restituisce immediatamente NULL. La seconda e la terza sottostringa dell'array `src` non vengono affatto analizzate.

C11

Le librerie C standard non contengono una versione thread-safe o re-entrant ma altre, come POSIX ' `strtok_r` . Nota che su MSVC l'equivalente `strtok` , `strtok_s` è thread-safe.

C11

C11 ha una parte opzionale, Annex K, che offre una versione thread-safe e re-entry chiamata `strtok_s` . Puoi provare la funzione con `__STDC_LIB_EXT1__` . Questa parte facoltativa non è ampiamente supportata.

La funzione `strtok_s` differisce dalla funzione `strtok_r` POSIX `strtok_r` dall'archiviazione all'esterno della stringa che viene tokenizzata e controllando i vincoli di runtime. Sui programmi scritti correttamente, però, `strtok_s` e `strtok_r` comportano allo stesso modo.

Usando `strtok_s` con l'esempio ora si ottiene la risposta corretta, in questo modo:

```
/* 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);
```

E l'output sarà:

```
[1.2]
 [1]
 [2]
[3.5]
 [3]
 [5]
[4.2]
 [4]
 [2]
```

Trova la prima / ultima occorrenza di un carattere specifico: strchr (), strrchr ()

Le funzioni `strchr` e `strrchr` trovano un carattere in una stringa, cioè in una matrice di caratteri con terminazione NUL. `strchr` restituisce un puntatore alla prima occorrenza e `strrchr` all'ultima.

```
#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;
}
```

Output (dopo aver generato un eseguibile chiamato `pos`):

```
$ ./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.
```

Un uso comune per `strrchr` è estrarre un nome di file da un percorso. Ad esempio per estrarre `myfile.txt` da `C:\Users\eak\myfile.txt` :

```
char *getFileName(const char *path)
{
    char *pend;

    if ((pend = strrchr(path, '\')) != NULL)
        return pend + 1;

    return NULL;
}
```

Iterating Over the Characters in a String

Se conosciamo la lunghezza della stringa, possiamo usare un ciclo for per scorrere i suoi caratteri:

```
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. */
}
```

In alternativa, possiamo usare la funzione standard `strlen()` per ottenere la lunghezza di una stringa se non sappiamo quale sia la stringa:

```
size_t length = strlen(string);
size_t i = 0;
for (; i < length; i++) {
    printf("%c\n", string[i]);    /* Print each character of the string. */
}
```

Infine, possiamo approfittare del fatto che le stringhe in C sono garantite per essere terminate con null (cosa che abbiamo già fatto passandole a `strlen()` nell'esempio precedente ;-)). Possiamo scorrere l'array indipendentemente dalle sue dimensioni e interrompere l'iterazione una volta raggiunto un carattere null:

```
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++;
}
```

Introduzione di base alle stringhe

In C, una **stringa** è una sequenza di caratteri che viene terminata da un carattere nullo ('\ 0').

Possiamo creare stringhe usando **stringhe letterali** , che sono sequenze di caratteri circondate da virgolette doppie; per esempio, prendi la stringa letterale `"hello world"` . I valori letterali stringa vengono automaticamente annullati.

Possiamo creare stringhe usando diversi metodi. Ad esempio, possiamo dichiarare un `char *` e inizializzarlo per puntare al primo carattere di una stringa:

```
char * string = "hello world";
```

Quando si inizializza un `char *` su una costante di stringa come sopra, la stringa stessa viene solitamente allocata in dati di sola lettura; `string` è un puntatore al primo elemento dell'array, che è il carattere `'h'` .

Poiché la stringa letterale è allocata nella memoria di sola lettura, non è modificabile [1] . Qualsiasi tentativo di modificarlo porterà a comportamenti non definiti , quindi è meglio aggiungere `const` per ottenere un errore in fase di compilazione come questo

```
char const * string = "hello world";
```

Ha un effetto simile a [2] come

```
char const string_arr[] = "hello world";
```

Per creare una stringa modificabile, puoi dichiarare una matrice di caratteri e inizializzarne il contenuto usando una stringa letterale, in questo modo:

```
char modifiable_string[] = "hello world";
```

Questo è equivalente al seguente:

```
char modifiable_string[] = {'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\0'};
```

Poiché la seconda versione utilizza l'inizializzatore racchiuso tra virgolette, la stringa non viene automaticamente terminata da null a meno che un carattere `'\0'` sia incluso esplicitamente nell'array di caratteri in genere come ultimo elemento.

---

1 Non modificabile implica che i caratteri nella stringa letterale non possono essere modificati, ma ricorda che la `string` del puntatore può essere modificata (può indicare da qualche altra parte o può essere incrementata o decrementata).

2 Entrambe le stringhe hanno un effetto simile, nel senso che i caratteri di entrambe le stringhe non possono essere modificati. Dovrebbe essere notato che la `string` è un puntatore al `char` ed è un valore l modificabile in modo che possa essere incrementato o puntare a qualche altra posizione mentre l'array `string_arr` è un valore-l non modificabile, non può essere modificato.

Creazione di matrici di stringhe

Una serie di stringhe può significare un paio di cose:

1. Un array i cui elementi sono `char *` s
2. Una matrice i cui elementi sono matrici di `char`

Possiamo creare una serie di puntatori di caratteri in questo modo:

```
char * string_array[] = {
    "foo",
    "bar",
    "baz"
};
```

Ricorda: quando assegniamo stringhe letterali a `char *` , le stringhe vengono allocate nella memoria di sola lettura. Tuttavia, l'array `string_array` è allocato nella memoria di lettura / scrittura. Ciò significa che possiamo modificare i puntatori dell'array, ma non possiamo modificare le stringhe a cui puntano.

In C, il parametro di main `argv` (la matrice di argomenti della riga di comando passati quando il

programma è stato eseguito) è un array di `char *` : `char * argv[]` .

Possiamo anche creare array di matrici di caratteri. Poiché le stringhe sono matrici di caratteri, una serie di stringhe è semplicemente una matrice i cui elementi sono matrici di caratteri:

```
char modifiable_string_array_literals[][4] = {
    "foo",
    "bar",
    "baz"
};
```

Questo è equivalente a:

```
char modifiable_string_array[][4] = {
    {'f', 'o', 'o', '\0'},
    {'b', 'a', 'r', '\0'},
    {'b', 'a', 'z', '\0'}
};
```

Si noti che si specifica `4` come dimensione della seconda dimensione dell'array; ciascuna delle stringhe nel nostro array è in realtà 4 byte poiché è necessario includere il carattere con terminazione nulla.

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` ricerca l'argomento `haystack` (primo) per la stringa puntata `needle` . Se trovato, `strstr` restituisce l'indirizzo dell'occorrenza. Se non riesce a trovare l' `needle` , restituisce NULL. Usiamo `zbpos` modo che non continuiamo a trovare lo stesso ago più e più volte. Per saltare la prima istanza, aggiungiamo un offset di `zbpos` . Un clone di Blocco note potrebbe chiamare `findnext` come questo, al fine di implementare il suo dialogo "Trova successivo":

```
/*
    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" */
}
```

Stringhe letterali

I valori letterali delle stringhe rappresentano array di `char` statici a terminazione nulla di `char` . Poiché hanno una durata di archiviazione statica, una stringa letterale o un puntatore allo stesso array sottostante può essere tranquillamente utilizzata in diversi modi che non è possibile per un puntatore a un array automatico. Ad esempio, restituire una stringa letterale da una funzione ha un comportamento ben definito:

```
const char *get_hello() {
    return "Hello, World!";  /* safe */
}
```

Per ragioni storiche, gli elementi dell'array corrispondenti a una stringa letterale non sono formalmente `const` . Tuttavia, qualsiasi tentativo di modificarli ha un comportamento indefinito . In genere, un programma che tenta di modificare la matrice corrispondente a una stringa letterale si bloccherà o funzionerà in altro modo.

```
char *foo = "hello";
foo[0] = 'y';  /* Undefined behavior - BAD! */
```

Laddove un puntatore punta a una stringa letterale - o dove a volte può farlo - è consigliabile dichiarare il `const` referente di quel puntatore per evitare di coinvolgere accidentalmente un comportamento non definito.

```
const char *foo = "hello";
/* GOOD: can't modify the string pointed to by foo */
```

D'altra parte, un puntatore verso o nell'array sottostante di un letterale stringa non è di per sé intrinsecamente speciale; il suo valore può essere liberamente modificato per indicare qualcos'altro:

```
char *foo = "hello";
foo = "World!"; /* OK - we're just changing what foo points to */
```

Inoltre, sebbene gli inizializzatori per i `char` di tipo `char` possano avere la stessa forma dei letterali di stringa, l'uso di tale inizializzatore non conferisce le caratteristiche di una stringa letterale sull'array inizializzato. L'inizializzatore indica semplicemente la lunghezza e il contenuto iniziale dell'array. In particolare, gli elementi sono modificabili se non dichiarati esplicitamente `const` :

```
char foo[] = "hello";
```

```
foo[0] = 'y';  /* OK! */
```

Azzeramento di una stringa

È possibile chiamare `memset` per azzerare una stringa (o qualsiasi altro blocco di memoria).

Dove `str` è la stringa da azzerare e `n` è il numero di byte nella stringa.

```
#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;
}
```

stampe:

```
'fortytwo'
''
```

Un altro esempio:

```
#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;
}
```

stampe:

```
'fortytwo'
'forty'
''
```

strspn e strcspn

Data una stringa, `strspn` calcola la lunghezza della sottostringa iniziale (span) costituita esclusivamente da un elenco specifico di caratteri. `strcspn` è simile, tranne che calcola la lunghezza della sottostringa iniziale costituita da qualsiasi carattere tranne quelli elencati:

```
/*
  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;
}
```

Funzioni analoghe che utilizzano stringhe di caratteri ampi sono `wcsspn` e `wcscspn` ; sono usati allo stesso modo.

Copia di stringhe

Le assegnazioni di puntatore non copiano le stringhe

È possibile utilizzare l'operatore `=` per copiare gli interi, ma non è possibile utilizzare l'operatore `=` per copiare le stringhe in C. Le stringhe in C sono rappresentate come matrici di caratteri con un carattere null terminante, quindi l'utilizzo dell'operatore `=` salverà solo l'indirizzo (puntatore) di una stringa.

```
#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;
}
```

L'esempio precedente è stato compilato perché abbiamo usato `char *d` piuttosto che `char d[3]` . L'utilizzo di quest'ultimo causerebbe un errore del compilatore. Non è possibile assegnare agli array in C.

```
#include <stdio.h>

int main(void) {
    char a[] = "abc";
    char b[8];

    b = a; /* compile error */
    printf("%s\n", b);

    return 0;
}
```

Copia di stringhe usando funzioni standard

strcpy()

Per copiare effettivamente le stringhe, la funzione `strcpy()` è disponibile in `string.h` . Abbastanza spazio deve essere assegnato per la destinazione prima di copiare.

```
#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()

Per evitare il sovraccarico del buffer, è possibile utilizzare `snprintf()` . Non è la migliore soluzione per quanto riguarda le prestazioni dal momento che deve analizzare la stringa del modello, ma è l'unica funzione di protezione dal limite del buffer per copiare stringhe prontamente disponibili nella libreria standard, che può essere utilizzata senza ulteriori passaggi.

```
#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()

Una seconda opzione, con prestazioni migliori, consiste nell'usare `strncat()` (una versione di controllo dell'overflow del buffer di `strcat()`) - richiede un terzo argomento che indica il numero massimo di byte da copiare:

```
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 */
```

Nota che questa formulazione usa `sizeof(dest) - 1` ; questo è cruciale perché `strncat()` aggiunge sempre un byte null (buono), ma non lo considera nella dimensione della stringa (causa di confusione e sovrascritture del buffer).

Si noti inoltre che l'alternativa - concatenare dopo una stringa non vuota - è ancora più complessa. Tenere conto:

```
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);
```

L'output è:

```
23: [Clownfish: Marvin and N]
```

Si noti, tuttavia, che la dimensione specificata come lunghezza *non* era la dimensione dell'array di destinazione, ma la quantità di spazio rimasto al suo interno, senza contare il byte null del terminale. Questo può causare grossi problemi di sovrascrittura. È anche un po 'dispendioso; per specificare correttamente l'argomento della lunghezza, si conosce la lunghezza dei dati nella destinazione, quindi è possibile specificare l'indirizzo del byte nullo alla fine del contenuto esistente, salvando `strncat()` dalla nuova scansione:

```
    strcpy(dst, "Clownfish: ");
    assert(len < sizeof(dst) - 1);
    strncat(dst + len, src, sizeof(dst) - len - 1);
    printf("%zu: [%s]\n", strlen(dst), dst);
```

Questo produce lo stesso risultato di prima, ma `strncat()` non deve eseguire la scansione del contenuto esistente di `dst` prima che inizi a copiare.

strncpy()

L'ultima opzione è la funzione `strncpy()` . Anche se si potrebbe pensare che dovrebbe venire prima, è una funzione piuttosto ingannevole che ha due trucchi principali:

1. Se la copia tramite `strncpy()` colpisce il limite del buffer, non verrà scritto un carattere null terminante.
2. `strncpy()` riempie sempre completamente la destinazione, con byte null se necessario.

(Tale implementazione bizzarra è storica e inizialmente era pensata per gestire nomi di file UNIX)

L'unico modo corretto per usarlo è quello di garantire manualmente la terminazione null:

```
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 */
```

Anche in questo caso, se si dispone di un buffer di grandi dimensioni diventa molto inefficiente utilizzare `strncpy()` causa di ulteriore riempimento nullo.

Converti stringhe in numero: atoi (), atof () (pericoloso, non usarle)

Avvertenza: le funzioni `atoi` , `atol` , `atoll` e `atof` sono intrinsecamente insicuri, poiché: *Se il valore del risultato non può essere rappresentato, il comportamento non è definito.* (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;
}
```

Quando la stringa da convertire è un intero decimale valido compreso nell'intervallo, la funzione funziona:

```
$ ./atoi 100
String value = 100, Int value = 100
$ ./atoi 200
String value = 200, Int value = 200
```

Per le stringhe che iniziano con un numero, seguito da qualcos'altro, viene analizzato solo il numero iniziale:

```
$ ./atoi 0x200
0
$ ./atoi 0123x300
123
```

In tutti gli altri casi, il comportamento non è definito:

```
$ ./atoi hello
Formatting the hard disk...
```

A causa delle ambiguità di cui sopra e di questo comportamento indefinito, la famiglia di funzioni `atoi` non dovrebbe mai essere utilizzata.

- Per convertire in `long int` , usa `strtol()` invece di `atol()` .
- Per convertire in `double` , usa `strtod()` invece di `atof()` .

C99

- Per convertire in `long long int` , usa `strtoll()` invece di `atoll()` .

lettura / scrittura di dati formattati a stringa

Scrivi dati formattati su stringa

```
int sprintf ( char * str, const char * format, ... );
```

usa la funzione `sprintf` per scrivere i dati float sulla stringa.

```
#include <stdio.h>
int main ()
{
  char buffer [50];
  double PI = 3.1415926;
  sprintf (buffer, "PI = %.7f", PI);
  printf ("%s\n",buffer);
  return 0;
}
```

Leggi i dati formattati dalla stringa

```
int sscanf ( const char * s, const char * format, ...);
```

usa la funzione `sscanf` per analizzare i dati formattati.

```
#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;
}
```

Converti in sicurezza le stringhe in numero: funzioni strtoX

C99

Dal momento che C99 la libreria C ha un set di funzioni di conversione sicure che interpretano una

stringa come un numero. I loro nomi sono di forma `strtoX` , dove `X` è uno di `l` , `ul` , `d` , ecc per determinare il tipo di destinazione della conversione

```
double strtod(char const* p, char** endptr);
long double strtold(char const* p, char** endptr);
```

Forniscono il controllo che una conversione ha avuto un over o underflow:

```
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 */
```

Se la stringa in effetti non contiene alcun numero, questo uso di `strtod` restituisce `0.0` .

Se ciò non è soddisfacente, è possibile utilizzare il parametro addizionale `endptr` . È un puntatore al puntatore che verrà puntato alla fine del numero rilevato nella stringa. Se è impostato su `0` , come sopra, o `NULL` , viene semplicemente ignorato.

Questo parametro `endptr` indica se è avvenuta una conversione corretta e, in caso affermativo, dove il numero è terminato:

```
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 */
```

Esistono funzioni analoghe per la conversione in tipi interi più ampi:

```
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);
```

Queste funzioni hanno un terzo parametro `nbase` che contiene la base numerica in cui è scritto il numero.

```
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 */
```

Il valore speciale `0` per `nbase` significa che la stringa viene interpretata nello stesso modo in cui i numeri letterali vengono interpretati in un programma C: un prefisso di `0x` corrisponde a una rappresentazione esadecimale, altrimenti uno `0` è ottale e tutti gli altri numeri sono visti come decimali.

Quindi il modo più pratico per interpretare un argomento da linea di comando come un numero sarebbe

```
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;
}
```

Ciò significa che il programma può essere chiamato con un parametro in ottale, decimale o esadecimale.

Leggi stringhe online: https://riptutorial.com/it/c/topic/1990/stringhe

Capitolo 56: Structs

introduzione

Le strutture forniscono un modo per raggruppare una serie di variabili correlate di tipi diversi in una singola unità di memoria. La struttura nel suo insieme può essere referenziata da un singolo nome o puntatore; anche i membri della struttura possono essere consultati individualmente. Le strutture possono essere passate alle funzioni e restituite dalle funzioni. Sono definiti utilizzando la `struct` parole chiave.

Examples

Strutture dati semplici

I tipi di dati strutturali sono utili per raggruppare i dati correlati e comportarsi come una singola variabile.

Dichiarazione di una `struct` semplice che contiene due membri `int` :

```
struct point
{
    int x;
    int y;
};
```

`x` e `y` sono chiamati *membri* (o *campi*) della struttura `point` .

Definire e usare le strutture:

```
struct point p;    // declare p as a point struct
p.x = 5;           // assign p member variables
p.y = 3;
```

Le strutture possono essere inizializzate alla definizione. Quanto sopra è equivalente a:

```
struct point p = {5, 3};
```

Le strutture possono anche essere inizializzate usando gli inizializzatori designati .

L'accesso ai campi viene fatto anche usando il . operatore

```
printf("point is (x = %d, y = %d)", p.x, p.y);
```

Typedef Structs

Combinare `typedef` con `struct` può rendere il codice più chiaro. Per esempio:

```
typedef struct
{
    int x, y;
} Point;
```

al contrario di:

```
struct Point
{
    int x, y;
};
```

potrebbe essere dichiarato come:

```
Point point;
```

invece di:

```
struct Point point;
```

Ancora meglio è usare il seguente

```
typedef struct Point Point;

struct Point
{
    int x, y;
};
```

avere il vantaggio di entrambe le possibili definizioni di `point` . Tale dichiarazione è più comoda se hai imparato prima il C ++, dove puoi omettere la parola chiave `struct` se il nome non è ambiguo.

`typedef` nomi `typedef` per le strutture potrebbero essere in conflitto con altri identificatori di altre parti del programma. Alcuni considerano questo uno svantaggio, ma per la maggior parte delle persone che hanno una `struct` e un altro identificatore lo stesso è piuttosto fastidioso. Notoria è ad esempio POSIX ' `stat`

```
int stat(const char *pathname, struct stat *buf);
```

dove vedi una funzione `stat` che ha un argomento che è `struct stat` .

`typedef` 'd senza nome di tag impongono sempre che l'intera dichiarazione `struct` sia visibile al codice che la usa. L'intera dichiarazione della `struct` deve quindi essere inserita in un file di intestazione.

Tenere conto:

```
#include "bar.h"

struct foo
```

```
{
    bar *aBar;
};
```

Quindi con una `typedef` d `struct` che non ha il nome del tag, il file `bar.h` deve sempre includere l'intera definizione di `bar` . Se usiamo

```
typedef struct bar bar;
```

in `bar.h` , i dettagli della struttura della `bar` possono essere nascosti.

Vedi Typedef

Puntatori alle strutture

Quando hai una variabile contenente una `struct` , puoi accedere ai suoi campi usando l'operatore punto (`.`). Tuttavia, se hai un puntatore a una `struct` , questo non funzionerà. Devi usare l'operatore della freccia (`->`) per accedere ai suoi campi. Ecco un esempio di un'implementazione terribilmente semplice (alcuni potrebbero dire "terribile e semplice") di uno stack che utilizza i puntatori per `struct` e dimostrare l'operatore della freccia.

```
#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);
    }
}
```

Membri di array flessibili

C99

Tipo di dichiarazione

Una struttura *con almeno un membro* può contenere inoltre un singolo membro di matrice di lunghezza non specificata alla fine della struttura. Questo è chiamato membro di un array flessibile:

```
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[];
};
```

Effetti su dimensioni e riempimento

Un membro di array flessibile viene considerato privo di dimensioni quando si calcola la dimensione di una struttura, sebbene il riempimento tra quel membro e il membro precedente della struttura possa ancora esistere:

```
/* 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));
```

Si considera che il membro di array flessibile abbia un tipo di array incompleto, quindi la sua dimensione non può essere calcolata utilizzando `sizeof` .

uso

È possibile dichiarare e inizializzare un oggetto con un tipo di struttura contenente un membro di array flessibile, ma non si deve tentare di inizializzare il membro di matrice flessibile poiché viene trattato come se non esistesse. È vietato provare a fare ciò e si verificheranno errori di compilazione.

Allo stesso modo, non si dovrebbe tentare di assegnare un valore a qualsiasi elemento di un membro di matrice flessibile quando si dichiara una struttura in questo modo poiché potrebbe non esserci abbastanza riempimento alla fine della struttura per consentire qualsiasi oggetto richiesto dal membro di array flessibile. Tuttavia, il compilatore non ti impedirà necessariamente di farlo, quindi questo può portare a un comportamento indefinito.

```
/* 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 */
```

Puoi invece scegliere di usare `malloc` , `calloc` o `realloc` per allocare la struttura con spazio di archiviazione aggiuntivo e in seguito liberarla, che ti consente di utilizzare il membro della matrice flessibile come desideri:

```
/* 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

La 'struct hack'

I membri di array flessibili non esistevano prima di C99 e sono considerati come errori. Una soluzione comune consiste nel dichiarare una matrice di lunghezza 1, una tecnica chiamata 'struct hack':

```
struct ex1
{
    size_t foo;
    int flex[1];
};
```

Ciò influenzerà la dimensione della struttura, tuttavia, a differenza di un vero membro della matrice flessibile:

```
/* 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));
```

Per usare il membro `flex` come membro di un array flessibile, lo `sizeof(*pe1)` con `malloc` come mostrato sopra, eccetto che `sizeof(*pe1)` (o la dimensione equivalente `sizeof(struct ex1)`) verrebbe sostituito con `offsetof(struct ex1, flex)` o il più lungo, type-agnostic expression `sizeof(*pe1)-sizeof(pe1->flex)` . In alternativa, è possibile sottrarre 1 dalla lunghezza desiderata della matrice "flessibile" poiché è già inclusa nella dimensione della struttura, assumendo che la lunghezza desiderata sia maggiore di 0. La stessa logica può essere applicata agli altri esempi di utilizzo.

Compatibilità

Se si desidera la compatibilità con i compilatori che non supportano membri di array flessibili, è possibile utilizzare una macro definita come `FLEXMEMB_SIZE` seguito:

```
#if __STDC_VERSION__ < 199901L
#define FLEXMEMB_SIZE 1
#else
#define FLEXMEMB_SIZE /* nothing */
#endif

struct ex1
{
    size_t foo;
    int flex[FLEXMEMB_SIZE];
```

```
};
```

Quando si assegnano gli oggetti, è necessario utilizzare il `offsetof(struct ex1, flex)` per fare riferimento alle dimensioni della struttura (escluso il membro della matrice flessibile) poiché è l'unica espressione che rimarrà coerente tra i compilatori che supportano membri di array flessibili e compilatori che eseguono non:

```
struct ex1 *pe10 = malloc(offsetof(struct ex1, flex) + n * sizeof(pe10->flex[0]));
```

L'alternativa è usare il preprocessore per sottrarre condizionatamente 1 dalla lunghezza specificata. A causa dell'aumentato potenziale di incoerenza e errore umano generale in questa forma, ho spostato la logica in una funzione separata:

```
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);
```

Passare le strutture alle funzioni

In C, tutti gli argomenti vengono passati alle funzioni in base al valore, incluse le strutture. Per le piccole strutture, questa è una buona cosa in quanto significa che non vi è alcun sovraccarico di accesso ai dati attraverso un puntatore. Tuttavia, rende anche molto facile passare accidentalmente un'enorme struttura con prestazioni scadenti, in particolare se il programmatore viene utilizzato in altre lingue in cui gli argomenti vengono passati per riferimento.

```
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;
}
```

Programmazione basata su oggetti mediante le strutture

Le strutture possono essere utilizzate per implementare il codice in modo orientato agli oggetti. Una struct è simile a una classe, ma mancano le funzioni che normalmente fanno parte di una classe, possiamo aggiungerle come variabili dei membri del puntatore di funzione. Per rimanere con il nostro esempio di coordinate:

```
/* 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);
```

E ora il file C di implementazione:

```
/* 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");
    }
}
```

Un esempio di utilizzo della nostra classe di coordinate sarebbe:

```
/* 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;
}
```

Leggi Structs online: https://riptutorial.com/it/c/topic/1119/structs

Capitolo 57: Strutture di prova

introduzione

Molti sviluppatori usano i test unitari per verificare che il loro software funzioni come previsto. I test unitari controllano piccole unità di pezzi di software più grandi e assicurano che le uscite corrispondano alle aspettative. Le strutture di test rendono più semplice il collaudo delle unità fornendo servizi di set-up / tear-down e coordinando i test.

Esistono molti framework di test unitari disponibili per C. Ad esempio, Unity è un framework C puro. Le persone usano spesso framework di test C ++ per testare il codice C; ci sono anche molti framework di test C ++.

Osservazioni

Collaudare l'imbragatura:

TDD - Test Driven Development:

Prova i doppi meccanismi in C:

1. Sostituzione del tempo di collegamento
2. Sostituzione del puntatore funzione
3. Sostituzione del preprocessore
4. Sostituzione combinata del puntatore del tempo di collegamento e della funzione

Nota sui framework di test C ++ usati in C: Usare framework C ++ per testare un programma C è una pratica abbastanza comune come spiegato qui .

Examples

CppUTest

CppUTest è un framework xUnit per unità di test C e C ++. È scritto in C ++ e mira alla portabilità e alla semplicità del design. Supporta il rilevamento delle perdite di memoria, crea mock e esegue i test insieme a Google Test. Viene fornito con script di supporto e progetti di esempio per Visual Studio ed 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);
}
```

Un gruppo di prova può avere un metodo `setup()` e `teardown()` . Il metodo di `setup` viene chiamato prima di ogni test e viene chiamato il metodo `teardown()` . Entrambi sono opzionali e possono essere omessi indipendentemente. Altri metodi e variabili possono anche essere dichiarati all'interno di un gruppo e saranno disponibili per tutti i test di quel gruppo.

```
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 è un framework di test in stile xUnit per il test dell'unità C. È scritto completamente in C ed è portatile, rapido, semplice, espressivo ed estensibile. È progettato per essere particolarmente utile anche per il test dell'unità per sistemi embedded.

Un semplice caso di test che controlla il valore restituito da una funzione potrebbe apparire come segue

```
void test_FunctionUnderTest_should_ReturnFive(void)
{
    TEST_ASSERT_EQUAL_INT( 5, FunctionUnderTest() );
}
```

Un file di test completo potrebbe essere simile a:

```
#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 viene fornito con alcuni progetti di esempio, makefile e alcuni script di Ruby Rake che aiutano a rendere un po 'più semplice la creazione di file di test più lunghi.

CMocka

CMocka è un elegante framework di test unitario per C con supporto per oggetti finti. Richiede solo la libreria C standard, funziona su una gamma di piattaforme di elaborazione (incluso embedded) e con diversi compilatori. Ha un tutorial su test con mock, documentazione API e una varietà di esempi .

```
#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;
}
```

Leggi Strutture di prova online: https://riptutorial.com/it/c/topic/6779/strutture-di-prova

Capitolo 58: Tipi di dati

Osservazioni

- Mentre `char` è richiesto per essere 1 byte, 1 byte **non** è richiesto per essere 8 bit (spesso chiamato anche un *ottetto*), anche se la maggior parte delle moderne piattaforme di computer lo definiscono come 8 bit. Il numero di bit di implementazione per `char` è fornito dalla macro `CHAR_BIT` , definita in `<limits.h>` . POSIX richiede 1 byte per essere 8 bit.
- I tipi di interi a larghezza fissa dovrebbero essere usati scarsamente, i tipi di C incorporati sono progettati per essere naturali su ogni architettura, i tipi a larghezza fissa dovrebbero essere usati solo se avete bisogno esplicitamente di un numero intero specifico (ad esempio per il networking).

Examples

Tipi interi e costanti

Gli interi firmati possono essere di questi tipi (l' `int` dopo l' `short` o l'opzione `long` è facoltativo):

```
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 */
```

Ognuno di questi tipi di interi con segno ha una versione senza firma.

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

Per tutti i tipi, ma `char` la `signed` versione si presume se il `signed` o `unsigned` parte viene omesso. Il tipo `char` costituisce un terzo tipo di carattere, diverso dal `signed char` e dal `signed char` `unsigned char` e la signness (o meno) dipende dalla piattaforma.

Diversi tipi di costanti intere (chiamate *letterali* in C jargon) possono essere scritti in basi diverse e larghezza diversa, in base al loro prefisso o suffisso.

```
/* 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) */
```

Le costanti decimali sono sempre `signed` . Le costanti esadecimali iniziano con `0x` o `0X` e le costanti ottali iniziano solo con uno `0` . Gli ultimi due sono `signed` o `unsigned` seconda che il valore si adatti al tipo firmato o meno.

```
/* 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 = 655361; /* l or L represent long int */
```

Senza un suffisso la costante ha il primo tipo che si adatta al suo valore, cioè una costante decimale che è maggiore di `INT_MAX` è di tipo `long` se possibile, o `long long` altrimenti.

Il file di intestazione `<limits.h>` descrive i limiti degli interi come segue. I loro valori definiti dall'implementazione devono essere uguali o maggiori in grandezza (valore assoluto) a quelli mostrati sotto, con lo stesso segno.

macro	genere	Valore
`CHAR_BIT`	l'oggetto più piccolo che non è un bit-field (byte)	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`	vedi sotto
`CHAR_MAX`	`char`	vedi sotto
`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

macro	genere	Valore
`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

Se il valore di un oggetto di tipo `char` sign-extends quando utilizzato in un'espressione, il valore di `CHAR_MIN` deve essere uguale a quello di `SCHAR_MIN` e il valore di `CHAR_MAX` deve essere uguale a quello di `SCHAR_MAX` . Se il valore di un oggetto di tipo `char` non si estende al segno quando utilizzato in un'espressione, il valore di `CHAR_MIN` deve essere 0 e il valore di `CHAR_MAX` deve essere uguale a quello di `UCHAR_MAX` .

C99

Lo standard C99 ha aggiunto una nuova intestazione, `<stdint.h>` , che contiene definizioni per numeri interi a larghezza fissa. Vedere l'esempio intero a larghezza fissa per una spiegazione più approfondita.

String letterali

Una stringa letterale in C è una sequenza di caratteri, terminata da uno zero letterale.

```
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' */
```

I valori letterali delle stringhe **non** sono **modificabili** (e infatti possono essere inseriti nella memoria di sola lettura come .rodata). Il tentativo di modificare i loro valori comporta un comportamento indefinito.

```
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! */
```

I letterali a più stringhe sono concatenati in fase di compilazione, il che significa che puoi scrivere un costrutto come questi.

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 */
```

I valori letterali stringa, come le costanti dei caratteri, supportano set di caratteri diversi.

```
/* 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";
```

Tipi interi a larghezza fissa (dal C99)

C99

L'intestazione `<stdint.h>` fornisce diverse definizioni di tipi interi a larghezza fissa. Questi tipi sono *facoltativi* e vengono forniti solo se la piattaforma ha un tipo intero della larghezza corrispondente e se il corrispondente tipo firmato ha una rappresentazione a complemento a due di valori negativi.

Vedere la sezione commenti per suggerimenti sull'utilizzo di tipi di larghezza fissa.

```
/* 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 */
```

Costanti a virgola mobile

Il linguaggio C ha tre tipi di virgola mobile reali obbligatori, `float` , `double` e `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 */
```

L'intestazione `<float.h>` definisce vari limiti per le operazioni in virgola mobile.

L'aritmetica in virgola mobile è definita dall'implementazione. Tuttavia, la maggior parte delle piattaforme moderne (arm, x86, x86_64, MIPS) utilizzano operazioni in virgola mobile IEEE 754 .

C ha anche tre tipi di floating point complessi opzionali derivati da quanto sopra.

Interpretazione delle dichiarazioni

Una peculiarità sintattica distintiva di C è che le dichiarazioni rispecchiano l'uso dell'oggetto dichiarato come sarebbe in un'espressione normale.

Il seguente insieme di operatori con identica precedenza e associatività viene riutilizzato nei dichiaratori, vale a dire:

- l'operatore unario `*` "dereferenziazione" che denota un puntatore;
- l'operatore binario `[]` "subscription in array" che denota una matrice;
- l'operatore (1 + n) -ary `()` "chiamata di funzione" che denota una funzione;
- il `()` raggruppa le parentesi che sovrascrivono la precedenza e l'associatività del resto degli operatori elencati.

I tre operatori precedenti hanno la seguente precedenza e associatività:

Operatore	Precedenza relativa	Associatività
`[]` (abbonamento array)	1	Da sinistra a destra
`()` (chiamata di funzione)	1	Da sinistra a destra
`*` (dereferenziazione)	2	Da destra a sinistra

Quando si interpretano le dichiarazioni, si deve partire dall'identificatore verso l'esterno e applicare gli operatori adiacenti nell'ordine corretto come nella tabella precedente. Ogni applicazione di un operatore può essere sostituita con le seguenti parole inglesi:

Espressione	Interpretazione
`thing[X]`	una serie di dimensioni `X` di ...
`thing(t1, t2, t3)`	una funzione che prende `t1` , `t2` , `t3` e restituisce ...
`*thing`	un puntatore a ...

Ne consegue che l'inizio dell'interpretazione inglese inizierà sempre con l'identificatore e terminerà con il tipo che si trova sul lato sinistro della dichiarazione.

Esempi

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

`[]` ha la precedenza su `*` , quindi l'interpretazione è: i `names` sono una matrice di dimensione 20 di un puntatore a `char` .

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

Nel caso in cui si utilizzino le parentesi per sovrascrivere la precedenza, viene applicato prima `*` : `place` è un puntatore a una matrice di dimensione 10 di `char` .

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

Non c'è alcuna precedenza da preoccupare qui: `fn` è una funzione che richiede `long` , `short` e ritorno `int` .

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

Il `fn ()` viene applicato per primo: `fn` è una funzione che prende il `void` e restituisce un puntatore a `int` .

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

Sovrascrivere la precedenza di `()` : `fp` è un puntatore a una funzione che prende il `void` e che restituisce `int` .

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

Gli array multidimensionali non sono un'eccezione alla regola; gli operatori `[]` sono applicati nell'ordine da sinistra a destra secondo l'associatività nella tabella: `arr` è una matrice di dimensione 5 di una matrice di dimensione 8 di `int` .

```
int **ptr;
```

I due operatori di dereferenziazione hanno la stessa precedenza, quindi l'associatività ha effetto. Gli operatori sono applicati nell'ordine da destra a sinistra: `ptr` è un puntatore a un puntatore a un `int` .

Dichiarazioni multiple

La virgola può essere utilizzata come separatore (* non * che agisce come l'operatore virgola) per

delimitare più dichiarazioni all'interno di una singola istruzione. La seguente dichiarazione contiene cinque dichiarazioni:

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

Gli oggetti dichiarati nell'esempio precedente sono:

- `fn` : una funzione che si `void` e restituisce `int` ;
- `ptr` : un puntatore a un `int` ;
- `fp` : un puntatore a una funzione che assume `int` e restituisce `int` ;
- `arr` : una matrice di dimensione 10 di una matrice di dimensione 20 di `int` ;
- `num` : `int` .

Interpretazione alternativa

Poiché il mirror delle dichiarazioni viene utilizzato, una dichiarazione può anche essere interpretata in termini di operatori che potrebbero essere applicati sull'oggetto e il tipo risultante finale di tale espressione. Il tipo che si trova sul lato sinistro è il risultato finale che viene restituito dopo l'applicazione di tutti gli operatori.

```
/*
 * 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];
```

Leggi Tipi di dati online: https://riptutorial.com/it/c/topic/309/tipi-di-dati

Capitolo 59: Tipo qualificatori

Osservazioni

I qualificatori di tipo sono le parole chiave che descrivono una semantica aggiuntiva su un tipo. Sono parte integrante delle firme dei tipi. Possono apparire sia al livello più alto di una dichiarazione (che influisce direttamente sull'identificatore) sia a sottolivelli (rilevanti solo per i puntatori che influiscono sui valori puntati):

Parola chiave	Osservazioni
`const`	Impedisce la mutazione dell'oggetto dichiarato (apparendo al livello più alto) o impedisce la mutazione del valore puntato (apparendo accanto a un sottotipo di puntatore).
`volatile`	Informa il compilatore che l'oggetto dichiarato (al livello più alto) o il valore puntato (nei sottotipi del puntatore) può cambiare il suo valore come risultato di condizioni esterne, non solo come risultato del flusso di controllo del programma.
`restrict`	Un suggerimento per l'ottimizzazione, pertinente solo ai puntatori. Dichiara l'intento che per tutta la durata del puntatore, non verranno utilizzati altri puntatori per accedere allo stesso oggetto puntato.

L'ordinamento di tipo qualificatori rispetto a specificatori della classe di memorizzazione (`static` , `extern` , `auto` , `register`), tipo di modificatori (`signed` , `unsigned` , `short` , `long`) e specificatori di tipo (`int` , `char` , `double` , ecc) non è applicata, ma la buona pratica è metterli nell'ordine di cui sopra:

```
static const volatile unsigned long int a = 5; /* good practice */
unsigned volatile long static int const b = 5; /* bad practice */
```

Qualifiche di alto livello

```
/* "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;
```

Qualifiche del sottotipo di puntatore

```
/* "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

Variabili (const) non modificabili

```
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 */
```

La qualifica `const` significa solo che non abbiamo il diritto di modificare i dati. Ciò non significa che il valore non può cambiare alle nostre spalle.

```
_Bool doIt(double const* a) {
   double rememberA = *a;
   // do something long and complicated that calls other functions

   return rememberA == *a;
}
```

Durante l'esecuzione delle altre chiamate `*a` potrebbe essersi verificato un cambiamento, e quindi questa funzione potrebbe tornare `false` o `true` .

avvertimento

Le variabili con qualifica `const` possono essere modificate usando i puntatori:

```
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" */
```

Ma farlo è un errore che porta a comportamenti non definiti. La difficoltà qui è che questo può comportarsi come previsto in semplici esempi come questo, ma poi andare male quando il codice cresce.

Variabili volatili

La parola chiave `volatile` indica al compilatore che il valore della variabile può cambiare in qualsiasi momento come risultato di condizioni esterne, non solo come risultato del flusso di controllo del programma.

Il compilatore non ottimizzerà tutto ciò che ha a che fare con la variabile volatile.

```
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;
```

Ci sono due ragioni principali per utilizzare variabili volatili:

- Interfaccia con hardware con registri I / O mappati in memoria.
- Quando si utilizzano variabili che vengono modificate al di fuori del flusso di controllo del programma (ad esempio, in una routine di servizio di interrupt)

Vediamo questo esempio:

```
int quit = false;

void main()
{
    ...
    while (!quit) {
      // Do something that does not modify the quit variable
    }
    ...
}

void interrupt_handler(void)
{
  quit = true;
}
```

Il compilatore può notare che il ciclo while non modifica la variabile `quit` e converte il ciclo in un ciclo `while (true)` senza fine. Anche se la variabile `quit` è impostata sul gestore di segnale per `SIGINT` e `SIGTERM` , il compilatore non lo sa.

Dichiarare `quit` come `volatile` dirà al compilatore di non ottimizzare il ciclo e il problema sarà risolto.

Lo stesso problema si verifica quando si accede all'hardware, come vediamo in questo esempio:

```
uint8_t * pReg = (uint8_t *) 0x1717;

// Wait for register to become non-zero
while (*pReg == 0) { } // Do something else
```

Il comportamento dell'ottimizzatore è di leggere il valore della variabile una volta, non c'è bisogno

di rileggerlo, poiché il valore sarà sempre lo stesso. Quindi finiamo con un ciclo infinito. Per forzare il compilatore a fare ciò che vogliamo, modifichiamo la dichiarazione in:

```
uint8_t volatile * pReg = (uint8_t volatile *) 0x1717;
```

Leggi Tipo qualificatori online: https://riptutorial.com/it/c/topic/2588/tipo-qualificatori

Capitolo 60: typedef

introduzione

Il meccanismo `typedef` consente la creazione di alias per altri tipi. Non crea nuovi tipi. Le persone usano spesso `typedef` per migliorare la portabilità del codice, per fornire alias per i tipi di struttura o unione, o per creare alias per i tipi di funzione (o puntatore di funzione).

Nello standard C, `typedef` è classificato come "classe di memoria" per comodità; si verifica sintatticamente dove potrebbero apparire classi di memoria come `static` o `extern` .

Sintassi

- typedef nome_esistente nome_alias;

Osservazioni

Svantaggi di Typedef

`typedef` potrebbe portare all'inquinamento del namespace nei grandi programmi in C.

Svantaggi di Typedef Structs

Inoltre, le strutture `typedef` senza un nome di tag sono una delle cause principali di imposizione inutile delle relazioni di ordinamento tra i file di intestazione.

Tenere conto:

```
#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
```

Con tale definizione, non usando `typedefs` , è possibile che un'unità di compilazione includa `foo.h` per ottenere la definizione `FOO_DEF` . Se non tenta di dereferenziare il membro della `bar` della struttura `foo` non sarà necessario includere il file `bar.h`

Typedef vs #define

`#define`

è una direttiva per il pre-processore C che viene anche utilizzata per definire gli alias per vari tipi di dati simili a `typedef` ma con le seguenti differenze:

- `typedef` è limitato a dare nomi simbolici a tipi solo dove come `#define` può essere usato per definire alias anche per valori.
- `typedef` interpretazione `typedef` viene eseguita dal compilatore mentre le istruzioni `#define` vengono elaborate dal pre-processore.
- Si noti che `#define cptr char *` seguito da `cptr a, b;` non fa lo stesso di `typedef char *cptr;` seguito da `cptr a, b;` . Con il `#define` , `b` è un semplice `char` variabile, ma è anche un puntatore con il `typedef` .

Examples

Typedef per strutture e unioni

Puoi dare nomi alias a una `struct` :

```
typedef struct Person {
    char name[32];
    int age;
} Person;

Person person;
```

Rispetto al modo tradizionale di dichiarare le strutture, i programmatori non avrebbero bisogno di avere `struct` ogni volta che dichiarano un'istanza di quella struttura.

Si noti che il nome `Person` (al contrario di `struct Person`) non è definito fino al punto e virgola finale. Pertanto, per gli elenchi collegati e le strutture ad albero che devono contenere un puntatore allo stesso tipo di struttura, è necessario utilizzare:

```
typedef struct Person {
    char name[32];
    int age;
    struct Person *next;
} Person;
```

o:

```
typedef struct Person Person;

struct Person {
    char name[32];
    int age;
    Person *next;
};
```

L'uso di un `typedef` per un tipo di `union` è molto simile.

```
typedef union Float Float;

union Float
{
    float f;
    char  b[sizeof(float)];
};
```

Una struttura simile a questa può essere utilizzata per analizzare i byte che costituiscono un valore `float` .

Usi semplici di Typedef

Per dare nomi brevi a un tipo di dati

Invece di:

```
long long int foo;
struct mystructure object;
```

si può usare

```
/* write once */
typedef long long ll;
typedef struct mystructure mystruct;

/* use whenever needed */
ll foo;
mystruct object;
```

Ciò riduce la quantità di digitazione necessaria se il tipo viene utilizzato più volte nel programma.

Migliorare la portabilità

Gli attributi dei tipi di dati variano tra diverse architetture. Ad esempio, un `int` può essere un tipo a 2 byte in un'implementazione e un tipo a 4 byte in un altro. Supponiamo che un programma debba utilizzare un tipo di 4 byte per funzionare correttamente.

In un'implementazione, la dimensione di `int` sia 2 byte e quella di `long` 4 byte. In un altro, la dimensione di `int` sia 4 byte e quella di `long` 8 byte. Se il programma è scritto usando la seconda implementazione,

```
/* program expecting a 4 byte integer */
int foo; /* need to hold 4 bytes to work */
/* some code involving many more ints */
```

Affinché il programma venga eseguito nella prima implementazione, tutte le dichiarazioni `int` dovranno essere modificate a `long` .

```
/* program now needs long */
```

```
long foo; /*need to hold 4 bytes to work */
/* some code involving many more longs - lot to be changed */
```

Per evitare ciò, si può usare `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 */
```

Quindi, solo l'istruzione `typedef` dovrebbe essere cambiata ogni volta, invece di esaminare l'intero programma.

C99

L'intestazione `<stdint.h>` e l'intestazione `<inttypes.h>` correlata definiscono nomi di tipi standard (usando `typedef`) per numeri interi di varie dimensioni, e questi nomi sono spesso la scelta migliore nel codice moderno che ha bisogno di numeri interi fissi. Ad esempio, `uint8_t` è un tipo intero a 8 bit senza segno; `int64_t` è un tipo intero a 64 bit con `int64_t` . Il tipo `uintptr_t` è un tipo intero senza segno abbastanza grande da contenere qualsiasi puntatore all'oggetto. Questi tipi sono teoricamente facoltativi, ma è raro che non siano disponibili. Ci sono varianti come `uint_least16_t` (il più piccolo tipo di intero senza segno con almeno 16 bit) e `int_fast32_t` (il tipo di intero con `int_fast32_t` più veloce con almeno 32 bit). Inoltre, `intmax_t` e `uintmax_t` sono i tipi interi più grandi supportati dall'implementazione. Questi tipi sono obbligatori.

Per specificare un utilizzo o migliorare la leggibilità

Se un insieme di dati ha uno scopo particolare, si può usare `typedef` per dargli un nome significativo. Inoltre, se la proprietà dei dati cambia in modo tale che il tipo di base deve essere modificato, solo l'istruzione `typedef` deve essere modificata, invece di esaminare l'intero programma.

Typedef per puntatori di funzioni

Possiamo usare `typedef` per semplificare l'uso dei puntatori di funzione. Immagina di avere alcune funzioni, tutte con la stessa firma, che usano il loro argomento per stampare qualcosa in modi diversi:

```
#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);
}
```

Ora possiamo usare un `typedef` per creare un tipo di puntatore a funzione denominata chiamato stampante:

```
typedef void (*printer_t)(int);
```

Questo crea un tipo, denominato `printer_t` per un puntatore a una funzione che accetta un singolo argomento `int` e non restituisce nulla, che corrisponde alla firma delle funzioni che abbiamo sopra. Per usarlo creiamo una variabile del tipo creato e assegniamo un puntatore a una delle funzioni in questione:

```
printer_t p = &print_to_n;
void (*p)(int) = &print_to_n; // This would be required without the type
```

Quindi chiamare la funzione puntata dalla variabile del puntatore della funzione:

```
p(5);           // Prints 1 2 3 4 5 on separate lines
(*p)(5);        // So does this
```

Pertanto, `typedef` consente una sintassi più semplice quando si gestiscono i puntatori di funzione. Ciò diventa più evidente quando i puntatori di funzione vengono utilizzati in situazioni più complesse, come gli argomenti per le funzioni.

Se si utilizza una funzione che accetta un puntatore di funzione come parametro senza un tipo di puntatore di funzione definito, la definizione della funzione sarebbe,

```
void foo (void (*printer)(int), int y){
    //code
    printer(y);
    //code
}
```

Tuttavia, con `typedef` è:

```
void foo (printer_t printer, int y){
    //code
    printer(y);
    //code
}
```

Allo stesso modo, le funzioni possono restituire i puntatori di funzione e, di nuovo, l'uso di un `typedef` può semplificare la sintassi quando lo si fa.

Un classico esempio è la funzione di `signal` da `<signal.h>` . La dichiarazione per questo (dallo standard C) è:

```
void (*signal(int sig, void (*func)(int)))(int);
```

Questa è una funzione che accetta due argomenti: un `int` e un puntatore a una funzione che accetta un `int` come argomento e non restituisce nulla, e che restituisce un puntatore che

funziona come il suo secondo argomento.

Se abbiamo definito un tipo `SigCatcher` come alias per il puntatore al tipo di funzione:

```
typedef void (*SigCatcher)(int);
```

quindi potremmo dichiarare `signal()` usando:

```
SigCatcher signal(int sig, SigCatcher func);
```

Nel complesso, questo è più facile da capire (anche se lo standard C non ha scelto di definire un tipo per fare il lavoro). La funzione `signal` accetta due argomenti, un `int` e un `SigCatcher` , e restituisce un `SigCatcher` - dove un `SigCatcher` è un puntatore a una funzione che accetta un argomento `int` e non restituisce nulla.

Sebbene l'utilizzo di nomi `typedef` per i tipi di puntatore a funzioni semplifica la vita, può anche portare a confusione per gli altri che manterranno il codice in un secondo momento, quindi utilizzare con cautela e documentazione adeguata. Vedi anche Function Pointers .

Leggi typedef online: https://riptutorial.com/it/c/topic/2681/typedef

Capitolo 61: Valgrind

Sintassi

- valgrind *nome-programma argomenti facoltativi < test input*

Osservazioni

Valgrind è uno strumento di debug che può essere utilizzato per diagnosticare errori relativi alla gestione della memoria nei programmi C. Valgrind può essere utilizzato per rilevare errori come l'utilizzo di un puntatore non valido, inclusa la scrittura o la lettura oltre lo spazio allocato, oppure effettuare una chiamata non valida a `free()` . Può anche essere utilizzato per migliorare le applicazioni attraverso funzioni che gestiscono il profiling della memoria.

Per maggiori informazioni vedi http://valgrind.org .

Examples

Esecuzione di Valgrind

```
valgrind ./my-program arg1 arg2 < test-input
```

Questo eseguirà il tuo programma e produrrà un report di eventuali allocazioni e de-allocazioni che ha fatto. Ti avviserà anche di errori comuni come l'uso di memoria non inizializzata, i puntatori di dereferenziamento in posti strani, la cancellazione della fine dei blocchi allocati usando malloc, o il fallimento dei blocchi liberi.

Aggiungere bandiere

Puoi anche attivare altri test, come ad esempio:

```
valgrind -q --tool=memcheck --leak-check=yes ./my-program arg1 arg2 < test-input
```

Vedi valgrind --help per maggiori informazioni sulle (molte) opzioni, o guarda la documentazione su http://valgrind.org/ per informazioni dettagliate su cosa significa l'output.

Byte persi - Dimenticare di liberare

Ecco un programma che chiama malloc ma non è gratuito:

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

int main(int argc, char **argv)
{
```

```
    char *s;

    s = malloc(26); // the culprint

    return 0;
}
```

Senza argomenti aggiuntivi, valgrind non cercherà questo errore.

Ma se `--leak-check=yes` o `--tool=memcheck` , si lamenterà e mostrerà le linee responsabili di quelle perdite di memoria se il programma è stato compilato in modalità debug:

```
$ 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==
```

Se il programma non è compilato in modalità debug (ad esempio con il flag `-g` in GCC) ci mostrerà comunque dove si è verificata la perdita in termini della funzione rilevante, ma non delle righe.

Questo ci permette di tornare indietro e vedere quale blocco è stato allocato in quella linea e provare a tracciare in avanti per capire perché non è stato liberato.

Errori più comuni riscontrati durante l'utilizzo di Valgrind

Valgrind ti fornisce le *linee in cui si è verificato l'errore* alla fine di ogni riga nel formato `(file.c:line_no)` . Gli errori in valgrind sono riassunti nel modo seguente:

```
ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Gli errori più comuni includono:

1. **Errori di lettura / scrittura non validi**

```
==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
```

Questo accade quando il codice inizia ad accedere alla memoria che non appartiene al programma. La dimensione della memoria a cui si accede fornisce anche un'indicazione di quale variabile è stata utilizzata.

2. **Uso di variabili non inizializzate**

```
==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)
```

Secondo l'errore, alla riga 7 del `main` di `valg.c` , la chiamata a `printf()` passato una variabile non inizializzata a `printf` .

3. **Liberazione illegale della memoria**

```
==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)
```

Secondo valgrind, il codice liberava la memoria illegalmente (una seconda volta) alla *riga 10* di `valg.c` , mentre era già liberato alla *riga 9* , e il blocco stesso era allocata nella memoria alla *riga 7* .

Leggi Valgrind online: https://riptutorial.com/it/c/topic/2674/valgrind

Capitolo 62: vincoli

Osservazioni

I vincoli sono un termine usato in tutte le specifiche C esistenti (recentemente ISO-IEC 9899-2011). Sono una delle tre parti del linguaggio descritte nella clausola 6 dello standard (lungo la sintassi laterale e la semantica).

ISO-IEC 9899-2011 definisce un vincolo come:

> restrizione, sintattica o semantica, mediante la quale l'esposizione degli elementi linguistici deve essere interpretata

(Si noti inoltre, in termini di standard C, un "vincolo di runtime" non è un tipo di vincolo e ha regole estensivamente diverse.)

In altre parole, un vincolo descrive una regola della lingua che renderebbe illegale un programma altrimenti sintatticamente valido. Sotto questo aspetto i vincoli sono in qualche modo come un comportamento non definito, qualsiasi programma che non li segue non è definito in termini di linguaggio C.

I vincoli d'altra parte hanno una differenza molto significativa rispetto ai comportamenti non definiti. Vale a dire che è necessaria un'implementazione per fornire un messaggio diagnostico durante la fase di traduzione (parte della compilazione) se un vincolo viene violato, questo messaggio potrebbe essere un avvertimento o potrebbe interrompere la compilazione.

Examples

Duplicare nomi di variabili nello stesso ambito

Un esempio di un vincolo come espresso nello standard C sta avendo due variabili dello stesso nome dichiarate in uno scope [1)] , ad esempio:

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

Questo codice viola il vincolo e deve produrre un messaggio diagnostico in fase di compilazione. Questo è molto utile rispetto al comportamento non definito, in quanto lo sviluppatore verrà informato del problema prima dell'esecuzione del programma, potenzialmente facendo qualsiasi cosa.

I vincoli tendono quindi ad essere errori facilmente rilevabili in fase di compilazione come questo, problemi che risultano in un comportamento indefinito ma che sarebbero difficili o impossibili da rilevare in fase di compilazione non sono quindi vincoli.

1) formulazione esatta:

C99

Se un identificatore non ha alcun collegamento, non deve esserci più di una dichiarazione dell'identificatore (in un dichiaratore o specificatore del tipo) con lo stesso ambito e nello stesso spazio dei nomi, ad eccezione dei tag specificati in 6.7.2.3.

Operatori aritmetici unari

Gli operatori unari `+` e `-` sono utilizzabili solo su tipi aritmetici, quindi se per esempio si tenta di usarli su una struct il programma produrrà una diagnostica ad esempio:

```
struct foo
{
    bool bar;
};

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

Leggi vincoli online: https://riptutorial.com/it/c/topic/7397/vincoli

Capitolo 63: X-macro

introduzione

Le macro X sono una tecnica basata sul preprocessore per ridurre al minimo il codice ripetitivo e mantenere le corrispondenze di dati / codice. Più espansioni di macro distinte basate su un insieme comune di dati sono supportate rappresentando l'intero gruppo di espansioni tramite una singola macro principale, con il testo di sostituzione di quella macro costituito da una sequenza di espansioni di una macro interna, una per ciascun dato. La macro interna è tradizionalmente chiamata `X()` , da cui il nome della tecnica.

Osservazioni

Si prevede che l'utente di una macro master stile X-macro fornisca la propria definizione per la macro `X()` interna e, nell'ambito della sua estensione, per espandere la macro master. I riferimenti macro interni del master vengono quindi espansi in base alla definizione dell'utente di `X()` . In questo modo, la quantità di codice boilerplate ripetitivo nel file sorgente può essere ridotta (appare solo una volta, nel testo sostitutivo di `X()`), come favorito dagli aderenti alla filosofia "Do not Repeat Yourself" (DRY).

Inoltre, ridefinendo `X()` ed espandendo la macro master una o più volte aggiuntive, le macro X possono facilitare il mantenimento di dati e codice corrispondenti: una espansione della macro dichiara i dati (come elementi di array o membri di enum, ad esempio), e le altre espansioni producono codice corrispondente.

Sebbene il nome "X-macro" derivi dal nome tradizionale della macro interna, la tecnica non dipende da quel particolare nome. Qualsiasi nome di macro valido può essere utilizzato al suo posto.

Le critiche includono

- i file sorgente che si basano su macro X sono più difficili da leggere;
- come tutte le macro, le macro X sono rigorosamente testuali - non forniscono intrinsecamente alcun tipo di sicurezza; e
- Le macro X forniscono la *generazione del codice* . Rispetto alle alternative basate sulle funzioni di chiamata, le macro X rendono efficace il codice più grande.

Una buona spiegazione dei macro X può essere trovata nell'articolo [X-Macros] di Randy Meyers nel Dr. Dobbs (http://www.drdobbs.com/the-new-cx-macros/184401387) .

Examples

Uso banale di X-macros per 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 questo esempio il preprocessore genererà il seguente codice:

```
printf("X(%d) made this print\n", 1);
printf("X(%d) made this print\n", 2);
printf("X(%d) made this print\n", 3);
```

Valore enumerativo e identificativo

```
/* 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;
}
```

Successivamente puoi usare il valore enumerato nel tuo codice e stampare facilmente il suo identificatore usando:

```
printf("%s\n", enum2string(MyEnum_item2));
```

Estensione: assegna la macro X come argomento

L'approccio X-macro può essere generalizzato un po 'rendendo il nome della macro "X" un argomento della macro master. Questo ha il vantaggio di aiutare a evitare le collisioni dei nomi di macro e di consentire l'uso di una macro generica come la macro "X".

Come sempre con le macro X, la macro master rappresenta un elenco di elementi il cui significato è specifico per quella macro. In questa variante, una tale macro potrebbe essere definita in questo modo:

```
/* declare list of items */
#define ITEM_LIST(X) \
      X(item1) \
      X(item2) \
      X(item3) \
/* end of list */
```

Si potrebbe quindi generare codice per stampare i nomi degli oggetti in questo modo:

```
/* define macro to apply */
#define PRINTSTRING(value) printf( #value "\n");

/* apply macro to the list of items */
ITEM_LIST(PRINTSTRING)
```

Questo si espande a questo codice:

```
printf( "item1" "\n"); printf( "item2" "\n"); printf( "item3" "\n");
```

In contrasto con le macro X standard, dove il nome "X" è una caratteristica incorporata della macro master, con questo stile potrebbe non essere necessario o addirittura non desiderabile dopo aver indefinito la macro utilizzata come argomento (`PRINTSTRING` in questo esempio).

Generazione del codice

X-Macros può essere utilizzato per la generazione di codice, scrivendo codice ripetitivo: iterare su un elenco per eseguire alcune attività o per dichiarare un insieme di costanti, oggetti o funzioni.

Qui usiamo X-macros per dichiarare un enum contenente 4 comandi e una mappa dei loro nomi come stringhe

Quindi possiamo stampare i valori stringa dell'enum.

```
/* 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]());
```

Allo stesso modo, possiamo generare una tabella di salto per chiamare le funzioni in base al valore enum.

Ciò richiede che tutte le funzioni abbiano la stessa firma. Se non accettano argomenti e restituiscono un int, lo inseriremo in un'intestazione con la definizione enum:

```
/* 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[];
```

Tutti i seguenti possono essere in diverse unità di compilazione assumendo che la parte sopra sia inclusa come intestazione:

```
/* 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 */}
```

Un esempio di questa tecnica utilizzata nel codice reale è l' invio di comandi GPU in Chromium .

Leggi X-macro online: https://riptutorial.com/it/c/topic/628/x-macro

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con 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	- classificazione e conversione dei caratteri	Alejandro Caro, Jonathan Leffler, Roland Illig, Toby
3	Aliasing e tipo effettivo	2501, 4386427, Jens Gustedt
4	Ambito dell'identificatore	embedded_guy, Firas Moalla, Jean-Baptiste Yunès, Jens Gustedt, Jonathan Leffler
5	Argomenti della riga di comando	4386427, A B, alk, drov, dvhh, Jonathan Leffler, Malcolm McLean, Shog9, syb0rg, Toby, Woodrow Barlow, Yotam Salmon
6	Argomenti variabili	2501, Blacksilver, eush77, Jean-Baptiste Yunès, Jonathan Leffler, Leandros, mirabilos, syb0rg, Toby
7	Array	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
8	Assemblaggio in linea	bevenson, EsmaeelE, Jonathan Leffler
9	Asserzione	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	Atomics	Jens Gustedt
11	booleano	alk, Bob__, Braden Best, Chrono Kitsune, dhein, Insane, Jens Gustedt, Magisch, Mateusz Piotrowski, Peter, Toby
12	Classi di archiviazione	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
13	Commenti	Ankush, Chandrahas Aroori, Jonathan Leffler, Toby
14	Compilazione	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
15	Comportamento definito dall'implementazione	Jens Gustedt, John Bollinger, P.P.
16	Comportamento non definito	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
17	Conversioni implicite ed esplicite	alk, Firas Moalla, Jens Gustedt, Jeremy Thien, kdopen, Lundin, Toby
18	Crea e includi i file di intestazione	4444, Jonathan Leffler, patrick96, Sirsireesh Kodali
19	Dichiarazione vs definizione	Ashish Ahuja, foxtrot9, Kerrek SB, Toby
20	dichiarazioni	alk, AnArrayOfFunctions, Blacksilver, Firas Moalla, J Wu, Jens Gustedt, Jonathan Leffler, Jonathon Reinhart
21	Dichiarazioni di selezione	alk, bevenson, Blagovest Buyukliev, Faisal Mudhir, GoodDeeds, gsamaras, jxh, L.V.Rao, lordjohncena, MikeCAT, NeoR, noamgot, OznOg, P.P., Toby, tofro

22	Discussioni (native)	alk, Jens Gustedt, P.P.
23	Effetti collaterali	EsmaeelE, Jonathan Leffler, L.V.Rao, madD7, RamenChef, Sirsireesh Kodali, Toby
24	enumerazioni	Alejandro Caro, alk, jasoninnn, Jens Gustedt, Jonathan Leffler, OznOg, Toby
25	File e flussi I / O	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
26	Generazione di numeri casuali	dylanweber, ganchito55, haccks, hexwab, Jonathan Leffler, Leandros, Malcolm McLean, MikeCAT, Toby
27	Gestione degli errori	Jens Gustedt, stackptr
28	Gestione del segnale	3442, alk, Dariusz, Jens Gustedt, Leandros, mirabilos
29	Gestione della memoria	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
30	I bit-field	alk, EvilTeach, Fantastic Mr Fox, haccks, Ishay Peled, Jens Gustedt, John Odom, Jonathan Leffler, Lundin, madD7, Paul Hutchinson, RamenChef, Rishikesh Raje, Toby, vkgade
31	Idiomi di programmazione C comuni e pratiche di sviluppo	Chandrahas Aroori, Jonathan Leffler, Nityesh Agarwal, Shubham Agrawal
32	Imbottitura e imballaggio della struttura	EsmaeelE, Jarrod Dixon, Jedi, Jesferman, Jonathan Leffler, Liju Thomas, MayeulC, tilz0R
33	Inizializzazione	Jonathan Leffler, Liju Thomas, P.P.
34	inlining	Alex, EsmaeelE, Jens Gustedt, Jonathan Leffler, Toby
35	Input / Output formattato	alk, fluter, Jonathan Leffler, Jossi, lardenn, MikeCAT, polarysekt, StardustGogeta

36	Insidie comuni	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
37	Interprocess Communication (IPC)	cLDSEED, EsmaeelE, Jonathan Leffler, Toby
38	Iterazioni / loop di iterazione: per, while, do-while	alk, GoodDeeds, Jens Gustedt, jxh, L.V.Rao, Malcolm McLean, Nagaraj, RamenChef, reshad, Toby
39	Jump Statements	alk, Jens Gustedt, Jonathan Leffler, lordjohncena, Malcolm McLean, Sourav Ghosh, syb0rg, Toby
40	Letterali composti	alk, haccks, Jens Gustedt, Kerrek SB
41	Letterali per numeri, caratteri e archi	Jens Gustedt, Jonathan Leffler, Klas Lindbäck, Neui, Paul92, Toby
42	Liste collegate	4386427, alk, Andrea Biondo, bevenson, iRove, Jonathan Leffler, Jossi, Leandros, Mateusz Piotrowski, Ryan, Toby
43	Matematica standard	Alejandro Caro, alk, Blagovest Buyukliev, immerhart, Jonathan Leffler, manav m-n, Toby
44	multithreading	Parham Alvani, Toby
45	operatori	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
46	Parametri di funzione	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
47	Passa le matrici 2D alle funzioni	deamentiaemundi, Malcolm McLean, Shrinivas Patgar, Toby
48	Preprocessore e	Alex Garcia, alk, bevenson, bwoebi, Dariusz, DrPrItay, Erlend

	macro	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
49	puntatori	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
50	Puntatori di funzione	Alejandro Caro, alk, David Refaeli, Filip Allberg, hlovdal, John Burger, Leandros, Malcolm McLean, P.P., Srikar, stackptr, Toby
51	Punti di sequenza	2501, Armali, bta, Community, haccks, Jens Gustedt, John Bode , Toby
52	Selezione generica	2501, Jens Gustedt, Sun Qingyao
53	Sequenza di caratteri multi-carattere	Jonathan Leffler, PassionInfinite, Toby
54	sindacati	Jossi, RamenChef, Toby, Vality
55	stringhe	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
56	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
57	Strutture di prova	Community, EsmaeelE, Jonathan Leffler, lordjohncena, Toby, user2314737, vuko_zrno
58	Tipi di dati	2501, alk, Blagovest Buyukliev, Firas Moalla, Jens Gustedt, Keith Thompson, Ken Y-N, Leandros, P.P., Peter, WMios
59	Tipo qualificatori	alk, Blagovest Buyukliev, Jens Gustedt, Jesferman, madD7, tversteeg
60	typedef	Buser, Chandrahas Aroori, GoodDeeds, Jonathan Leffler, mame98, PhotometricStereo, Stephen Leppik, Toby
61	Valgrind	abacles, alk, Ankush, Chandrahas Aroori, Devansh Tandon, drov, Firas Moalla, J F, Jonathan Leffler, vasili111
62	vincoli	Armali, Toby, Vality
63	X-macro	Cimbali, Jens Gustedt, John Bollinger, Leandros, MD XF, mpromonet, poolie, RamenChef, technosaurus, templatetypedef, Toby