eBook Gratuit

APPRENEZ

D Language

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#d

Table des matières

À propos

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

It is an unofficial and free D 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 D 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

Chapitre 1: Démarrer avec le langage D

Remarques

D est un langage de programmation système avec une syntaxe de type C et un typage statique. Il combine l'efficacité, le contrôle et la puissance de modélisation avec la sécurité et la productivité du programmeur.

Versions

Version	Changelog	Date de sortie
ré	http://www.digitalmars.com/d/1.0/changelog.html	2007-01-23
D2	https://dlang.org/changelog/2.000.html	2007-06-17

Examples

Installation ou configuration

Le compilateur standard DMD du langage de programmation D peut s'exécuter sur toutes les principales plates-formes. Pour installer DMD, cliquez ici . Pour installer en ligne de commande, vous pouvez exécuter la commande (trouvée sur le site Web D):

```
curl -fsS https://dlang.org/install.sh | bash -s dmd
```

Gestionnaires de paquets

Arch Linux

```
pacman -S dlang
```

Chocolaté

```
choco install dmd
```

Gentoo

```
layman -f -a dlang
```

OSX Homebrew

```
brew install dmd
```

Debian / Ubuntu

L'installation sur les distributions Debian / Ubuntu nécessite que le référentiel APT soit ajouté à la liste des sources.

```
wget http://master.dl.sourceforge.net/project/d-apt/files/d-apt.list -O
/etc/apt/sources.list.d/d-apt.list
wget -qO - https://dlang.org/d-keyring.gpg | sudo apt-key add -
apt-get update
apt-get install dmd-bin
```

Autres compilateurs

LDC est un compilateur D qui utilise le frontend du compilateur officiel DMD et LLVM comme backend.

GDC est un compilateur D qui utilise le backend GCC pour générer du code.

IDE

Pour vous faciliter la vie, vous pouvez également installer un environnement de développement intégré (IDE). Le wiki D-Language a une liste des IDE et plug-ins disponibles pour toutes les plates-formes ici .

Bonjour le monde

```
import std.stdio;

// Let's get going!
void main()
{
    writeln("Hello World!");
}
```

Pour compiler et exécuter, enregistrez ce texte dans un fichier appelé `main.d` À partir de la ligne de commande, exécutez `dmd main.d` pour compiler le programme. Enfin, lancez `./main` pour exécuter le programme dans un shell bash ou cliquez sur l'exécutable sous Windows.

Bonjour le monde!

Pour créer le programme d'impression classique "Hello, world", créez un fichier appelé `hello.d`

avec un éditeur de texte contenant le code suivant:

```
import std.stdio;

void main() {
    writeln("Hello, World!");    //writeln() automatically adds a newline (\n) to the output
}
```

Explication:

```
import std.stdio
```

Cette ligne indique au compilateur que les fonctions définies dans le module standard de bibliothèque `std.stdio` seront utilisées. Tout module peut être importé, à condition que le compilateur sache où les chercher. De nombreuses fonctions sont fournies dans la bibliothèque standard massive de D.

```
void main() {
```

Cette ligne déclare la fonction `main` , retournant `void` . Notez que contrairement à C et C ++, D permet à main d'être de type `void` . La fonction `main` est spéciale car c'est le point d'entrée du programme, c'est-à-dire que commence l'exécution du programme. Quelques notes sur les fonctions en général:

- Le nom d'une fonction peut être tout ce qui commence par une lettre et se compose de lettres, de chiffres et de traits de soulignement.
- Les paramètres attendus seront une liste de noms de variables séparés par des virgules et leurs types de données.
- La valeur que la fonction est censée renvoyer peut être un type de données existant et doit correspondre au type d'expression utilisé dans l'instruction return au sein de la fonction.

Les accolades `{ … }` sont utilisées par paires pour indiquer où commence et se termine un bloc de code. Ils peuvent être utilisés de nombreuses manières, mais dans ce cas ils indiquent où la fonction commence et se termine.

```
writeln("Hello, World!");
```

`writeln` est une fonction déclarée dans `std.stdio` qui écrit ses outils sur `stdout` . Dans ce cas, son argument est `"Hello, World"` , qui sera écrit sur la console. Différents caractères de format, similaires à ceux utilisés par C `printf` peuvent être utilisés, comme `\n` , `\r` , etc.

Chaque déclaration doit être terminée par un point-virgule.

Les commentaires sont utilisés pour indiquer quelque chose à la personne qui lit le code et sont traités comme un blanc par le compilateur. Dans le code ci-dessus, ceci est un commentaire:

```
//writeln() automatically adds a newline (\n) to the output
```

Ce sont des morceaux de code qui sont ignorés par le compilateur. Il y a trois manières différentes de commenter en D:

1. `//` - Commentez tout le texte sur la même ligne, après le `//`
2. `/* comment text */` - Ce sont utiles pour les commentaires multilignes
3. `/+ comment text +` - Ce sont aussi des commentaires multilignes

Ils sont très utiles pour transmettre ce qu'une fonction / un morceau de code fait à un autre développeur.

Compiler et exécuter le programme

Pour exécuter ce programme, le code doit être compilé en un exécutable. Cela peut être fait avec l'aide du compilateur.

Pour compiler à l'aide de DMD, le compilateur D de référence, ouvrez un terminal, naviguez jusqu'à l'emplacement du fichier `hello.d` que vous avez créé, puis exécutez:

```
dmd hello.d
```

Si aucune erreur n'est trouvée, le compilateur générera un exécutable nommé d'après votre fichier source. Cela peut maintenant être exécuté en tapant

```
./hello
```

Lors de l'exécution, le programme imprimera `Hello, World!` , suivi d'une nouvelle ligne.

Lire les valeurs d'une chaîne

```
import std.format;

void main() {
    string s = "Name Surname 18";
    string name, surname;
    int age;
    formattedRead(s, "%s %s %s", &name, &surname, &age);
    // %s selects a format based on the corresponding argument's type
}
```

La documentation officielle pour les chaînes de format peut être trouvée à: https://dlang.org/phobos/std_format.html#std.format

Lire Démarrer avec le langage D en ligne: https://riptutorial.com/fr/d/topic/1036/demarrer-avec-le-langage-d

Chapitre 2: Boucles

Syntaxe

- for (<initializer>; <condition de boucle>; <instruction de boucle>) {<instructions>}
- while (<condition>) {<instructions>}
- faire {<instructions>} while (<condition>);
- foreach (<el>, <collection>)
- foreach_reverse (<el>, <collection>)

Remarques

- `for` boucle dans la *programmation en D* , spécification
- `while` boucle en *programmation en D* , spécification
- `do while` boucle dans la *programmation en D* , spécification
- `foreach` dans la *programmation en D* , opApply , spécification

Vous pouvez jouer avec des boucles et foreach en ligne.

Examples

Pour la boucle

```
void main()
{
    import std.stdio : writeln;
    int[] arr = [1, 3, 4];
    for (int i = 0; i < arr.length; i++)
    {
        arr[i] *= 2;
    }
    writeln(arr); // [2, 6, 8]
}
```

En boucle

```
void main()
{
    import std.stdio : writeln;
    int[] arr = [1, 3, 4];
    int i = 0;
    while (i < arr.length)
    {
        arr[i++] *= 2;
    }
    writeln(arr); // [2, 6, 8]
}
```

faire pendant

```
void main()
{
    import std.stdio : writeln;
    int[] arr = [1, 3, 4];
    int i = 0;
    assert(arr.length > 0, "Array must contain at least one element");
    do
    {
        arr[i++] *= 2;
    } while (i < arr.length);
    writeln(arr); // [2, 6, 8]
}
```

Pour chaque

Foreach permet une manière moins risquée et plus lisible d'itérer les collections. L'attribut `ref` peut être utilisé si l'on veut modifier directement l'élément itéré.

```
void main()
{
    import std.stdio : writeln;
    int[] arr = [1, 3, 4];
    foreach (ref el; arr)
    {
        el *= 2;
    }
    writeln(arr); // [2, 6, 8]
}
```

L'index de l'itération est également accessible:

```
void main()
{
    import std.stdio : writeln;
    int[] arr = [1, 3, 4];
    foreach (i, el; arr)
    {
        arr[i] = el * 2;
    }
    writeln(arr); // [2, 6, 8]
}
```

L'itération dans l'ordre inverse est également possible:

```
void main()
{
    import std.stdio : writeln;
    int[] arr = [1, 3, 4];
    int i = 0;
    foreach_reverse (ref el; arr)
    {
        el += i++; // 4 is incremented by 0, 3 by 1, and 1 by 2
    }
```

```
    writeln(arr); // [3, 4, 4]
}
```

Pause, continuer et étiquettes

```
void main()
{
    import std.stdio : writeln;
    int[] arr = [1, 3, 4, 5];
    foreach (i, el; arr)
    {
        if (i == 0)
            continue; // continue with the next iteration
        arr[i] *= 2;
        if (i == 2)
            break; // stop the loop iteration
    }
    writeln(arr); // [1, 6, 8, 5]
}
```

Les étiquettes peuvent également être utilisées pour `break` ou `continue` dans les boucles imbriquées.

```
void main()
{
    import std.stdio : writeln;
    int[] arr = [1, 3, 4];
    outer: foreach (j; 0..10) // iterates with j=0 and j=1
        foreach (i, el; arr)
        {
            arr[i] *= 2;
            if (j == 1)
                break outer; // stop the loop iteration
    }
    writeln(arr); // [4, 6, 8] (only 1 reaches the second iteration)
}
```

Lire Boucles en ligne: https://riptutorial.com/fr/d/topic/4696/boucles

Chapitre 3: Cordes

Remarques

- Les chaînes en D sont immuables; utiliser `.dup` pour faire un mutable `char` tableau si vous souhaitez modifier en place.

Examples

Inverser une chaîne

`string` est définie comme `alias string = immutable(char)[];` : il faut donc utiliser `dup` pour créer un tableau de caractères mutable avant de pouvoir l'inverser:

```
import std.stdio;
import std.string;

int main() {

    string x = "Hello world!";
    char[] x_rev = x.dup.reverse;

    writeln(x_rev); // !dlrow olleH

    return 0;

}
```

Test d'une chaîne vide ou nulle

Chaîne vide

La chaîne vide n'est pas nulle mais a une longueur nulle:

```
string emptyString = "";
// an empty string is not null...
assert(emptyString !is null);

// ... but it has zero lenght
assert(emptyString.length == 0);
```

Chaîne nulle

```
string nullString = null;
```

une chaîne vide est null (De Lapalisse)

```
assert(nullString is null);
```

mais, contrairement à C #, lire la longueur d'une chaîne vide ne génère pas d'erreur:

```
assert(nullString.length == 0);
assert(nullString.empty);
```

Test pour vide ou null

```
if (emptyOrNullString.length == 0) {
}

// or
if (emptyOrNullString.length) {
}

// or
import std.array;
if (emptyOrNullString.empty) {
}
```

Test pour null

```
if (nullString is null) {
}
```

Les références

- Quelle est la bonne façon de tester une chaîne vide?
- Est-ce que D a la chaîne de C #. Vide?

Convertir une chaîne en octets [] et vice versa

Chaîne à `ubyte[]` immuable `ubyte[]`

```
string s = "unogatto";
immutable(ubyte[]) ustr = cast(immutable(ubyte)[])s;

assert(typeof(ustr).stringof == "immutable(ubyte[])");
assert(ustr.length == 8);
assert(ustr[0] == 0x75); //u
assert(ustr[1] == 0x6e); //n
```

```
assert(ustr[2] == 0x6f); //o
assert(ustr[3] == 0x67); //g
assert(ustr[7] == 0x6f); //o
```

Chaîne de `ubyte[]`

```
string s = "unogatto";
ubyte[] mustr = cast(ubyte[])s;

assert(typeof(mustr).stringof == "ubyte[]");

assert(mustr.length == 8);
assert(mustr[0] == 0x75);
assert(mustr[1] == 0x6e);
assert(mustr[2] == 0x6f);
assert(mustr[3] == 0x67);
assert(mustr[7] == 0x6f);
```

`ubyte[]` pour enchaîner

```
ubyte[] stream = [ 0x75, 0x6e, 0x6f, 0x67];
string us  = cast(string)stream;
assert(us == "unog");
```

Les références

- Forum DLang

Lire Cordes en ligne: https://riptutorial.com/fr/d/topic/5760/cordes

Chapitre 4: Des classes

Syntaxe

- classe Foo {} // hérite de Object
- class Bar: Foo {} // La barre est aussi un Foo
- Foo f = new Foo (); // instancie de nouveaux objets sur le tas

Remarques

Voir la spécification , parcourir un chapitre de livre sur les classes , l' héritage et jouer de manière interactive .

Examples

Héritage

```
class Animal
{
    abstract int maxSize(); // must be implemented by sub-class
    final float maxSizeInMeters() // can't be overridden by base class
    {
        return maxSize() / 100.0;
    }
}

class Lion: Animal
{
    override int maxSize() { return 350; }
}

void main()
{
    import std.stdio : writeln;
    auto l = new Lion();
    assert(l.maxSizeInMeters() == 3.5);

    writeln(l.maxSizeInMeters()); // 3.5
}
```

Instanciation

```
class Lion
{
    private double weight; // only accessible with-in class

    this(double weight)
    {
        this.weight = weight;
    }
```

```
    double weightInPounds() const @property // const guarantees no modifications
    // @property functions are treated as fields
    {
        return weight * 2.204;
    }
}

void main()
{
    import std.stdio : writeln;
    auto l = new Lion(100);
    assert(l.weightInPounds == 220.4);

    writeln(l.weightInPounds); // 220.4
}
```

Lire Des classes en ligne: https://riptutorial.com/fr/d/topic/4695/des-classes

Chapitre 5: Désengagement

Syntaxe

- unittest {...} - un bloc qui n'est exécuté qu'en mode "unittesting"
- assert (<expression qui donne une valeur booléenne>, <message d'erreur facultatif>)

Examples

Blocs Unittest

Les tests sont un excellent moyen de garantir des applications stables et exemptes de bogues. Ils servent de documentation interactive et permettent de modifier le code sans crainte de casser la fonctionnalité. D fournit une syntaxe pratique et native pour les blocs les plus `unittest` dans le langage D. N'importe où dans un module D, les blocs les plus `unittest` utilisés peuvent être utilisés pour tester la fonctionnalité du code source.

```
/**
Yields the sign of a number.
Params:
    n = number which should be used to check the sign
Returns:
    1 for positive n, -1 for negative and 0 for 0.
*/
T sgn(T)(T n)
{
    if (n == 0)
        return 0;
    return (n > 0) ? 1 : -1;
}

// this block will only be executed with -unittest
// it will be removed from the executable otherwise
unittest
{
    // go ahead and make assumptions about your function
    assert(sgn(10)  ==  1);
    assert(sgn(1)   ==  1);
    assert(sgn(-1)  == -1);
    assert(sgn(-10) == -1);
}
```

Exécuter le plus souvent

Si l' `-unittest` flag est transmise au compilateur D, tous les blocs les plus communs seront exécutés. Il est souvent utile de laisser le compilateur générer une fonction `main` tronquée. En utilisant le wrapper de compilation et d'exécution `rdmd` , tester votre programme D devient aussi simple que:

```
rdmd -main -unittest yourcode.d
```

Bien sûr, vous pouvez également diviser ce processus en deux étapes si vous le souhaitez:

```
dmd -main -unittest yourcode.d
./yourcode
```

Pour les projets de `dub` , la compilation de tous les fichiers et l'exécution de leurs blocs les plus faciles à utiliser peuvent être

```
dub test
```

Conseil: définissez `tdmd` comme alias de shell pour éviter les basculements.

```
alias tdmd="rdmd -main -unittest"
```

puis testez vos fichiers avec:

```
tdmd yourcode.d
```

Annoté le plus souvent

Pour le code `@nogc` modèles, il est souvent utile de vérifier cela pour les attributs de fonction (par exemple, `@nogc` est déduit correctement. Pour garantir cela pour un test spécifique et donc pour taper tout le code unittest, il est possible de l'annoter)

```
@safe @nogc pure nothrow unittest
{
    import std.math;
    assert(exp(0)  ==  1);
    assert(log(1)  ==  0);
}
```

Notez bien sûr que dans D chaque bloc peut être annoté avec des attributs et que les compilateurs, bien sûr, vérifient qu'ils sont corrects. Ainsi, par exemple, ce qui suit serait similaire à l'exemple ci-dessus:

```
unittest
{
    import std.math;
    @safe {
        assert(exp(0)  ==  1);
        assert(log(1)  ==  0);
    }
}
```

Lire Désengagement en ligne: https://riptutorial.com/fr/d/topic/6201/desengagement

Chapitre 6: Évaluation de la fonction de temps de compilation (CTFE)

Remarques

CTFE est un mécanisme qui permet au compilateur d'exécuter des fonctions au moment de la compilation. Il n'y a pas de jeu spécial du langage D nécessaire pour utiliser cette fonctionnalité - chaque fois qu'une fonction dépend des valeurs connues de la compilation, le compilateur D peut décider de l'interpréter pendant la compilation.

Vous pouvez également jouer de manière interactive avec CTFE.

Examples

Evaluer une fonction à la compilation

```
long fib(long n)
{
    return n < 2 ? n : fib(n - 1) + fib(n - 2);
}

struct FibStruct(int n) { // Remarks: n is a template
    ubyte[fib(n)] data;
}

void main()
{
    import std.stdio : writeln;
    enum f10 = fib(10); // execute the function at compile-time
    pragma(msg, f10); // will print 55 during compile-time
    writeln(f10); // print 55 during runtime
    pragma(msg, FibStruct!11.sizeof); // The size of the struct is 89
}
```

Lire Évaluation de la fonction de temps de compilation (CTFE) en ligne: https://riptutorial.com/fr/d/topic/4694/evaluation-de-la-fonction-de-temps-de-compilation--ctfe-

Chapitre 7: Gammes

Remarques

Si un compilateur rencontre un foreach

```
foreach (element; range) {
```

il est réécrit en interne comme suit:

```
for (auto it = range; !it.empty; it.popFront()) {
    auto element = it.front;
    ...
}
```

Tout objet remplissant l'interface ci-dessus est appelé une plage d'entrée et est donc un type pouvant être itéré sur:

```
struct InputRange {
    @property bool empty();
    @property T front();
    void popFront();
}
```

Examples

Les chaînes et les tableaux sont des plages

```
import std.stdio;

void main() {
    auto s = "hello world";
    auto a = [1, 2, 3, 4];

    foreach (c; s) {
        write(c, "!"); // h!e!l!l!o! !w!o!r!l!d!
    }
    writeln();

    foreach (x; a) {
        write(x * x, ", "); // 1, 4, 9, 16,
    }
}
```

Créer un nouveau type de plage d'entrée

Le concept `InputRange` a trois fonctions, par exemple:

```
struct InputRange(T) {
```

```
    @property bool empty();
    @property T front();
    void popFront();
}
```

En bref, un moyen de

1. vérifier si la plage est vide
2. obtenir l'élément actuel
3. passer à l'élément suivant

Pour faire de notre type un `InputRange` , nous devons implémenter ces trois fonctions. Regardons la suite infinie de carrés.

```
struct SquaresRange {
    int cur = 1;

    @property bool empty() {
        return false;
    }

    @property int front() {
        return cur^^2;
    }

    void popFront() {
        cur++;
    }
}
```

Voir le D tour pour un exemple avec Fibonacci.

Lire Gammes en ligne: https://riptutorial.com/fr/d/topic/3106/gammes

Chapitre 8: Gardes de portée

Syntaxe

- scope (exit) - les instructions sont appelées quelle que soit la sortie du bloc actuel
- scope (success) - les instructions sont appelées lorsque le bloc en cours a été quitté normalement
- scope (failure) - les instructions sont appelées lorsque le bloc en cours a été quitté via le lancement d'exceptions

Remarques

L'utilisation de gardes de portée rend le code beaucoup plus propre et permet de placer l'allocation de ressources et de nettoyer le code les uns à côté des autres. Ces petits outils améliorent également la sécurité car ils garantissent que certains codes de nettoyage sont toujours appelés indépendamment des chemins réellement empruntés à l'exécution.

La fonction d'étendue D remplace efficacement l'idiome RAII utilisé en C ++, ce qui conduit souvent à des objets de surveillance de portée spéciale pour des ressources spéciales.

Les gardes de portée sont appelés dans l'ordre inverse de leur définition.

Jouez avec les gardes de la portée ou consultez un didacticiel complet .

Examples

Placez les codes d'allocation et de nettoyage les uns à côté des autres

Les gardes de portée permettent d'exécuter des instructions à certaines conditions si le bloc actuel est laissé.

```
import core.stdc.stdlib;

void main() {
    int* p = cast(int*)malloc(int.sizeof);
    scope(exit) free(p);
}
```

Plusieurs portées imbriquées

```
import std.stdio;

void main() {
    writeln("<html>");
    scope(exit) writeln("</html>");
    {
        writeln("\t<head>");
```

```
        scope(exit) writeln("\t</head>");
        "\t\t<title>%s</title>".writefln("Hello");
    } // the scope(exit) on the previous line is executed here

    writeln("\t<body>");
    scope(exit) writeln("\t</body>");

    writeln("\t\t<h1>Hello World!</h1>");
}
```

estampes

```
<html>
    <head>
        <title>Hello</title>
    </head>
    <body>
        <h1>Hello World!</h1>
    </body>
</html>
```

Lire Gardes de portée en ligne: https://riptutorial.com/fr/d/topic/4343/gardes-de-portee

Chapitre 9: Importations et modules

Syntaxe

- module mon.package;
- importer mon.package;
- import my.package: function;
- import fancyName = mypackage;
- import my.package: fancyFunctionName = function;

Remarques

Les modules fournissent automatiquement une étendue d'espace de noms pour leur contenu. Les modules ressemblent superficiellement aux classes, mais diffèrent par le fait que:

- Il n'y a qu'une seule instance de chaque module, et celle-ci est allouée de manière statique.
- Il n'y a pas de table virtuelle.
- Les modules n'héritent pas, ils n'ont pas de super modules, etc.
- Un seul module par fichier.
- Les symboles de module peuvent être importés.
- Les modules sont toujours compilés au niveau global et ne sont pas affectés par les attributs environnants ou d'autres modificateurs.
- Les modules peuvent être regroupés dans des hiérarchies appelées packages.

Les modules offrent plusieurs garanties:

- L'ordre dans lequel les modules sont importés n'affecte pas la sémantique.
- La sémantique d'un module n'est pas affectée par ce qui l'importe.
- Si un module C importe les modules A et B, toute modification apportée à B ne modifiera pas silencieusement le code en C qui dépend de A.

Examples

Importations mondiales

```
import std.stdio;
void main()
{
    writeln("Hello World!");
}
```

Plusieurs importations peuvent être spécifiées dans la même ligne, séparées par une `comma` ou dans une nouvelle ligne.

```
import std.stdio, std.math;
```

```
import std.datetime;
void main()
{
    writeln("2^4: ", pow(2, 4));
    writeln("Current time: ", Clock.currTime());
}
```

Importations sélectives

Les importations sélectives peuvent aider à nettoyer l'espace de nommage et à accélérer la compilation, car le compilateur n'a qu'à analyser les fonctions spécifiques sélectionnées.

```
import std.stdio: writeln;
void main()
{
    writeln("Hello world");
}
```

Importations locales

Vous pouvez également importer des symboles dans n'importe quelle portée, l'importation ne sera recherchée que lorsque la portée est requise (c.-à-d. Compilée) et les noms importés ne seront exposés que dans la portée importée. Le plus souvent, les possibilités d'importation locale sont les fonctions, les structures et les classes.

```
void main()
{
    import std.stdio: writeln;
    writeln("Hello world");
}
// writeln isn't defined here
```

Importations publiques

Les modules peuvent être exposés à d'autres modules avec `public imports` .

```
public import std.math;
// only exports the symbol 'pow'
public import std.math : pow;
```

Importations renommées

Un nom local pour une importation peut être donné, par lequel toutes les références aux symboles du module doivent être qualifiées avec:

```
import io = std.stdio;
void main()
{
    io.writeln("Hello world");
    std.stdio.writeln("hello!"); // error, std is undefined
    writeln("hello!");           // error, writeln is undefined
```

```
}
```

Les importations renommées sont pratiques lorsque vous utilisez des noms d'importation très longs.

Importations renommées et sélectives

Les importations sélectives peuvent également être renommées.

```
void main()
{
    import std.stdio : fooln = writeln;
    fooln("Hello world");
}
```

Déclaration de module

Les modules ont une correspondance univoque avec les fichiers sources. Le nom du module est, par défaut, le nom du fichier avec le chemin d'accès et l'extension supprimés, et peut être défini explicitement avec la déclaration du module. Le `ModuleDeclaration` définit le nom du module et le package `ModuleDeclaration` il appartient. S'il est absent, le nom du module correspond au même nom (sans chemin ni extension) du nom du fichier source.

```
module my.fancy.module;
```

Lire Importations et modules en ligne: https://riptutorial.com/fr/d/topic/4344/importations-et-modules

Chapitre 10: Les contrats

Remarques

Les assertions seront optimisées dans une version finale.

Examples

Contrats de fonction

Les contrats de fonction permettent au programmeur de vérifier les incohérences. Les incohérences incluent des paramètres non valides, vérifient la valeur de retour correcte ou un état non valide de l'objet.

Les vérifications peuvent avoir lieu avant et après l'exécution du corps de la fonction ou de la méthode.

```
void printNotGreaterThan42(uint number)
in {
    assert(number < 42);
}
body {
    import std.stdio : writeln;
    writeln(number);
}
```

Les assertions seront optimisées dans une version finale.

Contrats de fonction

Par exemple, si une méthode est appelée, l'état de l'objet peut ne pas permettre qu'une méthode soit appelée avec des paramètres spécifiques ou pas du tout.

```
class OlderThanEighteen {
    uint age;

    final void driveCar()
    in {
         assert(age >= 18); // variable must be in range
    }
    body {
         // step on the gas
    }
}
```

Lire Les contrats en ligne: https://riptutorial.com/fr/d/topic/6830/les-contrats

Chapitre 11: Mémoire et pointeurs

Syntaxe

- & <variable> - accès par référence (= récupère le pointeur sur les données de la variable)
- * <variable> - opérateur de déférence (= obtient l'objet de données à partir d'un pointeur)
- <type> * - type de données qui pointe sur <type> (par exemple, `int *)

Examples

Pointeurs

D est un langage de programmation système qui vous permet de gérer et de gâcher votre mémoire manuellement. Néanmoins, D utilise un garbage collector par défaut pour libérer de la mémoire inutilisée.

D fournit des types de pointeur T * comme dans C:

```
void main()
{
    int a;
    int* b = &a; // b contains address of a
    auto c = &a; // c is int* and contains address of a

    import std.stdio : writeln;
    writeln("a ", a);
    writeln("b ", b);
    writeln("c ", c);
}
```

Allouer sur le tas

Un nouveau bloc de mémoire sur le tas est alloué à l'aide de la `new` expression, qui renvoie un pointeur sur la mémoire gérée:

```
void main()
{
    int* a = new int;
    *a = 42; // dereferencing
    import std.stdio : writeln;
    writeln("a: ", *a);
}
```

@safe D

Dès que la mémoire référencée par un n'est plus référencée via une variable du programme, le ramasse-miettes va libérer sa mémoire.

D permet également l'arithmétique du pointeur, sauf dans le code marqué comme `@safe` .

```
void safeFun() @safe
{
    writeln("Hello World");
    // allocating memory with the GC is safe too
    int* p = new int;
}

void unsafeFun()
{
    int* p = new int;
    int* fiddling = p + 5;
}

void main()
{
    safeFun();
    unsafeFun();
}
```

Pour plus d'informations sur SafeD, consultez l' article de l'équipe de conception D.

Lire Mémoire et pointeurs en ligne: https://riptutorial.com/fr/d/topic/6374/memoire-et-pointeurs

Chapitre 12: Modèles

Syntaxe

- identifiant de modèle (TemplateParameterList) {...}
- identifiant struct (TemplateParameterList) {...}
- identifiant de classe (TemplateParameterList) {...}
- Identifiant ReturnType (TemplateParameterList) (Liste de paramètres) {...}
- identifiant! (TemplateInvocationList)

Examples

Fonctionne avec un modèle

```
import std.stdio;

T min(T)(in T arg1, in T arg2) {
    return arg1 < arg2 ? arg1 : arg2;
}

void main() {
    //Automatic type inference
    writeln(min(1, 2));

    //Explicit type
    writeln(min!(ubyte)(1, 2));

    //With single type, the parenthesis might be ommited
    writeln(min!ubyte(1, 2));
}
```

modèle

Un modèle peut être introduit avec un `template` . Il peut contenir des fonctions, des classes et d'autres constructions.

```
template StaticArray(Type, size_t Length) {
     class StaticArray {
          Type content[Length];

          size_t myLength() {
            return getLength(this);
          }
     }

     private size_t getLength(StaticArray arr) {
          return Length;
     }
}

void main() {
```

```
    StaticArray!(int, 5) arr5 = new StaticArray!(int, 5);
    import std.stdio;
    writeln(arr5.myLength());
}
```

Lire Modèles en ligne: https://riptutorial.com/fr/d/topic/3892/modeles

Chapitre 13: Structs

Examples

Définir une nouvelle structure

Pour définir la structure appelée Personne avec un type entier variable age, type entier variable variable de type float ageXHeight:

```
struct Person {
    int age;
    int height;
    float ageXHeight;
}
```

Généralement:

```
struct structName {
    /+ values go here +/
}
```

Constructeurs Struct

En D, nous pouvons utiliser des constructeurs pour initialiser des structures comme une classe. Pour définir une construction pour la structure déclarée dans l'exemple précédent, nous pouvons taper:

```
struct Person {
    this(int age, int height) {
        this.age = age;
        this.height = height;
        this.ageXHeight = cast(float)age * height;
    }
}

auto person = Person(18, 180);
```

Lire Structs en ligne: https://riptutorial.com/fr/d/topic/4075/structs

Chapitre 14: Tableaux associatifs

Examples

Utilisation standard

```
int[string] wordCount(string[] wordList) {
    int[string] words;
    foreach (word; wordList) {
        words[word]++;
    }
    return words;
}

void main() {
    int[string] count = wordCount(["hello", "world", "I", "say", "hello"]);
    foreach (key; count.keys) {
        writefln("%s: %s", key, count[key]);
    }
    // hello: 2
    // world: 1
    // I: 1
    // say: 1

    // note: the order in the foreach is unspecified
}
```

Littéraux

```
int[string] aa0 = ["x": 5, "y": 6];  //int values, string keys
auto aa1 = ["x": 5.0, "y": 6.0];  // double values, string keys
string[int] aa2 = [10: "A", 11: "B"];  //string values, int keys
```

Ajouter des paires clé-valeur

```
int[string] aa = ["x": 5, "y": 6];
// The value can be set by its key:
aa["x"] = 7;
assert(aa["x"] == 7);
// if the key does not exist will be added
aa["z"] = 8;
assert(aa["z"] == 8);
```

Supprimer les paires clé-valeur

Supposons un tableau associatif `aa` :

```
int[string] aa = ["x": 5, "y": 6];
```

Les éléments peuvent être supprimés à l'aide de `.remove()` , si les `.remove()` clé sont supprimés et

que la `remove` renvoie `true` :

```
assert(aa.remove("x"));
```

si la clé donnée n'existe pas, `remove` ne fait rien et retourne `false` :

```
assert(!aa.remove("z"));
```

Vérifier si une clé existe

```
int[string] numbers = ["a" : 10, "b" : 20];

assert("a" in numbers);
assert("b" in numbers);
assert("c" in numbers);
```

Lire Tableaux associatifs en ligne: https://riptutorial.com/fr/d/topic/3159/tableaux-associatifs

Chapitre 15: Tableaux et tranches dynamiques

Syntaxe

- <type> [] <nom>;

Remarques

Les tranches génèrent une nouvelle vue sur la mémoire existante. Ils ne créent pas de nouvelle copie. Si aucune tranche ne contient de référence à cette mémoire - ou à une partie coupée de celle-ci - elle sera libérée par le ramasse-miettes.

En utilisant des tranches, il est possible d'écrire du code très efficace pour, par exemple, des analyseurs qui opèrent sur un seul bloc de mémoire et divisent simplement les parties sur lesquelles ils doivent travailler, sans besoin d'allouer de nouveaux blocs de mémoire.

Examples

Déclaration et initialisation

```
import std.stdio;

void main() {
    int[] arr = [1, 2, 3, 4];

    writeln(arr.length); // 4
    writeln(arr[2]); // 3

    // type inference still works
    auto arr2 = [1, 2, 3, 4];
    writeln(typeof(arr2).stringof); // int[]
}
```

Opérations de tableau

```
import std.stdio;

void main() {
    int[] arr = [1, 2, 3];

    // concatenate
    arr ~= 4;
    writeln(arr); // [1, 2, 3, 4]

    // per element operations
    arr[] += 10
    writeln(arr); // [11, 12, 13, 14]
```

```
}
```

Tranches

```
import std.stdio;

void main() {
    int[] arr = [1, 2, 3, 4, 5];

    auto arr2 = arr[1..$ - 1]; // .. is the slice syntax, $ represents the length of the array
    writeln(arr2); // [2, 3, 4]

    arr2[0] = 42;
    writeln(arr[1]); // 42
}
```

Lire Tableaux et tranches dynamiques en ligne: https://riptutorial.com/fr/d/topic/2445/tableaux-et-tranches-dynamiques

Chapitre 16: Traits

Syntaxe

- __traits (TraitsKeyword, TraitsArguments ...)

Examples

Itérer sur les membres d'un struct

```
import std.stdio;

struct A {
    int b;
    void c();
    string d;
};

void main() {
    // The following foreach is unrolled in compile time
    foreach(name; __traits(allMembers, A)) {
        pragma(msg, name);
    }
}
```

Le `allMembers` renvoie un tuple de chaîne contenant les noms des membres du type donné. Ces chaînes sont connues à la compilation.

Itération sur les membres d'une structure / classe sans leurs membres hérités

```
module main;

auto getMemberNames(T)() @safe pure {
    string[] members;

    foreach (derived; __traits(derivedMembers, T)) {
        members ~= derived;
    }

    return members;
}

class Foo {
    int a;
    int b;
}

class Bar : Foo {
    int c;
    int d;
    int e;
}
```

```
void main() {
    import std.stdio;

    foreach (member; getMemberNames!Bar) {
        writeln(member);
    }
}
```

dérivMembers renvoie un tuple de chaînes littérales, chaque chaîne étant le nom du membre.

Les exemples de sortie:

```
c
d
e
```

Lire Traits en ligne: https://riptutorial.com/fr/d/topic/3416/traits

Chapitre 17: UFCS - Syntaxe d'appel de fonction uniforme

Syntaxe

- aThirdFun (anotherFun (myFun (), 42); // notation commune (également valide)
- myFun (). anotherFun (42) .aThirdFun (); // UFCS
- myFun.anotherFun (42) .aThirdFun; // les accolades vides peuvent être supprimées

Remarques

Dans un appel `ab(args...)` , si le type `a` n'a pas de méthode nommée `b` , alors le compilateur essaiera de réécrire l'appel comme `b(a, args...)` .

Examples

Vérifier si un numéro est premier

```
import std.stdio;

bool isPrime(int number) {
    foreach(i; 2..number) {
        if (number % i == 0) {
            return false;
        }
    }

    return true;
}

void main() {
    writeln(2.isPrime);
    writeln(3.isPrime);
    writeln(4.isPrime);
    5.isPrime.writeln;
}
```

UFCS avec des gammes

```
void main() {
    import std.algorithm : group;
    import std.range;
    [1, 2].chain([3, 4]).retro; // [4, 3, 2, 1]
    [1, 1, 2, 2, 2].group.dropOne.front; //  tuple(2, 3u)
}
```

UFCS avec des durées de std.datetime

```
import core.thread, std.stdio, std.datetime;

void some_operation() {
    // Sleep for two sixtieths (2/60) of a second.
    Thread.sleep(2.seconds / 60);
    // Sleep for 100 microseconds.
    Thread.sleep(100.usecs);
}

void main() {
    MonoTime t0 = MonoTime.currTime();
    some_operation();
    MonoTime t1 = MonoTime.currTime();
    Duration time_taken = t1 - t0;

    writeln("You can do some_operation() this many times per second: ",
            1.seconds / time_taken);
}
```

Lire UFCS - Syntaxe d'appel de fonction uniforme en ligne:
https://riptutorial.com/fr/d/topic/4155/ufcs---syntaxe-d-appel-de-fonction-uniforme

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec le langage D	Alessio Sacco, André Puel, Bauss, Bennet Leff, Cauterite, Community, EsmaeelE, Gassa, Sirsireesh Kodali, Some coder, T.Furholzer, TuxCopter
2	Boucles	greenify, o3o
3	Cordes	Harry, o3o, RamenChef
4	Des classes	greenify, o3o
5	Désengagement	greenify
6	Évaluation de la fonction de temps de compilation (CTFE)	André Puel, greenify
7	Gammes	André Puel, Cauterite, Shriken
8	Gardes de portée	André Puel, greenify
9	Importations et modules	greenify
10	Les contrats	Quonux
11	Mémoire et pointeurs	greenify
12	Modèles	André Puel, Bauss, Quonux
13	Structs	Bennet Leff
14	Tableaux associatifs	Bauss, greenify, o3o, Shriken
15	Tableaux et tranches dynamiques	André Puel, Bauss, greenify, Harry, Shriken
16	Traits	André Puel, Bauss
17	UFCS - Syntaxe d'appel de fonction uniforme	André Puel, greenify, tre0n