Partie 2.1 - Le langage C

ECE_3TC31_TP/INF107

Guillaume Duc

2025

Introduction

Où en sommes-nous ?

- Nous avons construit un micro-processeur
- Nous avons vu comment le programmer en langage machine puis en assembleur
- Écrire un programme directement en langage machine ou en assembleur est extrêmement fastidieux
- Nous allons donc maintenant aborder un langage de programmation de plus haut niveau
- Il existe des centaines de langages de programmation
- Nous allons étudier le langage C

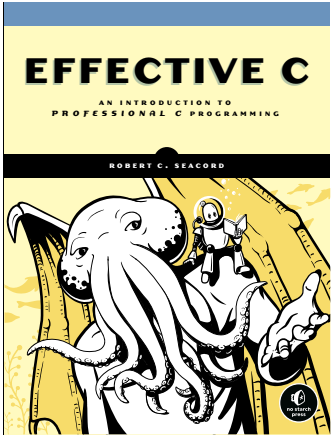
Pourquoi le C ?

- Le C est, depuis de très nombreuses années, un langage très populaire[1]
- C'est un langage de programmation de bas niveau (ou langage système) permettant d'utiliser directement certaines ressources de l'ordinateur (important pour la troisième partie du cours qui parlera de systèmes d'exploitation)
- Simple à prendre en main
  - ⚠ Cette simplicité apparente est en fait un gros défaut pour les débutants
  - Le compilateur est très permissif, il est donc très facile de faire des erreurs
- ⚠ Pour des raisons pédagogiques, certaines informations de ce cours sont volontairement simplifiées

[1] https://www.tiobe.com/tiobe-index/

Ressources pour aller plus loin

- **Livre**
  EFFECTIVE C
  An Introduction to Professional C Programming
  Robert C. Seacord
  No Starch Press, 2020
  ISBN-13: 978-1-71850-104-1
- **CPP Reference**
  https://en.cppreference.com/w/c



Bref historique

1972: Invention par *Dennis Ritchie* et Ken Thompson (Bell Telephone Laboratories) pour développer leur propre système d'exploitation … Unix (voir troisième partie du cours)

1989: Premier standard (ANSI C ou C89)
Adopté par l'ISO l'année suivante (C90)

1999: Nouveau standard ISO (C99) - très largement supporté
Type Boolean
Types entiers avec tailles fixes

2011: Nouveau standard ISO (C11) - bien supporté de nos jours
Support de l'Unicode
Support du multi-threading et des types atomiques

2017: Nouveau standard C (C17) - principalement des corrections

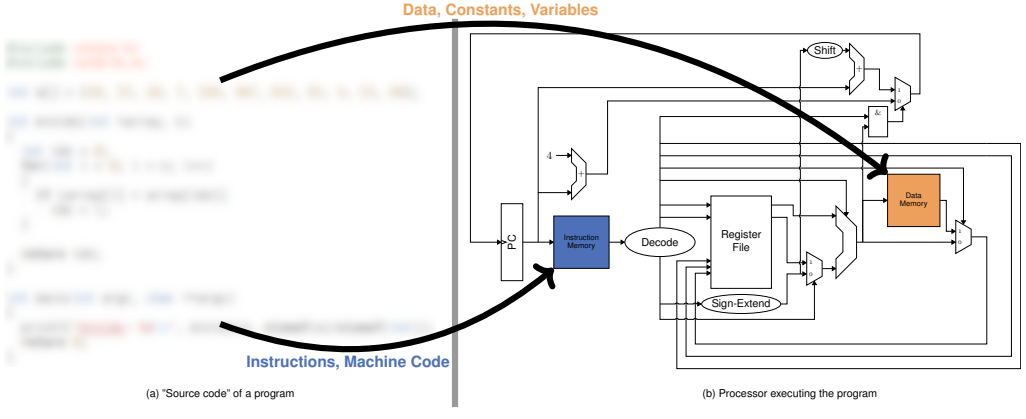
2024: Nouveau standard C (C23)

Standards

- Définissent le langage et la bibliothèque standard
- Standard $\neq$ Implémentation
  - Les implémentations n'intègrent par toujours l'intégralité du standard
  - Certaines fonctionnalités diffèrent en fonction de l'architecture ou du système d'exploitation
  - Certaines fonctionnalités sont **implementation-defined**, **unspecified**, ou même **undefined**
- **Nous utiliserons le standard C11 pour ce cours**
  - Moderne mais bien supporté

Comment passer du code source au langage machine : le compilateur

Du code source au langage machine (1)

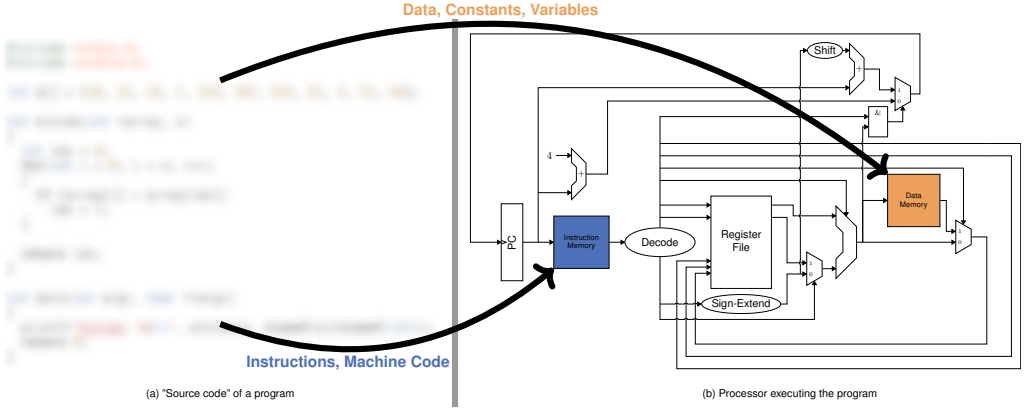


(a) "Source code" of a program

(b) Processor executing the program

Un outils, le **compilateur** traduit le code source en code machine (instructions) et en données (variables…) destinés à être placés dans les mémoires

Du code source au langage machine (2)

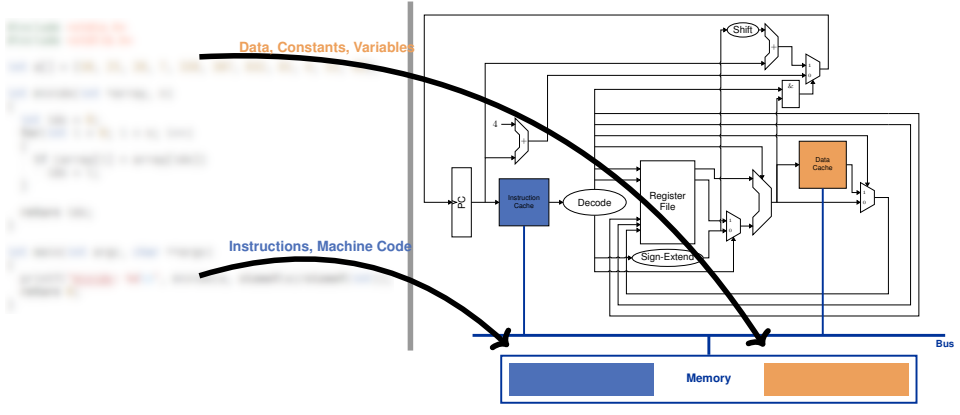


(a) "Source code" of a program

(b) Processor executing the program

Dans la partie 1, nous avons abordé une **architecture Harvard** (une mémoire pour le code et une pour les données) mais…

Du code source au langage machine (3)



(a) "Source code" of a program

(b) Processor executing the program

… nous utiliserons par la suite une **architecture Von Neumann** (le code et les données sont dans la même mémoire)

Compilateur

Un compilateur traduit du code source de haut niveau (exemple du C) en un fichier exécutable. On trouve dans ce dernier :

- Du code machine résultant de la traduction des instructions
  - Par exemple, une addition (`+`) devient l'instruction `add` (ou `addi`) pour le RISC-V
- Des informations concernant les données (variables, constantes…) utilisées par le programme
  - Adresses en mémoire et tailles
  - Éventuellement valeurs initiales

Pour que le programme s'exécute, il faut que le contenu du fichier exécutable soit correctement placé en mémoire (c'est le rôle du *loader* dans un système d'exploitation)

Un premier exemple

Le code source

```c
/* hello.c */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  printf("Hello world!\n");
  return EXIT_SUCCESS;
}
```

Compilation et exécution

Pour pouvoir exécuter ce premier programme, il faut le compiler pour produire un fichier exécutable puis demander l'exécution de ce fichier exécutable :

```
ouessant:~/tmp$ ls
hello.c

ouessant:~/tmp$ gcc -Wall -pedantic -std=c11 -g -o hello hello.c

ouessant:~/tmp$ ls
hello  hello.c

ouessant:~/tmp$ ./hello
Hello world!
```

⚠ **Attention** : si vous modifiez le fichier source (`hello.c`), vous devez relancer le compilateur pour produire le nouveau fichier exécutable avant de l'exécuter…

Compilation et exécution (détails pas à pas)

```
ouessant:~/tmp$ ls
hello.c
```

- `ouessant:~/tmp$` est l'invite de commande (*prompt*) affichée par votre interpréteur de commande (*shell*) pour vous… inviter à taper une commande. Elle peut varier et est représentée par `$` dans les TP
- `ls` est la commande tapée par l'utilisateur
- Utilisée sans arguments, cette commande liste les fichiers contenus dans le répertoire courant
- `hello.c` est le résultat de cette commande : il y a donc un seul fichier dans le répertoire courant et il est nommé `hello.c` (c'est le fichier contenant le code source)

Compilation et exécution (détails pas à pas)

```
ouessant:~/tmp$ gcc -Wall -pedantic -std=c11 -g -o hello hello.c
```

- `gcc` est le compilateur que l'on utilisera en TP. Il est suivi d'un certain nombre d'arguments qui permettent de contrôler son exécution
  - `-Wall` active tous les avertissements liés à des constructions souvent problématiques
  - `-pedantic` active les avertissements liés au respect strict du standard
  - `-std=c11` indique au compilateur de se baser sur le standard C11
  - `-g` indique au compilateur d'ajouter des informations au fichier exécutable pour faciliter le débogage par la suite
  - `-o hello` indique que le fichier exécutable produit doit s'appeler `hello` (par défaut : `a.out`)
  - `hello.c` indique le nom du fichier source à compiler

Le compilateur n'affiche rien, tout s'est donc bien passé. Si ce n'était pas le cas, lisez attentivement les messages, en commençant toujours par le début (et non la fin).

Compilation et exécution (détails pas à pas)

```
ouessant:~/tmp$ ls
hello  hello.c
```

- `ls` nous montre maintenant qu'il y a deux fichiers dans le répertoire courant :
  - `hello` le fichier exécutable produit par le compilateur
  - `hello.c` le fichier source

Compilation et exécution (détails pas à pas)

```
ouessant:~/tmp$ ./hello
Hello world!
```

- Pour lancer l'exécution de notre programme, il faut taper son nom : `hello`
- Cependant, pour des raisons expliquées ultérieurement, il faut dans notre cas préciser explicitement son emplacement d'où `./hello` :
  - `.` est le répertoire courant
  - `/` est le délimiteur de chemin
  - `hello` est le nom de l'exécutable
  - `./hello` est donc un chemin relatif : le fichier nommé `hello` dans le répertoire courant
- `Hello world!` a été affiché par notre programme
- ⚠ C'est le fichier exécutable qui est exécuté, pas le code source (vous pourriez même supprimer le fichier source). Si vous changez le code source, il faudra compiler de nouveau pour mettre à jour l'exécutable

Le langage C

Contenu d'un programme en C

```c
/* Inclusion de fichiers d'en-tête
   de la bibliothèque standard */
#include <stdio.h>
#include <stdlib.h>

// Définition de deux variables globales
char msg[] = "The answer is:";
int a = 21;

// Définition d'une fonction
int compute_answer() {
  return a * 2;
}

// Définition de la fonction main
int main(int argc, char *argv[])
{
  printf("%s %d\n", msg, compute_answer());
  return EXIT_SUCCESS;
}
```

Dans un programme en C, on peut trouver :

- Des commentaires (débutant par `//` et jusqu'à la fin de la ligne ou débutant par `/*` et jusqu'à `*/`)
- Des directives pour le pré-processeur, ici des `#include` pour inclure des fichiers d'en-têtes afin d'utiliser des fonctionnalités de la bibliothèque standard (`stdio.h` pour la fonction `printf` et `stdlib.h` pour `EXIT_SUCCESS`)
- Des déclarations ou des définitions de variables globales (ici la définition de deux variables `msg` et `a`)
- Des déclarations ou des définitions de fonctions (ici la définition de la fonction `compute_answer` et `main`)
  - La fonction `main` a une signification particulière : lors de l'exécution, le programme commence au début de cette fonction

Indentation

Contrairement à certains langages comme Python, l'indentation est totalement ignorée par le compilateur.

Néanmoins, l'indentation correcte de votre code est indispensable pour pouvoir facilement le lire !

```c
if (cond)
   printf("Inside if");
   printf("Outside if");
```

```c
if (cond)
   printf("Inside if");
printf("Outside if");
```

Les deux codes sont identiques. Par contre l'identation du premier est erronée.

Mots clés du langage (*Keywords*)

Ces mots clés ont une signification particulière, ils ne peuvent pas être utilisés comme nom de fonction, de variable…

`break`	`extern`	`static`	`auto`	`_Atomic` (C11)
`case`	`float`	`struct`	`goto`	`_Complex` (C99)
`char`	`for`	`switch`	`inline` (C99)	`_Generic` (C11)
`const`	`if`	`typedef`	`register`	`_Imaginary` (C99)
`continue`	`int`	`union`	`restrict` (C99)	`_Noreturn` (C11)
`default`	`long`	`unsigned`	`volatile`	`_Thread_local` (C11)
`do`	`return`	`void`		`_Alignas` (C11)
`double`	`short`	`while`		
`else`	`signed`	`_Alignof` (C11)		
`enum`	`sizeof`	`_Bool` (C99)		
		`_Static_assert` (C11)		

https://en.cppreference.com/w/c/keyword

Les types de base du C

Qu'est-ce qu'un type ?

La notion de type est un concept commun dans les langages de programmation.

Le type d'une variable ou d'une expression spécifie l'ensemble des valeurs acceptables, les règles de stockage (taille en mémoire, alignement…), les opérateurs utilisables, etc.

On classe les langages en deux familles :

- Les langages typés statiquement : tous les types (variables, expressions, etc.) doivent être connus et sont vérifiés à la compilation (ex. **C**, C++, Rust…)
- Les langages typés dynamiquement : tous les types sont déterminés et vérifiés à l'exécution (ex. Python, JavaScript…)

Note : le système de type peut vous permettre de détecter des erreurs. Tout comme une expression en physique doit être homogène du point de vue des unités, c'est la même chose en programmation. Si vous affectez une expression d'un certain type à une variable d'un autre type, il faut vous poser des questions (ce n'est pas nécessairement faux mais cela peut l'être).

Les types de base du C

- **Le type** `void` : Un type particulier sans valeurs
- **Le type booléen** : Pour représenter des valeurs booléennes (`true`/`false` ou `0`/`1`).
- **Les types entiers** : Pour représenter des nombres entiers
- **Les types à virgule flottante** : Pour représenter des nombres en utilisant la représentation à virgule flottante
- D'autres types seront vus dans la suite du cours (pointeurs, tableaux, types composés…)

https://en.cppreference.com/w/c/language/type

Le type `void`

La langage C définit un type spécial `void`

- Ce type n'a pas de valeurs
- Il est utilisé
  - Pour indiquer qu'une fonction ne renvoie pas de valeur
  - Pour indiquer qu'une fonction ne prend pas d'arguments
  - Pour indiquer un pointeur sans type particulier

Le type booléen

Le standard C99 a introduit le type `_Bool` qui peut prendre deux valeurs : 0 (alias `false`) et 1 (alias `true`).

Toute valeur différente de 0 est convertie implicitement vers 1 (`true`) si besoin :

`_Bool done = false;`	Initialise la variable `done` à `0`.
`_Bool isFalse = 0;`	Initialise la variable `isFalse` à `0`.
`_Bool isTrue = 5;`	Initialise la variable `isTrue` à `1`.
`_Bool isTrueToo = true;`	Initialise la variable `isTrueToo` à `1`.

Un alias `bool` est définit dans `stdbool.h` pour faciliter l'écriture. Il suffit d'ajouter `#include <stdbool.h>` pour pouvoir l'utiliser (à noter que `bool` est devenu un type à part entière en C23).

https://en.cppreference.com/w/c/language/arithmetic_types#Boolean_type

Les types entiers « classiques »

Le C définit plusieurs types entiers (liste non exhaustive) :

Signé	Non signé	Taille minimale garantie	En TP
`signed char`	`unsigned char`	au moins 8 bits	8 bits
`short int`	`unsigned short int`	au moins 16 bits	16 bits
`int`	`unsigned int`	au moins 16 bits	32 bits
`long int`	`unsigned long int`	au moins 32 bits	64 bits
`long long int`	`unsigned long long int`	au moins 64 bits	64 bits

La représentation exacte en mémoire n'est pas spécifiée mais en pratique, il s'agit d'une simple représentation en base 2 pour les entiers non signés et en complément à deux pour les nombre signés, ce qui donne, pour une taille de n bits, une plage de nombres représentables de $[0; 2^n - 1]$ pour les non signés et $[-2^{n-1}; 2^{n-1} - 1]$ pour les signés

https://en.cppreference.com/w/c/language/arithmetic_types

Alias (1)

Il existe plusieurs façon de nommer les types entiers signés

Type signé	Alias

`short int`	`short`
	`signed short`
	`signed short int`

`int`	`signed` `signed int`

`long int`	`long`
	`signed long`
	`signed long int`

`long long int`	`long long`
	`signed long long`
	`signed long long int`

Alias (2)

De même pour les types non signés

Type non signé	Alias
`unsigned short int`	`unsigned short`
`unsigned int`	`unsigned`
`unsigned long int`	`unsigned long`
`unsigned long long int`	`unsigned long long`

Aparté : littéraux entiers

On a souvent besoin d'écrire des nombres (*littéraux*) entiers dans un programme.

Par défaut, un littéral est interprété comme étant exprimé en base 10, sauf s'il est préfixé par :

- `0x` ou `0X` auquel cas il est exprimé en base 16, exemple `0x1A` (26 en base 10)
- `0` auquel cas il est exprimé en base 8, exemple `012` (10 en base 10) ⚠

Par défaut un littéral entier sera de type `int`, `long int` ou `long long int` (le premier type permettant de représenter le nombre) sauf s'il est suffixé par :

- `u` ou `U` auquel cas il sera `unsigned` (`unsigned int` ou `unsigned long`...)
- `l` ou `L` auquel cas il sera `long int` ou `long long int`
- `ll` ou `LL` auquel cas il sera `long long int`

https://en.cppreference.com/w/c/language/integer_constant

Hors programme : Les types entiers C99

C99 introduit de nouveaux types entiers (nécessitent `stdint.h`) :

- Des types de taille fixe : `intX_t` et `uintX_t` avec X = 8, 16, 32, 64 bits (ex. `int8_t`, `uint64_t`...)
- Des types les plus rapides de taille au moins X : `int_fastX_t` et `uint_fastX_t`
- Des types les plus petits de taille au moins X : `int_leastX_t` et `uint_leastX_t`
- Les types les plus grands possibles : `intmax_t`, `uintmax_t`
- Des types permettant de stocker des pointeurs dans des entiers

Types et littéraux flottants

Le C a trois types pour représenter des nombres réels en virgule flottante (au standard IEEE 754)

`float`	Simple-précision (32 bits)
`double`	Double-précision (64 bits)
`long double`	Précision étendue si disponible (128 bit)

Exemples de littéraux :

- `.5` (0.5)
- `4.2e-3` ($4.2 \cdot 10^{-3}$)
- `3.2e18` ($3.2 \cdot 10^{18}$)

https://en.cppreference.com/w/c/language/floating_constant

Types caractères

⚠ Nous ne parlons pour le moment que de caractères individuels, pas de chaînes de caractères.

En informatique, la gestion des caractères, au delà des 128 caractères de base (table ASCII) a longtemps été compliquée. De part son âge, cette complexité se retrouve en C.

Le type de base en C pour stocker un caractère est `char`. Il est équivalent, en fonction des implémentations, à un `signed char` or `unsigned char`, de taille typique 8 bits et peut donc stocker 256 valeurs (0 à 255 ou -128 à 127).

C'est un type **entier**. Il stocke donc nativement des nombres entiers. Comme il ne stocke que des nombres, il faut donc une méthode pour transformer des caractères en nombres et vice-versa.

Typiquement, c'est la table ASCII qui est utilisée. Elle affecte une valeur numérique entre 0 et 127 à un ensemble de caractères (26 lettres minuscules et majuscules, chiffres 0 à 9, des caractères spéciaux et des caractères de contrôle).

Types caractères : littéraux

Il est possible d'écrire un caractère littéral en le mettant entre simples quotes (`'`), exemple : `'A'`.

Un caractère littéral est de type `int` et sa valeur est la valeur donnée par une table de conversion, typiquement la table ASCII, pour ce caractère. Par exemple `'A'` est un entier de valeur 65.

https://en.cppreference.com/w/c/language/character_constant
https://en.wikipedia.org/wiki/ASCII

Types caractères : littéraux particuliers

Certains caractères ne peuvent pas être représentés littéralement directement et nécessite une **séquence d'échappement** :

Séquence	Description	Code ASCII	Séquence	Description	Code ASCII
`\f`	Form feed	12	`\'`	Single quote	39
`\n`	Line feed	10	`\"`	Double quote	34
`\r`	Carriage return	13	`\?`	Question mark	63
`\t`	Horizontal tab	9	`\\`	Backslash	92
`\v`	Vertical tab	11	`\a`	Audible bell	7
`\b`	Backspace	8			
`\n`	n an octal number	n	`\xh`	h a hex number	h

https://en.cppreference.com/w/c/language/escape

Types caractères : au delà de l'ASCII

Pour stocker des caractères au delà des 128 caractères ASCII classiques, plusieurs solutions existent

- `wchar_t` : un type caractère étendu, de taille définie par l'implémentation
- `char16_t` (C11) : un type caractère de 16 bits permettant de stocker un caractère Unicode encodé en UTF-16
- `char32_t` (C11) : un type caractère de 32 bits permettant de stocker un caractère Unicode encodé en UTF-32
- `char8_t` (C23) : un type caractère de 8 bits permettant de stocker un caractère Unicode encodé en UTF-8

Chaînes de caractères

Contrairement à certains langage, le C n'a pas de type spécifique pour les chaînes de caractères.

Une chaîne de caractères littérale est exprimée entre double quotes : `"Hello world\n"`.

Pour ce premier cours, nous traiterons ces chaînes comme des tableaux de `char` (sera raffiné par la suite).

Exemple :

```
char hi[] = "Hello World\n";
```

https://en.cppreference.com/w/c/language/string_literal

Déclarations et définitions

Structure d'un fichier C

On a vu qu'un fichier C pouvait contenir :

- Des commentaires (déjà vus)
- Des directives pour le pré-processeur (sera vu plus tard)
- Des déclarations ou définitions globales de variables
- Des déclarations ou définitions globales de fonctions (sera vu plus tard)

Déclaration

Un **déclaration** permet d'introduire au compilateur un nouvel identifiant (*identifier*).

- Un identifiant est un nom qui a une signification particulière dans un programme (nom d'une variable, d'une fonction…)
  - C'est une suite de caractères (lettres, chiffres, underscore…)
  - Il ne peut pas commencer par un chiffre
  - Il est sensible à la casse (majuscule/minuscule)

En plus d'introduire ce nouvel identifiant, une déclaration précise

- Son type (pour une variable, pour une fonction : son type de retour, le nombre et le type de ses arguments)
- Quelques propriétés supplémentaires (constantes par exemple)

https://en.cppreference.com/w/c/language/identifier

Déclaration et définition

Une **définition** réalise une déclaration et en plus :

- Pour une variable, la définition demande explicitement au compilateur de réserver de la mémoire pour cette variable
- Pour une fonction, la définition introduit le corps (code) de la fonction

Déclaration et définition : pourquoi cette distinction ?

- Lorsque le compilateur rencontre un identifiant, il doit avoir vu au préalable sa déclaration (ou sa définition) pour vérifier son usage correct vis-à-vis de la syntaxe, du système de type, etc. Dans ce rôle, la déclaration suffit
- Pour construire le programme final, la mémoire pour chaque variable doit être réservée, et le compilateur doit avoir le code de toutes les fonctions utilisées. Pour cette usage, il faut que chaque variable et fonction ait été définie (une et une seule fois)
- Pour chaque variable et fonction
  - On peut avoir 0 ou plus déclarations
  - On doit avoir une et une seule définition
  - Il faut que toutes les déclarations et la définition soient cohérentes
- L'utilité des déclarations n'est peut être pas évidente (mais reste utile néanmoins) si le programme est constitué d'un seul fichier source mais le devient avec plusieurs fichiers source

Variables

Syntaxe simplifiée (pour les variables)

Cette syntaxe est simplifiée et sera complétée par la suite…

Déclaration

```
extern <type> <identifiant>;
```

Définition

```
<type> <identifiant> [= <valeur initiale>];
```

https://en.cppreference.com/w/c/language/declarations

Syntaxe simplifiée (pour les variables) : exemples

`int counter = 0;`	Définit la variable `counter` de type `int` avec comme valeur initiale 0
`char ch;`	Définit la variable `ch` de type `char` sans valeur initiale
`char ch2 = 'A';`	Définit la variable `ch2` de type `char` avec comme valeur initiale 65
`extern float val;`	Déclare la variable `val` de type `float`

Portée d'une variable

Une déclaration/définition peut être :

- À l'extérieur d'une fonction, on parle alors de variable **globale**
- À l'intérieur d'une fonction, on parle alors de variable **locale**

Chaque variable est associée à une **portée** (*scope*) qui représente la portion du code où cette variable est visible/utilisable.

- La portée d'une variable globale est l'ensemble du programme (moyennant que la variable est été déclarée avant son utilisation)
  - Sauf si la définition est précédée du mot clé `static` auquel cas la portée est limitée à l'unité de compilation (fichier C) (*static linkage*)
- La portée d'une variable locale est le bloc de code dans laquelle elle a été définie (exemple : corps d'une fonction, bloc de code imbriqué…)

Durée de vie d'une variable

Une notion de **durée de vie** est également associée à chaque variable et représente le laps de temps durant lequel elle est utilisable.

- La durée de vie d'une variable globale est la durée de vie du programme
- La durée de vie d'une variable locale est la durée de vie de la fonction/bloc de code dans laquelle est définie
  - Sauf si la définition est précédée du mot clé `static` auquel cas la durée de vie est étendue à celle du programme

Constantes

```c
const int answer = 42;
```

Le mot clé `const` permet d'indiquer que le contenu de la variable est constant.

Tableaux

Tableaux (première partie)

Définition d'un tableau

```
int a[10]; // Tableau contenant 10 int
short b[3] = {1, 2, 3}; // Tableau de 3 shorts (1, 2 et 3)
int c[] = {4, 5}; // Tableau de 2 int (4 et 5)
char d[] = "Hello"; // Tableau de 6 caractères 'H', 'e', 'l', 'l', 'o', '\0'
```

Utilisation d'un tableau

```
e = a[0] + c[1];
```

Les indices sont numérotés à partir de 0

⚠ Aucune vérification n'est faite concernant un éventuel dépassement des limites du tableau ⚠

Fonctions

Syntaxe de déclaration et définition d'une fonction

Déclaration

```
[extern] <type de retour> <nom de la fonction>(<types des arguments>);
```

Le mot clé `extern` est facultatif ici.

Définition

```
<type de retour> <nom de la fonction>(<types et noms des arguments>) {
   <corps>
}
```

https://en.cppreference.com/w/c/language/function_definition

La définition d'une fonction est nécessairement globale (à l'extérieur de toute fonction). À noter, quelques compilateurs supportent des fonctions imbriquées.

Exemples de déclarations

- `void foo();` déclare une fonction `foo` qui ne prend aucun argument et qui ne retourne rien
- `int bar(char, short);` déclare une fonction `bar` qui prend deux arguments, un de type `char` et l'autre de type `short` et renvoie une valeur de type `int`
- `int baz(char a, short b);` déclare une fonction `baz` qui prend deux arguments, un de type `char` nommé `a` et l'autre de type `short` nommé `b` et renvoie une valeur de type `int`

À noter, il n'est pas nécessaire de nommer les arguments lors des *déclarations* des fonctions.

Portée des fonctions

Par défaut, la portée des fonctions est l'ensemble du programme (moyennant déclaration avant utilisation).

Il est possible de restreindre la portée d'une fonction à une unité de compilation (fichier C) en préfixant sa définition avec le mot clé `static` (*static linkage*). Cela est utile pour les fonctions qui n'ont pas besoin d'être utilisées en dehors d'un fichier C en particulier.

Corps d'une fonction

Le corps d'une fonction (fourni lors de sa définition) est une séquence de **statements** et de **déclarations/définitions** :

- *Compound statement* : séquence de statements et de déclarations encadrés par des accolades (`{ }`)
- *Expression statement* : une expression suivie d'un `;` (exemple `a = 1 + 2;`)
- *Selection statement*
  - Constructions `if` ou `if-else`
  - Construction `switch`
- *Iteration statement*
  - Constructions `while`, `do-while` ou `for`
- *Jump statement*
  - `return`
- Déclaration ou définition d'une variable

https://en.cppreference.com/w/c/language/statements

Expressions

Expressions

Une expression est une séquence d'opérateurs avec leurs opérandes qui spécifie un calcul à effectuer.

Exemples :

- `a = 4 + 5`
- `foo(a, 8 * 9) % 42`

Les opérandes peuvent être :

- Des littéraux (exemple `4`)
- Des noms de variables (exemple `a`)
- Des fonctions avec leurs arguments (exemple `foo(a, 8 * 9)`), la fonction est alors appelée avec ses arguments et est ensuite remplacée par sa valeur de retour

https://en.cppreference.com/w/c/language/expressions

Priorité et associativité des opérateurs

Chaque opérateur possède une priorité et une associativité :

- **Associativité** : Détermine dans quel ordre traiter des opérateurs de même priorité
  - Associativité à gauche
    `a - b + c + d` est égal à `((a - b) + c) + d`
  - Associativité à droite
    `- ~ -a` est égal à `(- (~ (- a)))`
- **Priorité** : Détermine dans quel ordre traiter des opérateurs de priorités différentes
  - `-a + b * c` est égal à `(-a) + (b * c)`

https://en.cppreference.com/w/c/language/operator_precedence

Opérateurs (1)

Priorité	Opérateur	Description	Associativité
1	++ --	Post-incrémentation/décrémentation	Gauche
	[ ]	Accès à un tableau	Gauche
	( )	Appel de fonction	Gauche
	++ --	Pré-incrémentation/décrémentation	Droite
2	+ -	Plus/moins unaire	Droite
	!	NON logique	
	~	NON bit-à-bit	
3	*	Multiplication	Gauche
	/	Division	
	%	Reste/Modulo	
4	+	Addition	Gauche
	-	Soustraction	
5	<< >>	Décalage à gauche Décalage à droite	Gauche

Opérateurs (2)

Priorité	Opérateur	Description	Associativité
6	<	Plus petit	Gauche
	<=	Plus petit ou égal	
	>	Plus grand	
	>=	Plus grand ou égal	
7	==	Égal	Gauche
	!=	Différent	
8	&	ET bit-à-bit	Gauche
9	^	OU exclusif bit-à-bit	Gauche
10	|	OU bit-à-bit	Gauche
11	&&	ET logique (court-circuit)	Gauche
12	||	OU logique (court-circuit)	Gauche
13	? :	Opérateur ternaire	Droite
14	=	Affectation	Droite

Signification des opérateurs

La plupart des opérateurs ont une signification évidente.

Pour les autres, n'hésitez pas à consulter la documentation :
https://en.cppreference.com/w/c/language/operator_arithmetic
https://en.cppreference.com/w/c/language/operator_logical
https://en.cppreference.com/w/c/language/operator_comparison
https://en.cppreference.com/w/c/language/operator_assignment

Structures de contrôle

if

(1) `'if' '(' <Cond> ')' <Statement-true>`
(2) `'if' '(' <Cond> ')' <Statement-true> 'else' <Statement-false>`

L'expression `<Cond>` est évaluée

- Si le résultat n'est pas 0, le statement `<Statement-true>` est exécuté
- Sinon, dans la deuxième variante, le statement `<Statement-false>` est exécuté

⚠ `<Statement-true>` et `<Statement-false>` correspondent chacun à un unique *statement*. Si vous voulez mettre plusieurs *statements*, il vous faut les groupes dans un *compound statement* avec des accolades ({ et })

Exemple

```c
if (a + b < c) c = a + b;
else {
  c = b / 2;
  d = 3;
}
```

https://en.cppreference.com/w/c/language/if

switch (1)

Une construction `switch` permet d'exécuter une branche de code parmi plusieurs en fonction d'une condition

```
'switch' '(' <Cond> ')' '{' <Cases> '}'
```

`<Cases>` admet deux formats

(1) `'case' <Const-expr> ':' <Sub-statement>`

(2) `'default' ':' <Sub-statement>`

- La condition `<Cond>` est évaluée
- L'exécution continue alors dans la branche dont la valeur `<Const-expr>` est égale au résultat de l'évaluation

https://en.cppreference.com/w/c/language/switch

`switch` (2)

- Si aucun cas ne correspond, l'exécution continue avec le cas `default` s'il est présent (un seul cas `default` est autorisé)
- ⚠ Une fois l'exécution débutée dans une branche, elle continue dans les branches suivantes sauf si un statement `break` est rencontré
  - Il est donc très fréquent de mettre un `break` à la fin de chaque branche (son absence est souvent une erreur)

`switch` : exemple

```
1  int counter = 0;
2  switch (cond) {
3    case 4: counter = counter + 1;
4    case 3: counter = counter + 1;
5    case 2: counter = counter + 1;
6            break;
7    case 1: break;
8    default: counter = 1000;
9  }
10 counter = counter * 2;
```

On suppose que `cond` est de type `int`

Valeur de `cond`	Lignes exécutées	Valeur finale de `counter`
1	1, 2, 7, 10	0
2	1, 2, 5-6, 10	2
3	1, 2, 4-6, 10	4
4	???	???
5	???	???

Boucle `while`

Dans une boucle `while`, le corps est exécuté tant que la condition est évaluée à une valeur différente de zéro

```
'while' '(' <Cond> ')' <Sub-statement>
```

- La condition `<Cond>` est évaluée
  - Si le résultat est différent de zéro, `<Sub-statement>` est exécuté
    - Puis la condition est de nouveau évaluée et ainsi de suite

⚠ `<Sub-statement>` est un *statement* (si vous en voulez plusieurs, `{ ... }`)

Note : Le corps de la boucle (`<Sub-statement>`) peut ne pas être exécuté du tout

```
while (counter > 5)
   counter = counter - 1;
```

https://en.cppreference.com/w/c/language/while

Boucle do ... while

```
'do' <Sub-statement> 'while' '(' <Cond> ')'
```

- <Sub-statement> est exécuté en premier
- Puis la condition <Cond> est évaluée
  - Si le résultat est différent de zéro, <Sub-statement> est de nouveau exécuté
    - Puis la condition est de nouveau évaluée et ainsi de suite

Note : Le corps de la boucle (<Sub-statement>) est exécuté au moins une fois

```
do
{
  counter = counter - 1;
} while (counter > 5)
```

https://en.cppreference.com/w/c/language/do

Boucle `for` (1)

```
'for' '(' <Init> ';' <Cond> ';' <Iteration> ')' <Sub-statement>
```

- L'expression `<Init>` est évaluée une seule fois au début
- La condition `<Cond>` est ensuite évaluée. Si le résultat est différent de zéro
  - Le corps de la boucle (`<Sub-statement>`) est exécuté
  - Puis l'expression `<Iteration>` est évaluée
  - Enfin la condition est réévaluée et ainsi de suite…

https://en.cppreference.com/w/c/language/for

Boucle `for` (2)

La boucle `for` suivante :

```
'for' '(' <Init> ';' <Cond> ';' <Iteration> ')' <Sub-statement>
```

est donc équivalente à :

```
<Init> ';'
'while' '(' <Cond> ')'
'{'
  <Sub-statement>
  <Iteration> ';'
'}'
```

Boucle for (3)

```c
for (int i = 0; i < 10; i = i + 1)
{
   printf("i = %d", i);
}
```

Diversion dans les boucles

Au sein d'une boucle

- `break`
  - Sort immédiatement de la boucle, l'exécution continue après la boucle
- `continue`
  - Saute le reste de l'itération courante de la boucle
  - Pour une boucle `while` et `do`, l'exécution continue par l'évaluation de la condition
  - Pour une boucle `for`, l'exécution continue par l'évaluation de l'expression de fin d'itération

Si plusieurs boucles sont imbriquées, `break` et `continue` agissent sur la boucle la plus imbriquée

```
while (cond1) {
  while (cond2) {
    break; // Sort de la boucle while (cond2)
  }
}
```

https://en.cppreference.com/w/c/language/break
https://en.cppreference.com/w/c/language/continue

`return`

`return` permet de quitter une fonction et éventuellement de spécifier la valeur de retour de celle-ci.

- Pour une fonction ne retournant pas de valeur (type de retour `void`)
  - `return;` quitte immédiatement la fonction
  - Si le corps de la fonction se termine sans `return`, la fonction se termine à ce moment là (équivalent à avoir un `return;` à la fin du corps de la fonction)
- Pour une fonction retournant une valeur de type autre que `void`
  - `return` doit spécifier la valeur de retour de cette fonction : `return <Expression> ;`.
  - ⚠ Si le corps de la fonction se termine sans un `return`, le comportement est non défini (à l'exception du cas particulier de la fonction `main`)

https://en.cppreference.com/w/c/language/return

Tout ensemble

Un exemple plus ambitieux : division par soustraction successives

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

unsigned int division(unsigned int dividend, unsigned int divisor) {
  unsigned int result = 0;
  for (unsigned int rest = dividend; rest >= divisor; result++)
    rest = rest - divisor;
  return result;
}

const char message[] = "Hello World";
short data = 25;
int division_result;

int main(int argc, char *argv[]) {
  division_result = division(data, 7) + 2;
  printf("%s\n", message);
  printf("%d\n", division_result);
  return EXIT_SUCCESS;
}
```

Contenu de `division.c`.

Exécutons notre programme

Pour rappel (voir les premiers slides)

```
ouessant:~/tmp$ ls
division.c

ouessant:~/tmp$ gcc -Wall -pedantic -std=c11 -g -o division division.c

ouessant:~/tmp$ ls
division  division.c

ouessant:~/tmp$ ./division
Hello World
5
```

La fonction `main`

- Signature `int main(int argc, char *argv[])`
- Exécutée automatiquement au démarrage du programme
- Arguments
  - `argc`: nombre d'arguments passés sur la ligne de commande
  - `argv`: tableau de chaînes de caractères contenant les différents arguments
- Valeur de retour (`int`) : code de sortie du programme, utilisé par le système d'exploitation pour déterminer s'il s'est bien exécuté ou non. Deux constantes (définies dans `stdlib.h`) peuvent faciliter l'écriture : `EXIT_FAILURE` et `EXIT_SUCCESS`
- Exemple : `./division one 2` sur la ligne de commande se traduit par
  `argc`: 3
  `argv[0]`: `"./division"`
  `argv[1]`: `"one"`
  `argv[2]`: `"2"`

La bibliothèque standard C (*C standard library*)

La bibliothèque standard

La bibliothèque standard C (aussi connue sous le nom `libc`) fournit des fonctions de base pour faciliter l'écriture d'un programme

- Par exemple
  - Des fonctions mathématiques https://en.cppreference.com/w/c/numeric
  - Des fonctions pour gérer la date, l'heure et le temps https://en.cppreference.com/w/c/chrono
  - Des fonctions pour manipuler des fichiers https://en.cppreference.com/w/c/io
  - Des fonctions pour manipuler des chaînes de caractères https://en.cppreference.com/w/c/string
- La liste complète https://en.cppreference.com/w/c/header

Utilisation des fonctions de la bibliothèque standard

On rappelle que le compilateur doit avoir vu la déclaration d'une fonction avant de pouvoir l'utiliser. C'est également le cas pour les fonctions de la bibliothèque standard.

Ces déclarations sont regroupés dans des fichiers d'en-tête (*header file*).

Un fichier d'en-tête est un fichier C normal (avec par convention une extension `.h` au lieu de `.c`). Il contient uniquement des déclarations de fonction, de variables et de types.

Donc pour utiliser une fonction de la bibliothèque standard, il suffit d'inclure le fichier d'en-tête qui contient sa déclaration. On utilisera la direction `#include` du pré-processeur pour le faire.

Exemple, pour utiliser la fonction `printf`, on ajoutera en début du fichier source :
`#include <stdio.h>`

Dans quel fichier se trouve l'en-tête d'une fonction ? Voir `man` ou
https://en.cppreference.com/w/c/header.html

Fonction `printf` (1)

La fonction `printf` permet d'afficher des informations formattées (entiers, caractères, chaînes…)

Déclaration : `int printf(const char format[], ... );`

Le premier argument est toujours une chaîne de caractères indiquant à `printf` quoi faire (chaîne de format)

`printf` peut prendre d'autres arguments, en fonction de ce qui est indiqué par `format` (c'est une fonction un peu particulière appelée *variadic* https://en.cppreference.com/w/c/variadic)

Exemple : `printf("A number: %d\n", 5)`

https://en.cppreference.com/w/c/io/fprintf

Fonction `printf` (2)

La chaîne `format` en premier argument est traitée de la manière suivante :

- Les caractères réguliers sont simplement affichés à l'écran
- Le caractère % est particulier et introduit une instruction de formatage
  - Chaque % va se référer à un argument successif de l'appel à `printf` (le premier au deuxième argument de `printf`, le deuxième au troisième argument, etc.)
  - Les caractères suivant % indique comment interpréter l'argument et comment l'afficher
- Dans un premier temps, voici les conversions

%c	Affiche le caractère correspondant
%d	Affiche un entier signé (`int`) en base 10
%u	Affiche un entier non signé (`unsigned int`) en base 10
%x	Affiche un entier (`unsigned int`) en base 16
%f	Affiche un flottant (`double`) en décimal
%e	Affiche un flottant (`double`) en notation exponentielle
%s	Affiche une chaîne de caractères

`printf` : exemple (1)

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

int main(int argc, char *argv[])
{
  char c1 = 'a', c2 = 97;
  unsigned short s = 540;
  int i = 0xfbfb;
  float f = i * 1.133e5;
  static const char string[] = "Some string\nwith a line break.";

  printf("Character symbols: %c and %c are the same\n", c1, c2);
  printf("Characters as numbers: %d and 0x%x are the same\n", c1, c2);
  printf("Integer numbers (decimal) : %u and %d\n", s, i);
  printf("Integer numbers (hex): 0x%x and 0x%X\n", s, i);
  printf("Floating-point numbers: %f and %e\n", f, f);
  printf("String: %s\n", string);
  printf("Argument: %s\n", argv[0]);

  return EXIT_SUCCESS;
}
```

Contenu de `print.c`.

printf : exemple (2)

```
ouessant:~/tmp$ ls
print.c

ouessant:~/tmp$ gcc -Wall -pedantic -std=c11 -g -o print print.c

ouessant:~/tmp$ ls
print print.c

ouessant:~/tmp$ ./print
Character symbols: a and a are the same
Characters as numbers: 97 and 0x61 are the same
Integer numbers (decimal) : 540 and 64507
Integer numbers (hex): 0x21c and 0xFBFB
Floating-point numbers: 7308643328.000000 and 7.308643e+09
String: Some string
with a line break.
Argument: ./print
```

Annexes

Crédits

- Le cours se base sur celui écrit par Florian Brandner pour 3TC31