Base I

Programmation séquentielle en C, 2020-2021

Orestis Malaspinas (A401), ISC, HEPIA

2020-09-16

Inspirés des slides de F. Glück

Historique (1/2)

- Conçu initialement pour la programmation des systèmes d'exploitation (UNIX).
- Créé par Dennis Ritchie à Bell Labs en 1972 dans la continuation de CPL, BCPL et B.
- Standardisé entre 1983 et 1988 (ANSI C).
- La syntaxe de C est devenue la base d'autres langages comme C++, Objective-C, Java, Go, C#, Rust, etc.
- Révisions plus récentes, notamment C99, C11, puis C18.

Historique (2/2)

- Développement de C lié au développement d'UNIX.
- UNIX a été initialement développé en assembleur:
  - instructions de très bas niveau
  - instructions spécifiques à l'architecture du processeur.
- Pour rendre UNIX portable, un langage de *haut niveau* (en 1972) était nécessaire.
- Comparé à l'assembleur, le C est :
  - Un langage de "haut niveau": C offre des fonctions, des structures de données, des constructions de contrôle de flots (`while`, `for`, etc).
  - Portable: un programme C peut être exécuté sur un *très grand nombre* de plateformes (il suffit de recompiler le *même code* pour l'architecture voulue).

Qu'est-ce que le C?

- "Petit langage simple" (en 2020).
- Langage compilé, statiquement (et faiblement) typé, procédural, portable, très efficace.
- Langage "bas niveau" (en 2020): management explicite et manuelle de la mémoire (allocation/désallocation), grande liberté pour sa manipulation.
- Pas de structures de haut niveau: chaînes de caractères, vecteurs dynamiques, listes, ...
- Aucune validation ou presque sur la mémoire (pointeurs, overflows, ...).

Exemple de programme

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  printf("Enter n: "); // affichage
  int n = 0; // déclaration et initialisation de n
  scanf("%d", &n); // entrée au clavier
  int sum = 0; // déclaration et initialisation de sum
  for (int i = 0; i <= n; ++i) { // boucle
    sum += i;
  }
  printf("Sum of the %d first integers: %d\n", n, sum);
  if (sum != n * (n+1) / 2) { // branchement cond.
    printf("Error: the answer is wrong.\n");
    return EXIT_FAILURE; // code d'erreur
  }
  return EXIT_SUCCESS; // code de réussite
}
```

Génération d'un exécutable

- Pour pouvoir être exécuté un code C doit être d'abord compilé (avec `gcc` ou `clang`).
- Pour un code `prog.c` la compilation "minimale" est

```
$ clang prog.c
$ ./a.out # exécutable par défaut
```

- Il existe une multitude d'options de compilation:

```
$ clang -O1 -std=c11 -Wall -Wextra -g porg.c -o prog
-fsanitize=address -fsanitize=leak -fsanitize=undefined
```

1. `-std=c11` utilisation de C11.
2. `-Wall et -Wextra` activation des warnings.
3. `-fsanitize=...` contrôles d'erreurs à l'exécution (coût en performance).
4. `-g` symboles de débogages sont gardés.
5. `-o` défini le fichier exécutable à produire en sortie.
6. `-O1`, `-O2`, `-O3`: activation de divers degrés d'optimisation

La simplicité de C?

32 mots-clé et c'est tout

`auto`	`double`	`int`	**`struct`**
`break`	**`else`**	`long`	**`switch`**
`case`	**`enum`**	`register`	**`typedef`**
`char`	**`extern`**	**`return`**	**`union`**
`const`	`float`	`short`	`unsigned`
`continue`	**`for`**	`signed`	`void`
`default`	**`goto`**	**`sizeof`**	`volatile`
`do`	**`if`**	`static`	**`while`**

Déclaration et typage

En C lorsqu'on veut utiliser une variable (ou une constante), on doit déclarer son type

```c
const double two = 2.0; // déclaration et init.
int x;   // déclaration (instruction)
char c;  // déclaration (instruction)
x = 1;   // affectation (expression)
c = 'a'; // affectation (expression)
int y = x; // déclaration et initialisation en même temps
int a, b, c; // déclarations multiples
a = b = c = 1; // init. multiples
```

Types de base (1/4)

Numériques

Type	Signification (**gcc pour x86-64**)
`char`, `unsigned char`	Entier signé/non-signé 8-bit
`short`, `unsigned short`	Entier signé/non-signé 16-bit
`int`, `unsigned int`	Entier signé/non-signé 32-bit
`long`, `unsigned long`	Entier signé/non-signé 64-bit
`float`	Nombre à virgule flottante, simple précision
`double`	Nombre à virgule flottante, double précision

La signification de `short`, `int`, ... dépend du compilateur et de l'architecture.

Types de base (2/4)

Voir `<stdint.h>` pour des représentations **portables**

Type	Signification
`int8_t`, `uint8_t`	Entier signé/non-signé 8-bit
`int16_t`, `uint16_t`	Entier signé/non-signé 16-bit
`int32_t`, `uint32_t`	Entier signé/non-signé 32-bit
`int64_t`, `uint64_t`	Entier signé/non-signé 64-bit

Types de base (3/4)

Booléens

- Le ANSI C n'offre pas de booléens.
- L'entier `0` signifie *faux*, tout le reste *vrai*.
- Depuis C99, la librairie `stdbool` met à disposition un type `bool`.
- En réalité c'est un entier:
  - $1 \Rightarrow$ `true`
  - $0 \Rightarrow$ `false`
- On peut les manipuler comme des entier (les sommer, les multiplier, ...).

Quiz: booléens

Quiz: booléens

Types de base (4/4)

Conversions

- Les conversions se font de manière:
  - Explicite:

```
int a = (int)2.8;
double b = (double)a;
int c = (int)(2.8+0.5);
```

  - Implicite:

```
int a = 2.8; // warning, si activés, avec clang
double b = a + 0.5;
char c = b; // pas de warning...
int d = 'c';
```

Quiz: conversions

Expressions et opérateurs (1/6)

Une expression est tout bout de code qui est **évalué**.

Expressions simples

- Pas d'opérateurs impliqués.
- Les littéraux, les variables, et les constantes.

```
const int L = -1; // 'L' est une constante, -1 un littéral
int x = 0;        // '0' est un litéral
int y = x;        // 'x' est une variable
int z = L;        // 'L' est une constante
```

Expressions complexes

- Obtenues en combinant des *opérandes* avec des *opérateurs*

```
int x;     // pas une expression (une instruction)
x = 4 + 5; // 4 + 5 est une expression
           // dont le résultat est affecté à 'x'
```

Expressions et opérateurs (2/6)

Opérateurs relationnels

Opérateurs testant la relation entre deux *expressions*:

- `(a opérateur b)` retourne `1` si l'expression s'évalue à `true`, `0` si l'expression s'évalue à `false`.

Opérateur	Syntaxe	Résultat
`<`	`a < b`	1 si a $<$ b; 0 sinon
`>`	`a > b`	1 si a $>$ b; 0 sinon
`<=`	`a <= b`	1 si a $<=$ b; 0 sinon
`>=`	`a >= b`	1 si a $>=$ b; 0 sinon
`==`	`a == b`	1 si a $==$ b; 0 sinon
`!=`	`a != b`	1 si a $!=$ b; 0 sinon

Expressions et opérateurs (3/6)

Opérateurs logiques

Opérateur	Syntaxe	Signification
`&&`	`a && b`	ET logique
`||`	`a || b`	OU logique
`!`	`!a`	NON logique

Quiz: opérateurs logiques

Expressions et opérateurs (4/6)

Opérateurs arithmétiques

Opérateur	Syntaxe	Signification
+	`a + b`	Addition
-	`a - b`	Soustraction
*	`a * b`	Multiplication
/	`a / b`	Division
%	`a % b`	Modulo

Expressions et opérateurs (5/6)

Opérateurs d'assignation

Opérateur	Syntaxe	Signification
`=`	`a = b`	Affecte la valeur `b` à la variable `a` et retourne la valeur de `b`
`+=`	`a += b`	Additionne la valeur de `b` à `a` et assigne le résultat à `a`.
`-=`	`a -= b`	Soustrait la valeur de `b` à `a` et assigne le résultat à `a`.
`*=`	`a *= b`	Multiplie la valeur de `b` à `a` et assigne le résultat à `a`.
`/=`	`a /= b`	Divise la valeur de `b` à `a` et assigne le résultat à `a`.
`%=`	`a %= b`	Calcule le modulo la valeur de `b` à `a` et assigne le résultat à `a`.

Expressions et opérateurs (6/6)

Opérateurs d'assignation (suite)

Opérateur	Syntaxe	Signification
`++`	`++a`	Incrémente la valeur de a de 1 et retourne le résultat (`a += 1`).
`--`	`--a`	Décrémente la valeur de a de 1 et retourne le résultat (`a -= 1`).
`++`	`a++`	Retourne a et incrémente a de 1.
`--`	`a--`	Retourne a et décrémente a de 1.

Structures de contrôle: `if .. else if .. else` (1/2)

Syntaxe

```
if (expression) {
    instructions;
} else if (expression) { // optionnel
                         // il peut y en avoir plusieurs
    instructions;
} else {
    instructions; // optionnel
}
```

```
if (x) { // si x s'évalue à `vrai`
    printf("x s'évalue à vrai.\n");
} else if (y == 8) { // si y vaut 8
    printf("y vaut 8.\n");
} else {
    printf("Ni l'un ni l'autre.\n");
}
```

Structures de contrôle: if .. else if .. else (2/2)

Pièges

```
int x, y;
x = y = 3;
if (x = 2)
    printf("x = 2 est vrai.\n");
else if (y < 8)
    printf("y < 8.\n");
else if (y == 3)
    printf("y vaut 3 mais cela ne sera jamais affiché.\n");
else
    printf("Ni l'un ni l'autre.\n");
    x = -1; // toujours évalué
```

Quiz: `if ... else`

Quiz: `if ... else`

Les variables (1/2)

Variables et portée

- Une variable est un identifiant, qui peut être liée à une valeur (un expression).
- Une variable a une **portée** qui définit où elle est *visible* (où elle peut être accédée).
- La portée est **globale** ou **locale**.
- Une variable est **globale** est accessible à tout endroit d'un programme et doit être déclarée en dehors de toute fonction.
- Une variable est **locale** lorsqu'elle est déclarée dans un **bloc**, `{...}`.
- Une variable est dans la portée **après** avoir été déclarée.

Les variables (2/2)

Exemple

```
int bar() { // x, y pas visibles ici, max oui }

int foo() {
    int x = 1; // x est locale à foo
    {
        // x est visible ici, y pas encore
        int y = 2;
        bar(); // ni x ni y sont visible dans bar()
    } // y est détruite à la sortie du bloc
} // x est à la sortie de foo

float max; // variable globale accessible partout

int main() {
    int z; // locale, à main
} // z est détruite ici, max aussi
```

Quiz: compile ou compile pas?

Quiz: compile ou compile pas

Entrées/sorties: `printf()` (1/2)

Généralités

- La fonction `printf()` permet d'afficher du texte sur le terminal:

```
int printf(const char *format, ...);
```

- Nombre d'arguments variables.
- `format` est le texte, ainsi que le format (type) des variables à afficher.
- Les arguments suivants sont les expressions à afficher.

Entrées/sorties: `printf()` (2/2)

Exemple

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

int main() {
    printf("Hello world.\n");
    int val = 1;
    printf("Hello world %d time.\n", val);
    printf("%f squared is equal to %f.\n", 2.5, 2.5*2.5);
    return EXIT_SUCCESS;
}
```

Entrées/sorties: `scanf()` (1/2)

Généralités

- La fonction `scanf()` permet de lire du texte formaté entré au clavier:

```
int scanf(const char *format, ...);
```

- `format` est le format des variables à lire (comme `printf()`).
- Les arguments suivants sont les variables où sont stockées les valeurs lues.

Entrées/sorties: `scanf()` (2/2)

Exemple

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

int main() {
    printf("Enter 3 numbers: \n");
    int i, j, k;
    scanf("%d %d %d", &i, &j, &k);
    printf("You entered: %d %d %d\n", i, j, k);

    return EXIT_SUCCESS;
}
```