

Module Algorithmie et programmation

Introduction au langage C

Pierre Künzli

Adapté des cours de Paul Albuquerque, Guido Bologna et Orestis Malaspinas

Le langage C, historique

- 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.

Le langage C, historique

- 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 2022).
- Langage compilé, statiquement (et faiblement) typé, procédural, portable, très efficace.
- Langage “bas niveau” (en 2022) : gestion 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, ...).

La simplicité de C ?

32 mots-clé et c'est tout

auto 'dou	bleint'	struct
break 'els	elong'	switch
case 'enu	mregistre	rtypedef'
char 'ext	ernreturn'	union
const 'flo	atshort'	unsigned
continue 'for	signed'	void
default 'got	sizeof'	volatile
do if	static	while

Déclaration et typage

Déclaration et typage

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

```
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
```


Les variables

Variables et portée

- Une variable est un identifiant, qui peut être liée à une valeur (une 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.

Exemple

```
float max; // variable globale accessible partout
int foo() {
    // max est visible ici
    float a = max; // valide
    // par contre les variables du main() ne sont pas visibles
}
int main() {
    // max est visible ici
    int x = 1; // x est locale à main
    {
        // x est visible ici, y pas encore
        // on peut par exemple pas faire x = y;
        int y = 2;
    } // y est détruite à la sortie du bloc
} // x est détruite à la sortie de main
```

Représentation des variables en mémoire

La mémoire

- La mémoire est :
 - ... un ensemble de bits,
 - ... accessible via des adresses,

bits	00110101	10010000	...	00110011
addr	2000	2001	...	4000

- ... gérée par le système d'exploitation.
- ... séparée en deux parties : **la pile** et **le tas**.

Une variable

- Une variable, type `a = valeur`, possède :
 - un type (`char`, `int`, ...),
 - un contenu (une séquence de bits qui encode valeur),
 - une adresse mémoire (accessible via `&a`),
 - une portée.

Représentation des variables en mémoire

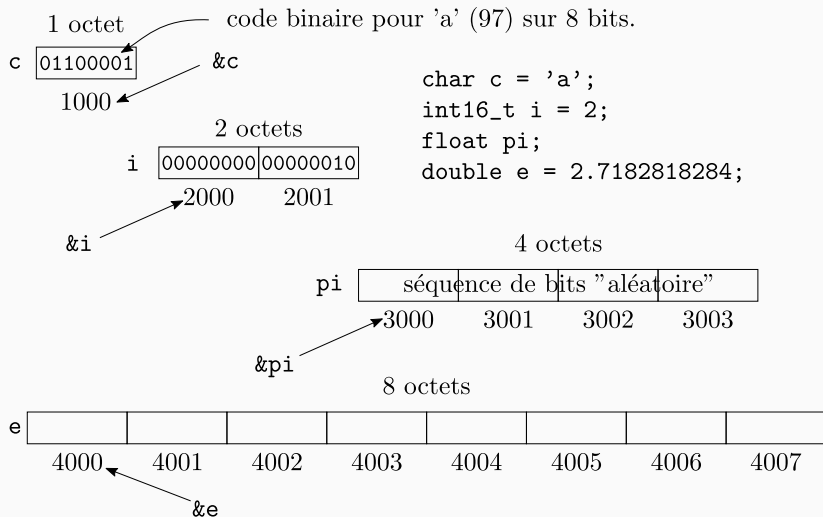


Fig. 1 : Les variables en mémoire.

Types de base

Numériques

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

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

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

Type	Signification
<code>int8_t</code> , <code>uint8_t</code> Entier s	igné/non-signé 8-bit
<code>int16_t</code> , <code>uint16_t</code> Entier s	igné/non-signé 16-bit
<code>int32_t</code> , <code>uint32_t</code> Entier s	igné/non-signé 32-bit
<code>int64_t</code> , <code>uint64_t</code> Entier s	igné/non-signé 64-bit

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

Void

Le type `void` est un type particulier, qui représente l'absence de donnée, de type ou un type indéterminé. On ne peut pas déclarer une variable de type `void`.

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';
```

Expressions et opérateurs

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 littéral
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'
```

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

Opérateurs logiques

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

Opérateurs arithmétiques

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

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.

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**

Syntaxe

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

```
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");  
}
```


Pièges

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

Structures de contrôle : switch .. case

Structures de contrôle : switch .. case

```
switch (expression) {  
    case constant-expression:  
        instructions;  
        break; // optionnel  
    case constant-expression:  
        instructions;  
        break; // optionnel  
    // ...  
    default:  
        instructions;  
}
```

Que se passe-t-il si break est absent ?

Structures de contrôle : switch .. case

```
int x = 0;
switch (x) {
    case 0:
    case 1:
        printf("0 ou 1\n");
        break;
    case 2:
        printf("2\n");
        break;
    default:
        printf("autre\n");
}
```

Dangereux, mais c'est un moyen d'avoir un “ou” logique dans un case.

Structures de contrôle : while

La boucle while

```
while (condition) {  
    instructions;  
}
```

ou

```
do {  
    instructions;  
} while (condition);
```

La boucle while, un exemple

```
int sum = 0; // syntaxe C99  
while (sum < 10) {  
    sum += 1;  
}  
do {  
    sum += 10;  
} while (sum < 100)
```

Structures de contrôle : for

La boucle for

```
for (expression1; expression2; expression3) {  
    instructions;  
}
```

La boucle for

```
int sum = 0; // syntaxe C99
for (int i = 0; i < 10; i++) {
    sum += i;
}

for (int i = 0; i != 1; i = rand() % 4) { // ésotérique
    printf("C'est plus ésotérique.\n");
}
```

Structures de contrôle :
continue, break

Structures de contrôle : continue, break

- continue saute à la prochaine itération d'une boucle.

```
int i = 0;
while (i < 10) {
    if (i == 3) {
        continue;
    }
    printf("%d\n", i);
    i += 1;
}
```

- break quitte le bloc itératif courant d'une boucle.

```
for (int i = 0; i < 10; i++) {
    if (i == 3) {
        break;
    }
    printf("%d\n", i);
}
```

Exercice : la factorielle

Exercice : la factorielle

Écrire un programme qui calcule la factorielle d'un nombre

$$N! = 1 \cdot 2 \cdot \dots \cdot (N - 1) \cdot N.$$

Ecrire un pseudo-code

Ecrire un pseudo-code

```
int factorielle(int n) {  
    i = 1;  
    fact = 1;  
    tant que i <= n {  
        fact *= i;  
        i += 1;  
    }  
}
```


Ecrire un code en C

```
#include <stdio.h>
int main() {
    int nb = 10;
    int fact = 1;
    int iter = 1;
    while (iter <= nb) {
        fact *= iter;
        iter++;
    }
}
```

Ecrire un code en C

```
#include <stdio.h>
int main() {
    int nb = 10;
    int fact = 1;
    int iter = 1;
    while (iter <= nb) {
        fact *= iter;
        iter++;
    }
}
```

Comment améliorer ce code ? A faire en exercice.

Entrées/sorties : printf()

- 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.

Exemple

```
#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;
}
```

Exemple

```
#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;
}
```

Remarque : ici la fonction `main` retourne `int` au lieu de `void` pour pouvoir retourner un code d'erreur.

Entrées/sorties : scanf()

- 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.

Exemple

```
#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;
}
```

Les fonctions

Les fonctions

- Les parties indépendantes d'un programme.
- Permettent de modulariser et compartimenter le code.
- Syntaxe :

```
type identificateur(paramètres) {  
    // variables optionnelles  
    instructions;  
    // type expression == type  
    return expression;  
}
```

Exemple

```
int max(int a, int b) {  
    if (a > b) {  
        return a;  
    } else {  
        return b;  
    }  
}  
  
int main() {  
    int c = max(4, 5);  
}
```

Les fonctions

- Il existe un type `void`, “sans type”, en C.
- Il peut être utilisé pour signifier qu’une fonction ne retourne rien, ou qu’elle n’a pas d’arguments.
- `return` utilisé pour sortir de la fonction.
- Exemple :

```
void show_text(void) { // second void optionnel
    printf("Aucun argument et pas de retour.\n");
    return; // optionnel
}
```

```
void show_text_again() { // c'est pareil
    printf("Aucun argument et pas de retour.\n");
}
```

Prototypes de fonctions

- Le prototype donne la **signature** de la fonction, avant qu'on connaisse son implémentation.
- L'appel d'une fonction doit être fait **après** la déclaration du prototype.

```
int max(int a, int b); // prototype

int max(int a, int b) { // implémentation
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

Arguments de fonctions

- Les arguments d'une fonction sont toujours passés **par copie**.
- Les arguments d'une fonction ne peuvent **jamais** être modifiés.

```
void set_to_two(int a) { // a: nouvelle variable
    // valeur de a est une copie de x
    // lorsque la fonction est appelée, ici -1

    a = 2; // la valeur de a est fixée à 2
} // a est détruite

int main() {
    int x = -1;
    set_to_two(x); // -1 est passé en argument
    // x vaudra toujours -1 ici
}
```


Arguments de fonctions : pointeurs

- Pour modifier une variable, il faut passer son **adresse mémoire**.
- L'adresse d'une variable, `x`, est accédée par `&x`.
- Un **pointeur** vers une variable entière a le type, `int *x`.
- La syntaxe `*x` sert à **déréférencer** le pointeur (à accéder à la mémoire pointée).

Exemple

```
void set_to_two(int *a) {  
    // a contient une copie de l'adresse de la  
    // variable passée en argument  
  
    *a = 2; // on accède à la valeur pointée par a,  
            // et on lui assigne 2  
} // le pointeur est détruit, pas la valeur pointée  
int main() {  
    int x = -1;  
    set_to_two(&x); // l'adresse de x est passée  
    // x vaudra 2 ici  
}
```