Chapitre 2 : Les bases du Langage C

Construction et maintenance de logiciels

Guy Francoeur

basé sur du matériel pédagogique d'Alexandre Blondin Massé, professeur

UQÀM | Département d'informatique

Table des matières

Table des matières

Historique

- **Années 70**. Naissance du langage C, créé par **Ritchie** et **Kernighan**.
- Origine liée au système Unix (90% écrit en C).
- **1978** Publication du livre **The C Programming Language**, par Kernighan et Ritchie. On appelle cette première version le **C K & R**.
- **1983** ANSI forme un comité dont l'objectif est la **normalisation** du langage C.
- **1989** Apparition de la norme **ANSI-C**. Cette seconde version est appelée **C ANSI**.

Utilisation du langage C

- Langage d'implémentation de certains **systèmes d'exploitation** (**Unix** et dérivés) :
- Approprié pour la construction de **bibliothèques**;
- Utilisé pour la construction de logiciels très connus;
- Utilisé construction de logiciels dit portable;
- Utilisé chez Microsoft, Oracle, Apple.

Caractéristiques du langage C (1/2)

- Langage **structuré**, conçu pour traiter les tâches d'un programme en les mettant dans des **blocs**;
- Il produit des programmes **efficaces** : il possède les mêmes possibilités de contrôle de la machine que le langage **assembleur** et il génère un code **compact et rapide**;
- C'est un langage **déclaratif**. Normalement, tout objet C doit être **déclaré** avant d'être utilisé. S'il ne l'est pas, il est considéré comme étant du type **entier**;
- La syntaxe est très **flexible** : la mise en page (indentation, espacement) est très libre, ce qui doit être exploité **adéquatement** pour rendre les programmes **lisibles**.

Caractéristiques du langage C (2/2)

- Le langage C est **modulaire**. On peut donc découper une application (logiciel) en modules qui peuvent être compilés **séparément**. Il est également possible de regrouper des programmes en **librairie**;
- Il est **flexible**. Peu de **vérifications** et d'**interdits**, hormis la syntaxe. Malheureusement, dans certains cas, ceci peut entraîner des problèmes de **lisibilités**;
- C'est un langage **transportable**. Les **entrées/sorties**, les **fonctions mathématiques** et les fonctions de **manipulation de chaînes de caractères** sont réunies dans des librairies, parfois **externes** au langage et sont fournies par le système d'exploitation (dans le cas des entrées/sorties par exemple).

Hello sans arguments

```c
// hello.c
#include <stdio.h> //printf()

int main() {
  printf("Bonjour le monde.");

  return 0;
}
```

Table des matières

Types de base

- **Types numériques** :

Type	Taille
char (signé ou pas)	1 octet
short (signé ou pas)	2 octets
int (signé ou pas)	2 ou 4 octets
long (signé ou pas)	4 ou 8 octets
float	4 octets
double	8 octets
long double	16 octets
___int128 signé ou pas	16 octets
int * ou char *	8 octets

- La grandeur du type (en octets) peut varier en fonction du processeur (ARM/intel/486)
- **Type vide** : void. Définit le type d'une fonction sans valeur de **retour** ou la valeur nulle pour les **pointeurs**.

Booléens

- Pas de type **booléen natif**.
- En C, la valeur 0 est considérée comme faux alors que toutes les autres valeurs **entières** sont considérées comme vrai.
- Depuis le standard **C99**, il existe la librairie stdbool.h qui définit les constantes true et false ainsi que le type bool.

```
1   #include <stdbool.h>
2   int main() {
3     bool valide = true;
4
5     if (valide) printf("OK\n");
6     else printf("ERREUR\n");
7
8     valide = !valide;
9
10    return 0;
11  }
```

Déclaration des variables

Une **variable**

- doit être **déclarée** avant son **utilisation**, en début de bloc;
- est **visible** seulement dans le **bloc** où elle est déclarée;
- peut/devrait/doit être **initialisée** lors de la déclaration;
- **non initialisée** a un comportement **imprévisible**, puisque la valeur qu'elle contient est non déterministe;
- pendra la valeur qui est à l'adresse ou elle est déclaré;

```
1  //exemple de déclarations
2  char c = 'e';
3  int a, b = 4;
4  float x, y;
5  unsigned int d = fact(10);
```

Constantes - Académique seulement

- À l'aide de l'instruction #define :

```
1 #define PI 3.141592654
```

- Avec le mot réservé const :

```
1 const float PI = 3.141592654;
2 const int x = 4;
3 // Ne fonctionne pas pour définir les dimensions de
    tableaux
```

- À l'aide d'un **type énumératif** :

```
1 enum WEEKEND { Samedi = 7, Dimanche = 1 };
2 // enum : Seulement des constantes entières
```

- Il est préférable de déclarer des **constantes** plutôt que des **valeurs (magiques) directement** dans les programmes.

Affectation - différentes bases

- le suffixe u ou U pour indiquer une valeur **non signée**;
- le suffixe l ou L pour indiquer une valeur **longue**.
- le préfixe 0 indique une **valeur octale**; Par exemple, 064 dénote le nombre décimal $\mathbf{6} \times 8^1 + \mathbf{4} \times 8^0 = 52$.
- le préfixe 0x indique une **valeur hexadécimale**; Par exemple, 0X34 dénote ce même décimal $\mathbf{3} \times 16^1 + \mathbf{4} \times 16^0 = 52$.
- Un **caractère**, entre apostrophes ', est un **nombre**; Par exemple, '4' correspond au décimal 52 (code ASCII).

```
1  char i = 52, j = 064, k = 0X34, l = '4';
2  printf("%d %d %d %d\n", i, j, k, l);
3  // affiche : 52 52 52 52
```

Exemple

```
1  //programme Ouille
2  #include <stdio.h>
3  int main() {
4    int a = 090;
5    printf("%d", a); // affiche : ?
6    return 0;
7  }
```

```
1  //programme oh boboy
2  #include <stdio.h>
3  int main() {
4    int a = 010;
5    printf("%d", a); // affiche : ?
6    return 0;
7  }
```

Caractères spéciaux

Quelques caractères utiles :

- \n, le caractère de **fin de ligne**;
- \t, le caractère de **tabulation**;
- \\, le caractère **“backslash”**;
- \', l'**apostrophe**;
- \", les **guillemets**.

Table des matières

Instruction for

```
for (<initialisation>; <condition>; <incrementation>)
    {
    <instruction 1>
    <instruction 2>
    ...
    <instruction n>
}
```

- <initialisation> est évaluée **une seule fois**, **avant** l'exécution de la boucle.
- <condition> est évaluée lors de **chaque passage**, **avant** d'exécuter les instructions dans le corps de la boucle;
- <incrémentation> est évaluée lors de **chaque passage**, **après** avoir exécuté les instructions dans le corps de la boucle.

Différence entre les standards du C

- **Attention**, on ne peut **déclarer le type** de l'itérateur dans l'initialisation qu'avec le standard **C99 et +**.
- Par exemple, le fragment de code suivant ne **compile pas** avec le standard **ANSI/C90** :

```
1  for (int i = 0; i < 10; ++i) {
2      printf("Valeur %d du tableau : %d", i, tab[i]);
3  }
```

- en C90, il faut plutôt écrire :

```
1  int i;
2  for (i = 0; i < 10; ++i) {
3      printf("Valeur %d du tableau : %d", i, tab[i]);
4  }
```

Instructions if, else if and else

```
if (<condition>) {
    <instruction>
}

if (<condition>) {
    <instruction 1>
} else {
    <instruction 2>
}

if (<condition 1>) {
    <instruction 1>
} else if (<condition 2>) {
    <instruction 2>
}
```

Blocs

- Un **bloc** est un ensemble d'instructions délimitées par des **accolades**;
- Les accolades sont **facultatives** dans les structures **conditionnelles** s'il n'y a qu'**une seule instruction**;
- Ainsi, les fragments suivants sont **équivalents** :

1.

```
1       if (!valide) printf("ERREUR");
```

2.

```
1       if (!valide)
2           printf("ERREUR");
```

3.

```
1       if (!valide) {
2           printf("ERREUR");
3       }
```

Instruction switch

```
switch (<variable>) {
    case <valeur 1> : <instruction 1>
    case <valeur 2> : <instruction 2>
    ...
    case <valeur n> : <instruction n>
    default : <instruction n + 1>
}
```

- Les instructions case sont parcourues **séquentiellement**, jusqu'à ce qu'il y ait une correspondance.
- Si c'est le cas, l'instruction correspondante est exécutée, ainsi que toutes les instructions suivantes, tant que le mot réservé break n'est pas rencontré.
- L'**ordre** d'énumération n'est pas important si on trouve une instruction break dans chaque cas.
- Le cas default est **optionnel**.

Boucles while et do-while

Syntaxe :

```
while (<condition>) {
    <instruction 1>
    <instruction 2>
    ...
    <instruction n>
}

do {
    <instruction 1>
    <instruction 2>
    ...
    <instruction n>
} while (<condition>);
```

Instruction break et continue

- break permet de **sortir** de la boucle;
- continue permet de **passer** immédiatement à l'itération **suivante**;
- Il est généralement à éviter d'utiliser **plusieurs** instructions break et continue dans la même boucle.

Table des matières

Opérateurs arithmétiques

Opérateur	Opération	Utilisation
+	addition	x + y
-	soustraction	x - y
*	multiplication	x * y
/	division	x / y
%	modulo	x % y

Lorsque les deux opérandes de la division sont des types **entiers**, alors la division est **entière** également.

Représentation interne

Représentation par le **complément à deux** :

	signe							
127 =	0	1	1	1	1	1	1	1
2 =	0	0	0	0	0	0	1	0
1 =	0	0	0	0	0	0	0	1
0 =	0	0	0	0	0	0	0	0
-1 =	1	1	1	1	1	1	1	1
-2 =	1	1	1	1	1	1	1	0
-127 =	1	0	0	0	0	0	0	1
-128 =	1	0	0	0	0	0	0	0

S'il y a **débordement**, il n'y a pas d'**erreur** :

```
1 signed char c = 127, c1 = c + 1;
2 printf("%d %d\n", c, c1);
3 // Affiche 127 -128
```

Opérateurs de comparaison et logiques

Opérateurs de **comparaison**

Opérateur	Opération	Utilisation
==	égalité	x == y
!=	inégalité	x != y
>	stricte supériorité	x > y
>=	supériorité	x >= y
<	stricte infériorité	x < y
<=	infériorité	x <= y

Opérateurs **logiques**

Opérateur	Opération	Utilisation
!	négation	!x
&&	et	x && y
||	ou	x || y

Évaluation **paresseuse** pour && et ||.

Opérateurs d'affectation et de séquençage

- =, +=, -=, *=, /=, %=;

```
1 int x = 1, y, z, t;
2 t = y = x;      // Equivaut à t = (y = x)
3 x *= y + x;     // Equivaut à x = x * (y + x)
```

- Incrémentation et décrémentation : ++ et −−;

```
1 int x = 1, y, z;
2 y = x++;        // y = 1, x = 2
3 z = ++x;        // z = 3, x = 3
```

- Opération de **séquençage** : évalue d'abord les expressions et retourne la dernière.

```
1 int a = 1, b;
2 b = (a++, a + 2);
3 printf("%d\n", b);
4 // Affiche 4
```

Opérateur ternaire

```
<condition> ? <instruction si vrai> : <instruction si faux>
```

- Très **utile** pour alléger le code;
- Très **utilisé**.

Quelles sont les valeurs affichées par le programme suivant ?

```c
#include <stdio.h>

int main() {
    int x = 1, y, z;
    y = (x-- == 0 ? 1 : 2);
    z = (++x == 1 ? 1 : 2);

    printf("%d %d\n", y, z);
    return 0;
}
```

Opérations bit à bit

Opérateur	Opération	Utilisation
&	et	x & y
|	ou	x | y
^	ou exclusif	x ^ y
~	not	x = ~n
<<	*shift* gauche	$x = x << 1$
>>	*shift* droit	$x = x >> 1$

Opérations bit à bit exemple

```c
// bitwise.c
#include <stdio.h>

int main() {
 int a = 9, b = 65; // x00001001, x01000001
 unsigned char c = 0;
 c = ~c;
 unsigned short d = 8;

 printf(" Bitwise AND Operator a&b = %d \n", a & b);
 printf(" Bitwise OR Operator a|b = %d \n", a | b);
 printf(" Bitwise EXCLUSIVE OR Operator a^b = %d \n", a ^ b);

 printf(" Bitwise NOT Operator ~c = %d \n", c);

 printf(" LEFT SHIFT Operator d<<1 = %d \n", d << 1);
 printf(" RIGHT SHIFT Operator d>>1 = %d \n", d >> 1);

 return 0;
}
```

Table des matières

Conversion en C

Les **conversions** (*cast*) implicite agissent selon la promotion suivante :

- *char* → *short* → *int* → *unsigned int* → *long* → *long long* → *unsigned long long* → *float* → *double* → *long double*

```c
//exo3.c
#include <stdio.h>

int main() {
  printf("%lu, ",sizeof(1 + 1L));
  printf("%lu, ",sizeof((float) 1 + 1.1));
  printf("%lu, ",sizeof((int) 1 + (long double)100))
      ;
  printf("%lu \n",sizeof((char) 1 + (short)100));
  // Affiche 8, 8, 16, 4
  return 0;
}
```

Conversions implicites

Attention aux conversions implicites entre types **signés** et **non signés**.

```
1  // exo4.c
2  #include <stdio.h>
3  int main() {
4      char x = -1, y = 20, v;
5      unsigned char z = 254;
6      unsigned short t;
7      unsigned short u;
8
9      t = x;
10     u = y;
11     v = z;
12     printf("%d %d %d\n", t, u, v);
13     // Affiche 65535 20 -2
14     return 0;
15 }
```

Conversions explicites

```c
//exo7.c
#include <stdio.h>
int main() {
    unsigned char x = 255;
    printf("%d\n", x);
    // Affiche 255
    printf("%d\n", (signed char)x);
    // Affiche -1
    int y = 3, z = 4;
    printf("%d %f\n", z / y, ((float)z) / y);
    // Affiche 1 1.333333
    return 0;
}
```

Priorité des opérateurs

Arité	Associativité	Par priorité décroissante
2	gauche, droite	( ), [ ]
2	gauche, droite	->, .
1	droite, gauche	!, ++, –, +, -, (int), *, &, sizeof
2	gauche, droite	*, /, %
2	gauche, droite	+, -
2	gauche, droite	<, <=, >, >=
2	gauche, droite	==, !=
2	gauche, droite	&&
2	gauche, droite	||
3	gauche, droite	? :
1	droite, gauche	=, +=, -=, *=, /=, %=
2	gauche, droite	,