Chapitre 2 : Introduction au langage C

Alexandre Blondin Massé

Département d'informatique
Université du Québec à Montréal

13 septembre 2016
Construction et maintenance de logiciels
INF3135

Table des matières

1. Introduction

2. Makefiles

3. Variables et constantes

4. Structures de contrôle

5. Opérateurs

6. Tableaux et chaînes de caractères

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) :
  - Près du **langage machine**;
  - Pointeurs **typés** mais **non contrôlés**;
- Adapté aux **petits programmes** et aux **bibliothèques** :
  - **Efficacité** du code généré;
  - Compilation **séparée**;
- Langage peu utilisé pour les applications de **grande envergure** :
  - Approche archaïque de la **modularité**;
  - Typage **laxiste**.

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 (1/2)

- Le langage C est **modulaire**. On peut donc découper une application 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 entraîne des problèmes de **lisibilités majeurs** et de **mauvaises habitudes de programmation**;
- C'est un langage **transportable**. Les **entrées/sorties**, **fonctions mathématiques** et fonctions de **manipulation de chaînes de caractères** sont réunies dans des librairies, parfois **externes** au langage (dans le cas des entrées/sorties par exemple).

Exemple de programme C

Fichier **exemple.c** :

```
#include <stdio.h>        /* directives au préprocesseur */
#define DEBUT -2
#define FIN 2
#define MSG "Programme de démonstration\n"

int carre(int x);         /* déclaration des fonctions */
int cube(int x);

int main() {              /* début du bloc principal */
    int i;                /* définition des variables locales */
    printf(MSG);
    for (i = DEBUT; i <= FIN; i++) {
        printf("%d carré: %d cube: %d\n", i, carre(i), cube(i));
    }
    return 0;
}                         /* fin du bloc principal */

int cube(int x) {         /* définition de la fonction cube */
    return x * carre(x);
}

int carre(int x) {        /* définition de la fonction carre */
    return x * x;
}
```

Compilation

1. **Édition** du programme source à l'aide d'un **éditeur de texte** ou d'un **environnement de développement**. L'extension du fichier est **.c**.
2. **Compilation** du programme source, traduction du langage C en **langage machine**. Le compilateur indique les **erreurs de syntaxe**, mais ignore les **fonctions** et les **bibliothèques** appelées par le programme. Le compilateur génère un fichier avec l'extension **.o**.
3. **Édition de liens**. Le code machine de différents fichiers **.o** est assemblé pour former un fichier **binaire**. Le résultat porte l'extension **.out** (sous Unix) ou **.exe** (sous Windows).
4. **Exécution du programme**. Soit en **ligne de commande** ou en **double-cliquant** sur l'icône du fichier binaire.

Exemple de compilation (1/2)

- Reprenons le fichier **exemple.c**
- On peut directement le **compiler** en exécutable :

  ```
  $ gcc exemple.c
  ```

  ce qui produit le fichier **a.out**.
- Puis ensuite, on l'**exécute** :

  ```
  $ ./a.out
  Programme de démonstration
  -2 carré: 4 cube: -8
  -1 carré: 1 cube: -1
  0 carré: 0 cube: 0
  1 carré: 1 cube: 1
  2 carré: 4 cube: 8
  ```

Exemple de compilation (2/2)

- En général, on compile en **deux étapes**;
- D'abord, de **.c** vers **.o** :

  ```
  $ gcc -c exemple.c
  ```

  ce qui produit le fichier **compilé** (objet) **exemple.o**.
- Puis ensuite, la commande

  ```
  $ gcc -o exemple exemple.o
  ```

  produit un fichier **exécutable** nommé **exemple**.
- Il s'exécute simplement en entrant

  ```
  $ ./exemple
  ```

Table des matières

1. Introduction

2. Makefiles

3. Variables et constantes

4. Structures de contrôle

5. Opérateurs

6. Tableaux et chaînes de caractères

Gestion de la compilation

- On a vu un peu plus tôt les **deux étapes** pour créer un **exécutable en C** :
  - On compile le fichier **.c** en un fichier **.o**;

    ```
    $ gcc -c exemple.c
    ```

  - On lie les fichiers **.o** en un seul fichier **exécutable**.

    ```
    $ gcc -o exemple exemple.o
    ```

- **Problème :** Il est **long** de relancer la compilation **chaque fois** qu'on apporte une modification au fichier **source**.
- **Solution :** Utilisation d'un **Makefile**.

Makefiles

- Existent depuis la fin des **années ’70**.
- Gèrent les **dépendances** entre les différentes composantes d’un programme;
- Automatisent la **compilation** en **minimisant** le nombre d’étapes;
- Malgré qu’ils soient **archaïques**, ils sont encore **très utilisés** (et le seront sans doute pour **très longtemps** encore);
- Certaines **limitations** des Makefiles sont corrigées par des outils comme **Autoconf** et **CMake**.

Exemple

- Reprenons notre fichier **exemple.c**
- Un **Makefile** minimal pour ce fichier serait le suivant :

```
exemple: exemple.o
	gcc -o exemple exemple.o

exemple.o: exemple.c
	gcc -c exemple.c
```

- La **syntaxe** est de la forme

```
<cible>: <dépendances>
<tab><commande>
```

(le caractère **`<tab>`** est très important !)

Exécution d'un Makefile

- Pour utiliser un **Makefile**, il suffit de taper

```
make
```

- On obtient alors

```
gcc -c exemple.c
gcc -o exemple exemple.o
```

  et les fichiers **.o** et l'**exécutable** sont produits.

- **Astuce :** Comme Vim interagit **directement** avec le terminal, je m'assure que les caractères **mm** déclenchent la commande **make**.

Table des matières

1. Introduction

2. Makefiles

3. Variables et constantes

4. Structures de contrôle

5. Opérateurs

6. Tableaux et chaînes de caractères

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 octets
`float`	4 octets
`double`	8 octets
`long double`	16 octets

- **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`**.

```
#include <stdbool.h>

...

bool valide = true;
if (valide) {
    printf("OK");
} else {
    printf("ERREUR");
}
valide = !valide;
```

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 être **initialisée** lors de la déclaration;
- **non initialisée** a un comportement **imprévisible**, puisque la valeur qu'elle contient peut être **quelconque**;

```
char c = 'e';
int a, b = 4;
float x, y;
unsigned int d = fact(10);
```

Constantes

- À l'aide de l'instruction **#define** :

```
#define PI 3.141592654
```

- Avec le mot réservé **const** :

```
const float PI = 3.141592654;
// Ne fonctionne pas pour les dimensions des
    tableaux
```

- À l'aide d'un **type énumératif** :

```
enum {
    MAX_SIZE = 34
};
// Seulement pour les constantes entières
```

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

Notation

- 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).

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

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

1. Introduction

2. Makefiles

3. Variables et constantes

4. Structures de contrôle

5. Opérateurs

6. Tableaux et chaînes de caractères

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 avec Java

- **Attention !** contrairement à Java et C++, on ne peut **déclarer le type** de l'itérateur dans l'initialisation, il faut le faire avant.

- Par exemple, on **ne peut pas** écrire :

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

- Il faut **plutôt** écrire

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

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

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

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

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>);
```

Quelle est la **différence** entre ces deux structures de boucles ?

Instruction break et continue

- **break** permet de **sortir** de la boucle;
- **continue** permet de **passer** immédiatement à l'itération **suivante**.

Le programme suivant termine-t-il ? Si oui, quelles valeurs sont affichées ?

```
#include <stdio.h>
#include <stdbool.h>

int main() {
    int tableau[] = {0,4,5,3,6,1,2};
    int i = 0;
    int trouve = false;
    while (!trouve && i < 7) {
        if (tableau[i] == 2) {
            trouve = true;
            break;
        } else if (tableau[i] == 4) {
            i = i - 1;
            continue;
        }
        i = i + 2;
    }
    printf("%d %d\n", i, trouve);
}
```

Table des matières

1. Introduction

2. Makefiles

3. Variables et constantes

4. Structures de contrôle

5. Opérateurs

6. Tableaux et chaînes de caractères

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** :

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

Conversions implicites

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

```
#include <stdio.h>

int main() {
    char x = -1, y = 20, v;
    unsigned char z = 254;
    unsigned short t;
    unsigned short u;

    t = x;
    u = y;
    v = z;
    printf("%d %d %d\n", t, u, v);
    // Affiche 65535 20 -2
}
```

Conversions

- Si un des opérandes est **`long double`**, alors le résultat est également **`long double`**.
- Sinon, si un des opérandes est **`double`**, alors le résultat est également **`double`**.
- Sinon, si un des opérandes est **`float`**, alors le résultat est également **`float`**.
- Sinon, il y a promotion vers le type **`int`** et **`unsigned`**.
- **Bref**, évitez de mélanger les types dans une même **opération** ou montrez les conversions de façon **explicite**.

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**

<table>
<tr><th>Opérateur</th><th>Opération</th><th>Utilisation</th></tr>
<tr><td>!</td><td>négation</td><td>!x</td></tr>
<tr><td>&&</td><td>et</td><td>x && y</td></tr>
<tr><td>||</td><td>ou</td><td>x || y</td></tr>
</table>

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

Opérateurs d'affectation et de séquençage

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

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

- Incrémentation et décrémentation : **++** et **--**;

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

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

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

Opérations bit à bit

Opérateur	Opération	Utilisation
`&`	et	`x & y`
`|`	ou	`x | y`
`^`	ou exclusif	`x ^ y`

- Utile pour simuler les opérations **ensemblistes** de façon très compacte;
- Par exemple, les ensembles $\{3, 1, 0\}$ et $\{3, 2\}$ peuvent être représentés par les nombres $x = (1011)_2$ et $y = (1100)_2$ respectivement;
- Les opérateurs **`&`**, **`|`** et **`^`** correspondent respectivement aux opérations **réunion**, **intersection** et **différence symétrique** sur les ensembles.

Conversion de types

```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
}
```

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	`,`

Table des matières

1. Introduction

2. Makefiles

3. Variables et constantes

4. Structures de contrôle

5. Opérateurs

6. Tableaux et chaînes de caractères

Tableaux

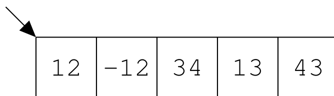
- Collection de données de **même type**;
- **Déclaration** :

```
int donnees[10];
// Réserve 10 "cases" de type "int" en mémoire
int donnees[taille];
// Non ! La taille doit être connue à la compilation
```

- **Définition** et **initialisation** :

```
int toto[] = {12,-12,34,13,43};
```

- Stockées de façon **contiguë** en mémoire;



Accès

- À l'aide de l'opérateur **[ ]** :

```
#include <stdio.h>

int main() {
    int donnees[] = {12,-12,34,13,43};
    int a, b;
    a = donnees[2];
    b = donnees[5];

    printf("%d %d\n", a, b);
        /* que vaut b ? */
}
```

- Le **premier** élément est à l'indice **0**;
- S'il y a **dépassement** de borne, **aucune erreur** ou un **avertissement** (**warning**).
- Source fréquente de **segfault**.

Chaînes de caractères

- Les **chaînes de caractères** sont représentées par des **tableaux de caractères**;
- Les chaînes constantes sont délimitées par les symboles `" "`. Les deux déclarations suivantes sont **équivalentes** :

```
char chaine[] = "tomate";
char chaine[] = {'t','o','m','a','t','e','\0'};
```

- Termine par le caractère `\0`;
  - Longueur de la chaîne `"tomate"` : **6**;
  - Taille du tableau représentant la chaîne `"tomate"` : **7**.

Exercice

Écrivez une fonction dont la signature est

```
unsigned int longueur(char[] chaine);
```

qui calcule la **longueur** d'une chaîne de caractères.