Chapitre 3 : Les bases du C, partie 2

Construction et maintenance de logiciels

Guy Francoeur

basé sur les travaux d'Alexandre Blondin Massé, professeur

5 septembre 2019

UQÀM | Département d'informatique

Table des matières

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

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

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

Quelles sont les valeurs affichées par le programme suivant ?

```
1  #include <stdio.h>
2
3  int main() {
4      int x = 1, y, z;
5      y = (x-- == 0 ? 1 : 2);
6      z = (++x == 1 ? 1 : 2);
7
8      printf("%d %d\n", y, z);
9      return 0;
10 }
```

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;
}
```

Conversion en C

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

- $char \rightarrow short \rightarrow int \rightarrow unsigned\ int \rightarrow long \rightarrow long\ long \rightarrow unsigned\ long\ long \rightarrow float \rightarrow double \rightarrow long\ double$

```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**.

```c
// exo4.c
#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
    return 0;
}
```

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	,

Table des matières

Tableaux

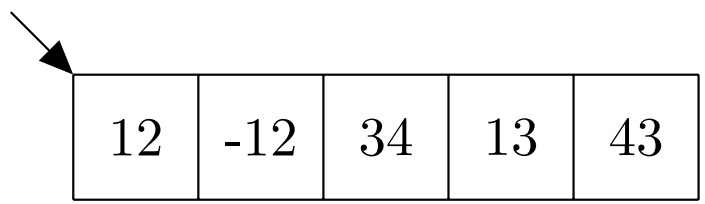
- Collection de données de **même type**;
- **Déclaration** :

```
1 int donnees[10];
2 // Réserve 10 "cases" de type "int" en mémoire
3 int donnees[taille];
4 // Seulement avec C99 et allocation sur la pile
```

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

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

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



Accès

- À l'aide de l'opérateur [ ] :

```c
// exo8.c
#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 a et b ? */
    return 0;
}
```

- 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 de guillemets " ".
- Les deux déclarations suivantes sont **équivalentes** :

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

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

Table des matières

Définition

- Une adresse est un emplacement **précis** en mémoire.
- Un pointeur est une variable qui contient l'adresse d'une autre variable en mémoire;
- On déclare un pointeur en utilisant le symbole *****;
- L'opérateur & retourne l'adresse d'une variable en mémoire.

Exemple

```c
//pointeur1.c
#include <stdio.h>

int main() {
  int *p; //un pointeur vers un entier

  printf("La variable p pointe vers l'adresse %p.\n", p);

  return 0;
}
```

Exemple

```c
// pointeur2.c
#include <stdio.h>

int main() {
    int *pi, x = 104;
    pi = &x;
    printf("x vaut %d et se trouve à l'adresse %p\n", x, &x);
    printf("pi vaut %p et pointe sur la valeur %d\n", pi, *pi);

    *pi = 350;
    printf("x vaut %d et se trouve à l'adresse %p\n", x, &x);
    printf("pi vaut %p et pointe sur la valeur %d\n", pi, *pi);
    return 0;
}
```

Affiche :

x vaut 104 et se trouve à l'adresse 0x7fff5fbff73c
pi vaut 0x7fff5fbff73c et pointe sur la valeur 104
x vaut 350 et se trouve à l'adresse 0x7fff5fbff73c
pi vaut 0x7fff5fbff73c et pointe sur la valeur 350

Affectation

- Impossible d'affecter directement une **adresse** à un pointeur :

```
1 int *pi;
2 pi = 0xdff1;          /* interdit */
```

- Par contre, avec une conversion **explicite**, c'est possible :

```
1 int *pi;
2 pi = (int*)0xdff1; /* permis, mais a eviter */
```

- On peut aussi utiliser une conversion pour associer une **même adresse** à des pointeurs de **types différents** :

```
1 int *pi;
2 char *pc;
3 pi = (int*)0xdff1;
4 pc = (char*)pi;
```

Lien entre tableaux et pointeurs

- Un tableau d'éléments de type t peut être vu comme un **pointeur constant** vers des valeurs de type t;
- **Exemple :** int a[3] définit un pointeur a vers des entiers;
- De plus, a pointe vers le **premier** élément du tableau :

```
1 //pointeur3.c
2 #include <stdio.h>
3 int main() {
4     int a[3] = {1,2,3}, *pi;
5     pi = a;              /* initialisation de pi */
6     printf("%p, %p,  %d,  %d,  %d,  %d\n",
7                a, pi, a[0], a[1], a[2], *pi);
8     return 0;
9 }
```

- **Affiche** : 0x7fff5fbff720 0x7fff5fbff720 1 2 3 1
- pi = a est valide, mais a = pi n'est pas **valide**.

Un extra sur les pointeurs

```c
//pointeur5.c
#include <stdio.h>

int main() {
  int a;
  int *b;
  int *c=NULL;

  printf("%p, %d\n", a, a);

  *b = 100000;
  printf("%p, %d\n", b, *b);

  *c = 10000000;
  printf("%p, %d\n", c, *c);

  return 0;
}
```

- Que donne le programme?

Un extra sur les pointeurs

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

struct une_s {
  unsigned long a;
  unsigned long b;
};

int main(void) {

  int a[3]={0,1,2};
  struct une_s b;
  unsigned __int128 c;

  int *a2=a;
  struct une_s *b2=&b;
  unsigned __int128 *c2;
//sizeof:
  printf("var a %lu, pointeur a2 %lu\n",sizeof a, sizeof a2);
  printf("var b %lu, pointeur b2 %lu\n",sizeof b, sizeof b2);
  printf("var c %lu, pointeur c2 %lu\n",sizeof c, sizeof c2);

  return 0;
}
```

Opération sur les pointeurs

- Considérons un tableau tab de n éléments. Alors
  - tab correspond à l'**adresse** de tab[0];
  - tab + 1 correspond à l'**adresse** de tab[1];
  - ...
  - tab + n - 1 correspond à l'**adresse** de tab[n - 1];
- On peut calculer la **différence** entre deux pointeurs de même type;
- De la même façon, l'**incrémentation** et la **décrémentation** de pointeurs sont possibles;
- Finalement, deux pointeurs peuvent être **comparés**.

Exemple

```
1  // pointeur4.c
2  #include <stdio.h>
3
4  int main(int argc, char *argv[]) {
5      int a[3] = {1, -1, 2}, *pi, *pi2;
6      pi = a;
7      pi2 = &a[2];
8      printf("%ld ", pi2 - pi);
9      printf("%d ", *(--pi2));
10     printf("%d\n", *(pi + 1));
11     if (pi + 1 == pi2)
12         printf("pi et pi2 pointent vers la même case mé
               moire.\n");
13     return 0;
14 }
```

Affiche :

2 -1 -1

pi et pi2 pointent vers la même case mémoire.

Gestion de la mémoire

```
1   int *pi, tab[10];
```

- La déclaration d'un tableau **réserve** l'espace mémoire nécessaire pour stocker le tableau;
- La déclaration de *pi ne réserve **aucun espace** mémoire (sauf l'espace pour stocker une adresse);
- Les expressions

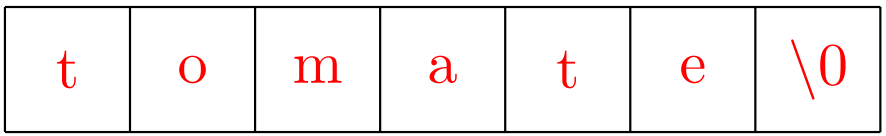
```
1 *pi = 6;
2 *(pi + 1) = 5;
```

sont **valides**, mais ne réservent pas l'espace mémoire correspondant. Autrement dit, le compilateur pourrait éventuellement **utiliser** cet espace.

Table des matières

Chaînes de caractères

- Une **chaîne de caractères** est représentée par un **tableau de caractères** terminant par le caractère \0;



- Des fonctions élémentaires sur les **caractères** se trouvent dans la bibliothèque ctype.h;
- D'autre part, la bibliothèque standard string.h fournit plusieurs fonctions permettant de **manipuler** les **chaînes de caractères**.

Arguments de la fonction main

- int main(int argc, char *argv[]);
- Le paramètre argv est un tableau de **pointeur vers des caractères**;
- argv[argc] == NULL est vrai;
- Quelle est la sortie affichée par le programme suivant avec la commande gcc ex8.c && ./a.out bonjour toi ?

```
1 // ex8.c
2 #include <stdio.h>
3 int main(int argc, char **argv) {
4   printf("argc est : %d\n", argc);
5   for (int i = 0; i < argc; ++i) {
6     printf("%s\n", argv[i]);
7   }
8   return 0;
9 }
```

La bibliothèque <ctype.h>

Fonction	Description
int isalpha(c)	Retourne une valeur non nulle si c est alphabétique, 0 sinon
int isupper(c)	Retourne une valeur non nulle si c est majuscule, 0 sinon
int islower(c)	Retourne une valeur non nulle si c est minuscule, 0 sinon
int isdigit(c)	Retourne une valeur non nulle si c est un chiffre, 0 sinon
int isalnum(c)	Retourne isalpha(c) || isdigit(c)
int isspace(c)	Retourne une valeur non nulle si c est un espace, un saut de ligne, un caractère de tabulation, etc.
char toupper(c)	Retourne la lettre majuscule correspondant à c
char tolower(c)	Retourne la lettre minuscule correspondant à c

Attention! Les fonctions toupper, tolower, etc. sont définies sur les **caractères** et non sur les **chaînes**.

La bibliothèque <string.h>

- La fonction unsigned int strlen(char *s) retourne la **longueur** d'une chaîne de caractères;
- La fonction int strcmp(char *s, char *t) retourne
  - une valeur **négative** si s < t selon l'ordre lexicographique;
  - une valeur **positive** si s > t;
  - la valeur **0** si s == t.
- Quelle est la différence entre s == t et strcmp(s, t) ?

Exemple

```
1  //ex1.c
2  #include <stdio.h>
3  #include <string.h>
4
5  int main() {
6      char s[] = "bonjour";
7      char t[] = "patate";
8
9      printf("Longueur de \"%s\" et \"%s\" : %lu, %lu\n",
10             s, t, strlen(s), strlen(t));
11     printf("strcmp(\"%s\", \"%s\") : %d\n", s, t,
12             strcmp(s, t));
13     return 0;
14 }
```

Sortie :

Longueur de "bonjour" et "patate" : 7, 6
strcmp("bonjour", "patate") : -14

Concaténation

- Les fonctions

```
1       char *strcat(char *s, const char *t);
2       char *strncat(char *s, const char *t, int n);
```

  permettent de **concaténer** deux chaînes de caractères;

- Plus précisément, la chaîne t est ajoutée à la fin de la chaîne s ainsi qu'un caractère \0;

- La chaîne s doit avoir une **capacité suffisante** pour contenir le résultat de la **concaténation**;

- Le paramètre n donne une limite **maximale** du nombre de caractères à concaténer.

Exercices

Quel résultat donne le code suivant ?

```c
// ex2.c
#include <stdio.h>
#include <string.h>

int main() {
    char s[10] = "Salut ";
    char t[] = "toi!";
    strcat(s, t);
    printf("%s\n", s);
    return 0;
}
```

Copie

- Les fonctions

```
1       char *strcpy(char *s, const char *t);
2       char *strncpy(char *s, const char *t, int n);
```

  permettent de **copier** une chaîne de caractère dans une autre;

- Dans ce cas, la chaîne t est copiée dans la chaîne s et un caractère \0 est ajouté à la fin;
- Comme pour strcat, la chaîne s doit avoir une **capacité suffisante** pour contenir la copie;
- Le paramètre n donne une limite **maximale** du nombre de caractères à copier;
- Quelle est la différence entre s = t et strcpy(s, t) ?

Segmentation d'une chaîne

- La fonction

```
1       char *strchr(char *s, int c);
```

retourne un **pointeur** vers la première occurrence du **caractère** c dans s.

- La fonction

```
1       char *strtok(char *s, const char *delim);
```

permet de **décomposer** une chaîne de caractères en **plus petites chaînes** délimitées par des caractèreres donnés;

- Le paramètre s correspond à la chaîne qu'on souhaite **segmenter**, alors que le paramètre delim donne la liste des caractères considérés comme **délimiteurs**;

- Très **utile** lorsqu'on souhaite extraire des données d'un **fichier texte**.

Décomposition avec champs vides

- La fonction **strtok** ne gère pas les cas où certains champs sont **vides**;
- Par exemple, si les données sont

```
1       "124:41:3::23:10"
```

il ne sera pas détecté qu'il y a une donnée **manquante** entre 3 et 23;

- La fonction

```
1       char *strsep(char **s, const char *delims);
```

résoud ce problème.

- **Attention !** Les fonctions **strtok** et **strsep** modifient la chaîne s.

Exemple (1/2)

```c
#include <stdio.h>
#include <string.h>
#define DELIMS ":"

int main() {
    char s[80];
    char *pc, *ps;

    strcpy(s, "124:41:3::23:10");
    printf("Avec strstok:\n");
    pc = strtok(s, DELIMS);
    while (pc != NULL) {
        printf("/%s/\n", pc);
        pc = strtok(NULL, DELIMS);
    }

    strcpy(s, "124:41:3::23:10");
    printf("Avec strsep:\n");
    ps = s;
    while ((pc = strsep(&ps, DELIMS)) != NULL) {
        printf("/%s/\n", pc);
    }
    return 0;
}
```

Exemple (2/2)

Résultat :

Avec strstok:
/124/
/41/
/3/
/23/
/10/
Avec strsep:
/124/
/41/
/3/
//
/23/
/10/

Table des matières

Utilité des fonctions

- Elles sont l'unité de **base** de programmation;
- Chaque fonction doit effectuer **une** tâche bien précise;
- Elles permettent d'appliquer la stratégie **diviser-pour-régner**;
- Elles sont à la base de la **réutilisation**;
- Elles favorisent la **maintenance** du code;
- Lorsqu'elles sont **appelées**, l'exécution du bloc appelant est suspendue jusqu'à ce que l'instruction return ou la **fin** de la fonction soit atteinte.

Arguments et paramètres

```
1 int max(int x, int y) {
2     if (x >= y) return x;
3     else return y;
4 }
5
6 printf(max(3, 4));
```

- Un **paramètre** d'une fonction est une **variable formelle** utilisée dans cette fonction (ex : x et y);
- Les fonctions ont **aucun**, **un** ou **plusieurs** paramètres d'**entrée**;
- Elles renvoient **au plus** un résultat en **sortie**.

Cas 1

Quelles sont les valeurs affichées par ce programme ?

```c
//error_swap.c
#include <stdio.h>
void echanger(int a, int b) {
    int z = a;
    a = b;
    b = z;
}

int main() {
    int a = 5, b = 6;
    echanger(a, b);
    printf("%d %d\n", a, b);
    return 0;
}
```

Cas 2

```
1  // swap.c
2  #include <stdio.h>
3  void echanger(int *a, int *b) {
4      int z = *a;
5      *a = *b;
6      *b = z;
7  }
8
9  int main() {
10     int a = 5, b = 6;
11     echanger(&a, &b);
12     printf("%d %d\n", a, b);
13     return 0;
14 }
```

Passage par valeur ou adresse

- Les types de base sont passés par **valeur**;
- Une **copie** de la valeur est transmise à la fonction;
- La **modification** de cette valeur à l'intérieur de la fonction **n'affecte pas** celle du bloc appelant.
- —
- La valeur n'est pas copié;
- La variable d'origine reçois les changement locaux.

Passage d'un tableau

- Les tableaux comme **paramètres** d'une fonction :

```
1 float produit_scalaire(float a[], float b[], int d);
```

- Un tableau est représenté par un **pointeur constant**;
- Il est donc passé par **adresse** lors de l'appel d'une fonction;
- Si la fonction n'est pas supposée **modifier** le tableau qu'elle reçoit en paramètre, il est convenable d'utiliser le mot réservé const.

```
1 float produit_scalaire(const float a[]
2                        ,const float b[]
3                        ,int d);
```

Exemple

```
1  //passage_tableau.c
2  #include <stdio.h>
3
4  float produit_scalaire(const float a[], const float b[],
5                         unsigned taille) {
6
7      float p = 0.0;
8      for (int i = 0; i < taille; ++i) {
9          p += a[i] * b[i];
10     }
11     return p;
12 }
13
14 int main() {
15     float u[] = {1.0, -2.0, 0.0};
16     float v[] = {-1.0, 1.0, 3.0};
17     printf("%f\n", produit_scalaire(u, v, 3));
18     return 0;
19 }
```

Affiche : -3.000000

Fonction retournant un tableau

- Une fonction ne peut pas **retourner** un **pointeur** créé dans la fonction, sauf s'il y a eu **allocation dynamique**;
- En particulier, on ne peut pas retourner un **tableau** comme résultat. Il faut plutôt que le tableau soit un des **arguments** de la fonction.

Exemple

```
1  #include <stdio.h>
2
3  int* initialise_tableau(unsigned taille) {
4      int tableau[taille];
5      int i;
6      for (i = 0; i < taille; ++i)
7          tableau[i] = 0;
8      return tableau;
9  }
10
11 int main() {
12     int *tableau;
13     tableau = initialise_tableau(4);
14     printf("%d\n", tableau[0]);
15     return 0;
16 }
```

Affiche :

exo18.c: In function ‘initialise_tableau’:
exo18.c:8: warning: function returns address of local variable 0

Déclaration et implémentation

- C'est une bonne pratique de déclarer les **prototypes** des fonctions au **début** du fichier où elles sont **définies** et/ou **utilisées**;
- Il n'est **pas nécessaire**, mais tout de même **encouragé** de donner un **nom** aux paramètres;
- Lors de la **définition**, le nom des variables est **obligatoire**.
- Contrairement à C++ et Java, la **surcharge** de fonctions est **interdite** :

```
1 int max(int x, int y);
2 int max(int x);
```

test.c:2: error: conflicting types for 'max'
test.c:1: error: previous declaration of 'max' was here

Autres variables

- Il est également possible de définir des variables **globales à plusieurs fichiers**, par l'intermédiaire du mot réservé extern;
- Par opposition aux **variables externes**, les variables **statiques**, déclarées à l'aide du mot réservé static, ont une portée limitée au **fichier** dans lequel elles sont déclarées.
- Les variables et fonctions globales sont **visibles** de leur déclaration jusqu'à la **fin du fichier** où elles sont définies;
- **Utilisables** jusqu'à la fin du programme;
- **Initialisées** à 0 par défaut;
- Les **fonctions** ont la même visibilité, accessibilité et durée de vie que les variables globales.

Variables et fonctions globales

Fichier main.c

```
1 #include <stdio.h>
2 #include "math.c"
3
4 int main() {
5     printf("PI = %f\n", PI);
6     printf("Le carre de %d
          est %d\n", 4, carre
          (4));
7     return 0;
8 }
```

Fichier math.c

```
1 const float PI =
      3.141592654;
2
3 int carre(int x) {
4     return x * x;
5 }
```

Affiche :

PI = 3.141593
Le carre de 4 est 16

Variables et fonctions statiques

```
1 static char tampon[TAILLE_TAMPON];
2 static int x;
3 static int factorielle(int n);
```

Les variables **locales statiques** sont

- associées à un espace de stockage **permanent**;
- existent même lorsque la fonction n'est pas **appelée**.

Les variables **globales statiques** et les **fonctions statiques** se comportent

- exactement comme les variables **globales** et les **fonctions**,
- à l'exception qu'elles ne peuvent être utilisées **en dehors du fichier** où elles sont définies.

Variables externes

- Permettent de définir des variables **globales à plusieurs fichiers**;
- Par défaut, toute variable **non locale** est considérée externe;
- Par l'intermédiaire du mot réservé extern;
- Uniquement pour une **déclaration** sans **initialisation**;
- Utiles lorsqu'on souhaite compiler les fichiers **séparément**;
- Ont une durée de vie aussi longue que celle du **programme**;
- Pour les **tableaux**, il n'est pas nécessaire d'indiquer une **taille**.

```
1 extern int x, a[];
```

La fonction main

- La fonction **principale** de tout programme C. C'est cette fonction que le **compilateur** recherche pour exécuter le programme;
- La fonction main d'un programme n'acceptant aucun **argument** est

```
1 int main();
```

- Par convention, la valeur de **retour** de la fonction main est 0 si tout s'est bien déroulé et un **entier** correspondant à un **code d'erreur** différent de 0 autrement.

Les arguments de la fonction main

- Lorsque la fonction main accepte des paramètres, elle est de la forme :

```
1       int main(int argc, char *argv[]);
```

- argc correspond au **nombre d'arguments** (**incluant** le nom du programme);
- argv est un tableau de **chaînes de caractères**, vues comme des **pointeurs**.
- argv[0] est une chaîne de caractères représentant le **nom du programme**;
- argv[1] est le **premier argument**, etc.

Récupération des arguments de la fonction main

- Fonctions provenant de la bibliothèque stdlib.h;

```
1 double strtod(const char *chaine, char **fin);
2 unsigned long strtoul(const char *chaine, char **fin,
3                        int base);
4 long strtol(const char *chaine, char **fin, int base);
5 ...
```

- chaine : chaîne qu'on veut **traiter**;
- fin : ce qui **reste de la chaîne** après traitement;
- base : base dans laquelle le nombre est **exprimé dans la chaîne**;
- Les fonctions atof, atoi, atol, etc. sont **déconseillées**, car elles ne permettent pas de **valider** si la conversion s'est bien déroulée.

Documentation d'une fonction *facultatif*

- Bien qu'il n'y ait pas de **standard** de documentation en C, on utilise souvent le standard **Javadoc** :
- Aussi, si la **déclaration** (du **prototype**) et l'**implémentation** sont séparées, on documente plutôt la **première**.

```
1  /**
2   * Calcule la n-ième puissance de x.
3   *
4   * La n-ième puissance d'un nombre réel x, n étant un entier
5   * positif, est le produit de ce nombre avec lui-même répété
6   * n fois. Par convention, si n = 0, alors on obtient 1.0.
7   *
8   * @param x  Le nombre dont on souhaite calculer la puissance
9   * @param n  L'exposant de la puissance
10  * @return   Le nombre x élevé à la puissance n
11  */
12 float puissance(float x, unsigned int n);
```