

# Chapitre 3 : Les bases du C, partie 2

Guy Francoeur

basé sur les travaux d'  
Alexandre Blondin Massé, professeur

Département d'informatique  
Université du Québec à Montréal

6 janvier 2019  
Construction et maintenance de logiciels  
INF3135

# Table des matières

1. Opérateurs et conversions
2. Tableaux
3. Pointeurs
4. Chaînes de caractères
5. Fonctions, paramètres, variables

# Table des matières

1. Opérateurs et conversions
2. Tableaux
3. Pointeurs
4. Chaînes de caractères
5. Fonctions, paramètres, variables

# 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
```

# 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

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

```
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 ?

```
#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	$x \wedge y$

- ▶ Souvent utilisés pour combiner des **options** (*flags*);
- ▶ Par exemple, la fonction `SDL_Init` :

```
...  
if (SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) < 0) {  
    fprintf(stderr, "SDL failed to initialize: %s\n",  
            SDL_GetError());  
    return NULL;  
}  
...
```

- ▶ Aussi utile pour simuler les opérations **ensemblistes** de façon très compacte.

# Types numériques de base

- ▶ Plusieurs **conversions** (*cast*) se font automatiquement;
- ▶ 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**.

# Conversions implicites

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

```
// 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;
}
```

# Conversion de types

```
//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

1. Opérateurs et conversions
2. Tableaux
3. Pointeurs
4. Chaînes de caractères
5. Fonctions, paramètres, variables

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

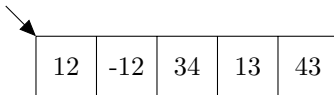
- **Déclaration** :

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

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

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

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



- ▶ À l'aide de l'opérateur `[]` :

```
// 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** :

```
char chaîne [] = "tomate";  
char chaîne [] = { '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

1. Opérateurs et conversions
2. Tableaux
3. Pointeurs
4. Chaînes de caractères
5. Fonctions, paramètres, variables

# Définition

- ▶ Un pointeur est l'**adresse** d'une donnée en **mémoire**;
- ▶ On déclare un pointeur en utilisant le symbole **\***;
- ▶ L'opérateur **&** retourne l'adresse d'une donnée en mémoire :

```
int *p; // Un pointeur vers un entier
```

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

```
int main() {  
    int x = 210;  
    printf("La variable x vaut %d et est stockée  
           à l'adresse hexa %p.\n", x, &x);  
  
    return 0;  
}
```

**Affiche** : La variable x vaut 210 et est stockée à l'adresse 0x7fff5fbff73c.

# Exemple

```
// 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 :

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

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

```
int *pi;  
pi = (int*)0xdff1; /* permis, mais à éviter */
```

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

```
int *pi;  
char *pc;  
pi = (int*)0xdff1;  
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 :

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

```
int main() {  
    int a[3] = {1,2,3}, *pi;  
    pi = a; /* initialisation de pi */  
    printf("%p %p %d %d %d %d\n",  
           a, pi, a[0], a[1], a[2], *pi);  
    return 0;  
}
```

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

# 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'**incréméntation** et la **décréméntation** de pointeurs sont possibles;
- ▶ Finalement, deux pointeurs peuvent être **comparés**.

# Exemple

```
// pointeur4.c
#include <stdio.h>

int main() {
    int a[3] = {1, -1, 2}, *pi, *pi2;
    pi = a;
    pi2 = &a[2];
    printf("%d ", pi2 - pi);
    printf("%d ", *(-pi2));
    printf("%d\n", *(pi + 1));
    if (pi + 1 == pi2)
        printf("pi et pi2 pointent vers la même case mé-
               moire.\n");
    return 0;
}
```

**Affiche :**

2 -1 -1

pi et pi2 pointent vers la même case mémoire.



```
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 l'adresse pointée);
- ▶ Les expressions

```
*pi = 6;
```

```
*(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

1. Opérateurs et conversions
2. Tableaux
3. Pointeurs
4. Chaînes de caractères
5. Fonctions, paramètres, variables

# Chaînes de caractères

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

t	o	m	a	t	e	\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 ?`

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

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 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

```
//ex1.c
#include <stdio.h>
#include <string.h>

int main() {
    char s[] = "bonjour";
    char t[] = "patate";

    printf("Longueur de \"%s\" et \"%s\" : %lu, %lu\n",
           s, t, strlen(s), strlen(t));
    printf("strcmp(\"%s\", \"%s\") : %d\n", s, t,
           strcmp(s, t));
    return 0;
}
```

## Sortie :

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

- ▶ Les fonctions

```
char *strcat(char *s, const char *t);  
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 ?

```
// 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;
}
```

- ▶ Les fonctions

```
char *strcpy(char *s, const char *t);  
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

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

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

- ▶ La fonction

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

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

"124:41:3::23:10"

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

- ▶ La fonction

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

résoud ce problème.

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

## Exemple (1/2)

```
#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 strtok:\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 strtok:

/124/

/41/

/3/

/23/

/10/

Avec strsep:

/124/

/41/

/3/

//

/23/

/10/

# Table des matières

1. Opérateurs et conversions
2. Tableaux
3. Pointeurs
4. Chaînes de caractères
5. Fonctions, paramètres, variables

# 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

```
int max(int x, int y) {  
    if (x >= y) return x;  
    else return y;  
}  
  
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 ?

```
//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

```
// 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;
}
```

# 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 reçoit les changements locaux.

# Passage d'un tableau

- ▶ Les tableaux comme **paramètres** d'une fonction :  
`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**.

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

# Exemple

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

float produit_scalaire(const float a[], const float b[],
                      unsigned taille) {

    float p = 0.0;
    for (int i = 0; i < taille; ++i) {
        p += a[i] * b[i];
    }
    return p;
}

int main() {
    float u[] = {1.0, -2.0, 0.0};
    float v[] = {-1.0, 1.0, 3.0};
    printf("%f\n", produit_scalaire(u, v, 3));
    return 0;
}
```

**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

```
#include <stdio.h>

int* initialise_tableau(unsigned taille) {
    int tableau[taille];
    int i;
    for (i = 0; i < taille; ++i)
        tableau[i] = 0;
    return tableau;
}

int main() {
    int *tableau;
    tableau = initialise_tableau(4);
    printf("%d\n", tableau[0]);
    return 0;
}
```

## 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** :

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

test.c:2: error: conflicting types for 'max'

test.c:1: error: previous declaration of 'max' was here

- ▶ 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

```
#include <stdio.h>
#include "math.c"

int main() {
    printf("PI = %f\n", PI);
    printf("Le carre de %d
           est %d\n", 4, carre
           (4));
    return 0;
}
```

## Affiche :

PI = 3.141593

Le carre de 4 est 16

## Fichier math.c

```
const float PI =
    3.141592654;

int carre(int x) {
    return x * x;
}
```

# Variables et fonctions statiques

```
static char tampon[TAILLE_TAMPON];  
static int x;  
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**.

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

- ▶ 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

```
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 :

```
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`;

```
double strtod(const char *chaine, char **fin);
unsigned long strtoul(const char *chaine, char **fin,
                     int base);
long strtol(const char *chaine, char **fin,
            int base);
...
```

- ▶ `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**.

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