

# INF3135

## Construction et maintenance de logiciels

### Chapitre 1: Bases du langage C

Alexandre Blondin Massé

Université du Québec à Montréal  
Département d'informatique

Été 2020

# Table des matières

- 1 Types
- 2 Variables et constantes
- 3 Structures de contrôle
- 4 Opérateurs
- 5 Conversions
- 6 Tableaux
- 7 Structures et unions
- 8 Fonctions
- 9 Compilation et Makefiles
- 10 Préprocesseur

# Types

# Types numériques

## Valeurs entières

- char:  $\geq$  **1 octet**
- short:  $\geq$  **2 octets**
- int: **2** ou **4 octets**
- long:  $\geq$  **4 octets**
- long long:  $\geq$  **8 octets**
- **Variantes:** signed (par défaut) ou unsigned

## Valeurs flottantes

- float:  $\geq$  **4 octets**
- double:  $\geq$  **8 octets**
- long double: **10**, **12** ou **16 octets**
- Toujours **signées**

# Exemple (entiers signés)

```
#include <stdio.h>

int main(void) {
    char  c = '+';
    short s = 67;
    int   i = -1;
    long  l = -28;
    // %c: character, %d: signed decimal, %l: long
    printf("c = %c %d\n", c, c);
    printf("s = %c %d\n", s, s);
    printf("i = %d\n", i);
    printf("l = %ld\n", l);
    return 0;
}
```

## Résultat:

```
c = + 43
s = C 67
i = -1
l = -28
```

# Exemple (entiers non signés)

```
#include <stdio.h>

int main(void) {
    unsigned char  c = -1;
    unsigned short s = -1;
    unsigned int   i = -1;
    unsigned long  l = -1;
    // %d: signed decimal, %u: unsigned decimal, %l: long
    printf("c = %d %u\n", c, c);
    printf("s = %d %u\n", s, s);
    printf("i = %d %u\n", i, i);
    printf("l = %ld %lu\n", l, l);
    return 0;
}
```

## Résultat:

```
c = 255 255
s = 65535 65535
i = -1 4294967295
l = -1 18446744073709551615
```

# Type booléen

## Avant C99

- Pas de type booléen **natif**
- 0: **faux**
- $\neq 0$ : **vrai**

## Depuis C99

- Ajout de la bibliothèque `stdbool.h`
- Définit le **type** `bool` (entier non signé)
- Et définit les **constantes** `true` et `false`

```
#define true 1  
#define false 0
```

# Exemple (booléens)

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

int main(void) {
    int u = 0;
    char c = 'A';
    bool t = true;
    bool f = false;
    if (u) printf("u ");
    if (c) printf("c ");
    if (t) printf("t ");
    if (f) printf("f ");
    if (c && t) printf("c&&t ");
    if (c == t) printf("c==t\n");
    return 0;
}
```

## Résultat:

c t c&&t



# Types énumératifs

- Une des façons de définir des **constantes**
- Par **défaut**: valeurs 0, 1, 2, etc.
- Une variable de type enum est traitée comme un int
- **Aucune vérification** n'est faite

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

```
enum day {MON, TUE, WED, THU, FRI, SAT, SUN};
```

```
enum http_code {  
    HTTP_CONTINUE           = 100, HTTP_OK           = 200,  
    HTTP_MULTIPLE_CHOICES   = 300, HTTP_BAD_REQUEST   = 400,  
    HTTP_FORBIDDEN          = 403, HTTP_NOT_FOUND      = 404  
};
```

```
int main(void) {  
    enum day d1 = MON, d2 = SAT;  
    enum http_code code = HTTP_NOT_FOUND;  
    printf("%d %d %d\n", d1, d2, code);  
}
```

**Résultat:**

0 5 404

# Types complexes

## Tableaux

- Permet de **concaténer** plusieurs valeurs de même type
- Les types doivent être **homogènes**

## Structures (produit)

- Permet de **concaténer** des types
- Les types peuvent être **hétérogènes**

## Unions (coproduit)

- Permet de proposer une **alternative** entre types
- Les types peuvent être **hétérogènes**

# Autres types (plus tard)

## Type vide

- Identifié par le mot `void`
- Définit le type d'une fonction **sans valeur** de retour
- Aussi la valeur **nulle** pour les pointeurs
- On va y **revenir** plus tard

## Pointeurs

- `char*`: pointeur vers `char`
- `int*`: pointeur vers `int`
- `int**`: pointeur vers `int*`
- `double***`: pointeur vers `double**`
- `void*`: pointeur « générique »
- On va y **revenir** plus tard

# Synonymes

- À l'aide de l'instruction typedef
- Aucune **vérification**

```
// Déclaration des synonymes
typedef unsigned int jour;
typedef float distance;
typedef enum {PIQUE, COEUR, CARREAU, TREFLE} couleur;

// Utilisation
jour lundi = 0, mardi = 1, samedi = 5;
distance d = 123.4;

void afficher_couleur(couleur c) {
    switch (c) {
        case PIQUE:    printf("pique");    break;
        case COEUR:    printf("coeur");    break;
        case CARREAU:  printf("carreau");  break;
        case TREFLE:   printf("trefle");   break;
    }
}
```

## Variables et constantes

# Affectation

## Syntaxe

Expression de la forme  $L = R;$

### *Left-value* ou *lvalue*

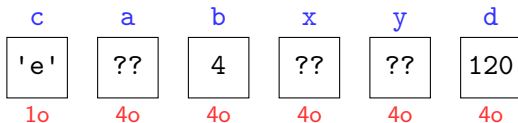
- Toute **expression** qui peut être placée à **gauche** de l'opérateur =
- Doit avoir un espace mémoire **réservé**
- **Exemples**: variable, champ d'une structure ou d'une union, pointeur

### *Right-value* ou *rvalue*

- Toute autre **expression** valide
- Ayant une **valeur**

# Variables

```
char c = 'e';  
int a, b = 4;  
float x, y;  
unsigned int d = factorial(5);
```



## Règles

- **Déclarée** avant son utilisation
- **Visible** seulement dans le bloc où elle est déclarée
- Peut être **initialisée** lors de la déclaration
- **Non initialisée** a une valeur indéterminée

# Type de variables (1/2)

## Automatiques (auto)

- Aussi appelées variables **locales**
- **Portée**: attachée au bloc qui la contient
- Mémoire **allouée** à la déclaration
- Et **libérée** à la fin du bloc
- Mot réservé auto **optionnel**

## Statiques (static)

- **Portée**: attachée à la fonction qui la contient
- Mot réservé static **requis**
- Mémoire **permanente**: **allouée** au début du programme
- Et **libérée** à la fin du programme
- On va y revenir **plus tard**



## Type de variables (2/2)

### Externes (extern)

- Aussi appelées variables **globales**
- **Portée**: tout le fichier qui la contient
- Les **fonctions** sont externes par défaut
- Mot réservé extern **optionnel**
- Permet de communiquer entre plusieurs **modules**
- Mémoire **allouée** dans un seul module
- Mais **déclarations multiples** permises
- Utile pour **interfacer** avec d'autres langages
- On va y revenir **plus tard**

# Constantes

## Trois mécanismes:

- **#define**: instruction au préprocesseur pour définir une **macro**
- **const**: **empêche** de modifier une variable
- **enum**: définition d'un type **énumératif**

```
#define PI 3.141592654
```

```
const float PI = 3.141592654;
```

```
enum sign {  
    NEG  = -1,  
    ZERO = 0,  
    POS  = 1  
};
```

# Valeurs littérales

```
unsigned int ui = 34u;  
long l = 34L;  
char c = 52, d = 064, e = 0X34, f = '4';  
// %u: unsigned decimal, %l: long, %d: decimal  
printf("%u %ld\n", ui, l);  
printf("%d %d %d %d\n", c, d, e, f);  
// Affiche :  
// 34 34  
// 52 52 52 52
```

- Suffixe u ou U: valeur **non signée**
- Suffixe l ou L: valeur **longue**
- Préfixe 0: valeur **octale**
- Préfixe 0x valeur **hexadécimale**

$$064 = 6 \times 8^1 + 4 \times 8^0 = 52$$

$$0X34 = 3 \times 16^1 + 4 \times 16^0 = 52$$

$$'4' = 52 \quad (\text{code ASCII})$$

# Caractères spéciaux

Quelques caractères utiles:

- `\n`: fin de ligne
- `\t`: tabulation
- `\\`: contre-oblique
- `\'`: apostrophe
- `\"`: guillemets

```
#include <stdio.h>

int main(void) {
    char c = '\\';
    printf("\\\"\\t%c\\\"\\n", c);
}
```

**Résultat:**

```
"\t"'
```

## Structures de contrôle

# Instruction for

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

- <initialisation> est évaluée une seule fois, au début
- <condition> est évaluée au début de chaque tour de boucle

<condition> est considérée  $\begin{cases} \textbf{faux}, & \text{si } \text{<condition>} == 0; \\ \textbf{vrai}, & \text{sinon.} \end{cases}$

- <incrémentation> est évaluée à la fin de chaque tour de boucle

## Déclaration et initialisation (1/2)

- On ne peut déclarer le type de l'itérateur dans l'**initialisation** d'une boucle for avec le standard **ANSI**:

```
/* Fichier for.c */
#include <stdio.h>

int main(void) {
    for (unsigned int i = 0; i < 10; ++i) {
        printf("%d ", i);
    }
    return 0;
}
```

```
$ gcc -ansi for.c
```

```
code/for.c: In function 'main':
```

```
code/for.c:5:5: error: 'for' loop initial declarations
    are only allowed in C99 or C11 mode
[...]
```

## Déclaration et initialisation (2/2)

**Deux corrections** possibles:

- Compiler selon un **standard** plus récent:

```
$ gcc -std=c99 for.c
```

```
$ gcc -std=c11 for.c
```

- Ou **réécrire** le programme:

```
/* Fichier for-ansi.c */  
#include <stdio.h>  
  
int main(void) {  
    unsigned int i;  
    for (i = 0; i < 10; ++i) {  
        printf("%d ", i);  
    }  
    return 0;  
}
```



# Instructions if, else if et else

```
if (<condition>) {  
    <suite instructions>  
}
```

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

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

- Branchement else optionnel
- **Accolades** optionnelles si instruction **unique**
- Attention aux structures **fortement imbriquées**

# Instruction switch

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

- Chaque expression case est examinée dans l'**ordre**
- Jusqu'à **correspondance** ou jusqu'à default
- L'instruction est alors **exécutée**
- Ainsi que toutes les instructions **suivantes**
- **Tant que** le mot réservé break n'est pas rencontré
- Le cas default est **optionnel**

## Exemple avec switch

```
#include <stdio.h>

void print_case(char c) {
    switch (c) {
        case 'A': printf("A");
        case 'B': printf("B"); break;
        case 'C': printf("C");
        default : printf("default");
    }
    printf("\n");
}

int main(void) {
    print_case('A'); print_case('B');
    print_case('C'); print_case('D');
    return 0;
}
```

### Résultat:

```
AB
B
Cdefault
default
```

# Boucles while et do-while

## Syntaxe:

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

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

- break permet de sortir de la boucle courante
- continue permet de passer immédiatement à l'itération suivante
- Utiliser break/continue seulement si simplifie la **lecture** ou
- Améliore l'**efficacité** du programme

# Opérateurs

# Opérateurs arithmétiques

- +: addition, -: soustraction
- \*: multiplication, /: division
- %: modulo

## Division entière

- entier / entier → **division entière**
- entier / flottant ou flottant / entier → **division flottante**

## Modulos

```
#include <stdio.h>

int main(void) {
    printf("%d %d %d %d\n",
           5 % 3, (-5) % 3, 5 % (-3), (-5) % (-3));
}
```

**Résultat:**

2 -2 2 -2

# Représentation interne

- Représentation par le **complément à deux** (cours INF2171):

	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

$$\rightarrow (128 - 1)_2 = (127)_2 = \mathbf{01111111}$$

$$\rightarrow (128 - 2)_2 = (126)_2 = \mathbf{01111110}$$

$$\rightarrow (128 - 127)_2 = (1)_2 = \mathbf{00000001}$$

$$\rightarrow (128 - 128)_2 = (0)_2 = \mathbf{00000000}$$

- S'il y a **débordement** (*overflow*), 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

- ==: égalité
- !=: inégalité
- >: stricte supériorité
- >=: supériorité
- <: stricte infériorité
- <=: infériorité

## Opérateurs logiques

- !: négation
- &&: et
- ||: ou (inclusif)
- L'évaluation est  **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
```

– (rarement utilisé) opérateur de séquençage ,: évalue les expressions dans l'ordre et retourne le résultat de la dernière

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

# Opérateur ternaire

<condition> ? <valeur si vrai> : <valeur si faux>

```
#include <stdio.h>

void print_room(unsigned int n) {
    printf("There %s %d pe%s in this room\n",
        n <= 1 ? "is" : "are",
        n,
        n <= 1 ? "rson" : "ople");
}

int main(void) {
    print_room(0);
    print_room(1);
    print_room(2);
}
```

## Résultat:

```
There is 0 person in this room
There is 1 person in this room
There are 2 people in this room
```

# Opérations bit à bit

- &: et
- |: ou inclusif
- ^: ou exclusif (xor)

## Utilité?

- Pour **optimiser** certains calculs (programmation vectorielle)
- Ou 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;  
}  
[...]
```

# L'opérateur sizeof

Retourne le **nombre d'octets** (`size_t`) utilisés par

- un **type** de données: `sizeof(int)`
- une valeur **littérale**: `sizeof("bonjour")`
- une **variable**: `sizeof(i)`
- un tableau de taille **fixe**: `sizeof(a)` (on va y revenir)

## Remarque

L'expression est évaluée à la **compilation**

## Utilité

- Code **plus portable**
- **Allocation dynamique** (on va y revenir)

# Taille des types entiers

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

int main(void) {
    printf("sizeof(bool)           = %ld\n", sizeof(bool));
    printf("sizeof(char)           = %ld\n", sizeof(char));
    printf("sizeof(short)          = %ld\n", sizeof(short));
    printf("sizeof(int)            = %ld\n", sizeof(int));
    printf("sizeof(long)           = %ld\n", sizeof(long));
    printf("sizeof(long long)      = %ld\n", sizeof(long long));
}
```

**Résultat** (varie selon l'architecture):

sizeof(bool)	= 1
sizeof(char)	= 1
sizeof(short)	= 2
sizeof(int)	= 4
sizeof(long)	= 8
sizeof(long long)	= 8

# Taille des types flottants

```
#include <stdio.h>

int main(void) {
    printf("sizeof(float)           = %ld\n", sizeof(float));
    printf("sizeof(double)          = %ld\n", sizeof(double));
    printf("sizeof(long double) = %ld",   sizeof(long double));
}
```

**Résultat** (varie selon l'architecture):

```
sizeof(float)           = 4
sizeof(double)          = 8
sizeof(long double) = 16
```

# Taille de variables et de tableaux

```
#include <stdio.h>

int main(void) {
    int i;
    char c;
    double xs[4];
    printf("sizeof i      = %ld\n", sizeof i);
    printf("sizeof c      = %ld\n", sizeof c);
    printf("sizeof(xs[0]) = %ld\n", sizeof xs[0]);
    printf("sizeof(xs)      = %ld\n", sizeof xs);
}
```

**Résultat** (varie selon l'architecture):

```
sizeof i      = 4
sizeof c      = 1
sizeof(xs[0]) = 8
sizeof(xs)    = 32
```

# Conversions



# Conversions des types numériques

- Souvent, on applique un opérateur **binaire**
- Sur deux valeurs de **types différents**
- Il y a alors **promotion**, ou **conversion** (*cast*) automatique
- Entre valeurs **entières**:

`bool → char → short → int → long → long long`

- Entre valeurs **flottantes**:

`float → double → long double`

- Promotion automatique **entier** → **flottant**
- Règles plus complexes pour types **signés** et **non signés**
- Éviter de **mélanger** les types dans une même opération
- Sauf cas **idiomatiques**
- Montrer les conversions de façon **explicite**

# Conversions implicites

Attention aux conversions implicites entre types signés et non signés:

```
#include <stdio.h>

int main(void) {
    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
}
```

# Conversion explicites

```
#include <stdio.h>

int main(void) {
    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	→	( ), [ ]
2	→	->, .
1	←	!, ++, --, +, -, (int), *, &, sizeof
2	→	*, /, %
2	→	+, -
2	→	<, <=, >, >=
2	→	==, !=
2	→	&&
2	→	
3	→	? :
1	←	=, +=, -=, *=, /=, %=
2	→	,

# Tableaux

# Tableaux

- Collection de données **homogènes** (de même type)
- Stockées de façon **contiguë** en mémoire

```
// Déclaration seulement
```

```
// Réserve un espace mémoire de taille 8 * sizeof(int)
```

```
int t1[8];
```

```
// Réserve un espace mémoire de taille n * sizeof(double)
```

```
// Seulement avec -std=c99 ou -std=c11
```

```
// Allocation sur la pile et non sur le tas (heap)
```

```
double t2[n];
```

```
// Définition et initialisation
```

```
// Réserve un espace mémoire de taille 8 * sizeof(int)
```

```
int t3[] = {5,2,0,1,3,4,7,6};
```

```
// Réserve un espace mémoire de taille 10 * sizeof(int)
```

```
// Deux premières valeurs initialisées à 1 et 1
```

```
// Autres valeurs indéterminées si variables automatiques
```

```
int t4[6] = {1,1};
```

# Représentation schématique de la mémoire

Tas

t1	0	1	2	3	4	5	6	7
	??	??	??	??	??	??	??	??

$$8 \times 4o = 32o$$

t3	0	1	2	3	4	5	6	7
	5	2	0	1	3	4	7	6

$$8 \times 4o = 32o$$

t4	0	1	2	3	4	5
	1	1	??	??	??	??

$$6 \times 4o = 24o$$

Pile

t2	0	1	2	3	4	...	n-1
	??	??	??	??	??	...	??

$$n \times 8o = (8n)o$$

# Opérateur []

```
#include <stdio.h>
#define ALPHABET_SIZE 26
#define SENTENCE "the quick brown fox jumps over a lazy dog"

int main(void) {
    // Déclaration
    unsigned int num_occurrences[ALPHABET_SIZE];
    // Initialisation
    for (unsigned int i = 0; i < ALPHABET_SIZE; ++i)
        num_occurrences[i] = 0;
    // Écriture
    char c;
    for (int i = 0; (c = SENTENCE[i]) != '\0'; ++i)
        if (c >= 'a' && c <= 'z')
            ++num_occurrences[c - 'a'];
    // Lecture
    for (int i = 0; i < ALPHABET_SIZE; ++i) {
        printf("%c: %d  ", i + 'a', num_occurrences[i]);
        if (i == 12) printf("\n");
    }
}
```

## Résultat:

a: 2 b: 1 c: 1 d: 1 e: 2 f: 1 g: 1 h: 1 i: 1 j: 1 k: 1 l: 1 m: 1  
n: 1 o: 4 p: 1 q: 1 r: 2 s: 1 t: 1 u: 2 v: 1 w: 1 x: 1 y: 1 z: 1



# Attention!

- Aucune **vérification** s'il y a dépassement
- Autant pour la **lecture** que pour l'**écriture**
- Source fréquente d'erreur de segmentation (*segfault*)

```
#include <stdio.h>

int main(void) {
    int a[] = {12, 24, 36, 48};
    int b[] = {60, 72};
    // %-3: alignement à gauche (-) sur 3 caractères (3)
    for (unsigned int i = 0; i <= 4; ++i)
        printf("a[%d] = %-3d  ", i, a[i]);
    printf("\n");
    a[1] = -24; a[4] = -60;
    for (unsigned int i = 0; i <= 4; ++i)
        printf("a[%d] = %-3d  ", i, a[i]);
}
```

## Résultat:

```
a[0] _u=12uuuuu a[1] _u=24uuuuu a[2] _u=36uuuuu a[3] _u=48uuuuu a[4] _u=-1140900672
a[0] _u=12uuuuu a[1] _u=-24uuuuu a[2] _u=36uuuuu a[3] _u=48uuuuu a[4] _u=-60
```

# Chaînes de caractères

- Cas **particulier** de tableau
- Ses éléments sont de type char
- Chaînes **littérales** délimitées par des guillemets " "
- Chaîne **bien formée**: doit terminer par le caractère \0

```
#include <stdio.h>

int main(void) {
    // En mémoire, ['l','i','n','u','x','\0']
    char s[] = "linux";
    for (unsigned int i = 0; i < 6; ++i) {
        printf("%d %c %d\n", i, s[i], s[i]);
    }
    return 0;
}
```

## Résultat:

```
0 l 108
1 i 105
2 n 110
3 u 117
4 x 120
5 0
```

## Bibliothèque `string.h`

Suppose que les chaînes sont **bien formées** (terminent par `\0`)

Plusieurs **fonctions** disponibles:

- `strlen`: longueur d'une chaîne
- `strcat/strncat`: concaténation de deux chaînes
- `strcmp/strncmp`: comparaison de deux chaînes
- `strcpy/strncpy`: copie d'une chaîne dans une autre
- `strstr`: première occurrence d'une sous-chaîne dans une chaîne
- `strtok`: segmentation (*tokenization*) d'une chaîne selon délimiteurs
- ...

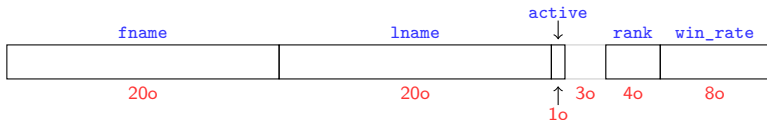
On va y revenir plus tard (pointeurs)

## Structures et unions

# Structure

- Aussi appelée **enregistrement**
- Regroupe en un même bloc des données **hétérogènes**
- Définit un **nouveau type** de données
- Déclarée à l'aide du mot réservé struct
- **Contigu** en mémoire
- **Alignement**: compilateur décale selon l'architecture
- Rend l'adressage plus **efficace**

```
struct Player {  
    char fname[20];  
    char lname[20];  
    bool active;  
    unsigned int rank;  
    double win_rate;  
};
```



# Déclaration, initialisation, lecture, écriture

```
#include <stdio.h>

// Déclaration d'un nouveau type
struct point2d {
    double x;
    double y;
};

int main(void) {
    // Déclaration d'une variable non initialisée
    struct point2d p1;
    // Déclaration et initialisation
    struct point2d p2 = {2.0, -1.2};
    // Écriture dans champs avec opérateur .
    p1.x = 3.6;
    p1.y = -4.9;
    // Lecture des champs avec opérateur .
    // %f: nombre flottant (float ou double)
    printf("p1 = (%f,%f)\n", p1.x, p1.y);
    printf("p2 = (%f,%f)\n", p2.x, p2.y);
}
```

## Résultat:

```
p1 = (3.600000,-4.900000)
p2 = (2.000000,-1.200000)
```

## Affectation (*compound literal*)

- Affectation en bloc possible
- Depuis standard **C99**

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

```
struct Rectangle {  
    float x;  
    float y;  
    float width;  
    float height;  
};
```

```
int main(void) {  
    struct Rectangle r = {1.0, 2.0, 5.0, 6.0};  
    // Affectation en bloc (conversion obligatoire)  
    r = (struct Rectangle){3.0, 8.0, 9.0, 7.0};  
    float a = 0.0, b = 0.0, c = 1.0, d = 2.0;  
    // Affectation en bloc avec champs nommés  
    r = (struct Rectangle){.x      = a,  
                           .y      = d,  
                           .width  = b,  
                           .height = c};  
  
    return 0;  
}
```

# Copie de structures

- Opérateur = sur les structures: copie les **champs** un par un
- Lors d'appel de fonctions, la structure est **copiée**

```
#include <stdio.h>

struct point2d { double x; double y; };

void print_point(struct point2d p) {
    printf("point(%f,%f)", p.x, p.y);
}

int main(void) {
    struct point2d p1 = {3.2, -1.4};
    struct point2d p2 = p1;
    p2.y *= -1;
    print_point(p1); printf("\n"); print_point(p2);
    return 0;
}
```

## Résultat:

```
point(3.200000,-1.400000)
point(3.200000,1.400000)
```



# Structures imbriquées

- Les structures peuvent être **imbriquées**
- Elles peuvent aussi être composées avec des **tableaux**
- Et avec des **unions** et des **pointeurs** (plus tard)

```
struct point2d {  
    double x;  
    double y;  
};
```

```
struct segment {  
    struct point2d p;  
    struct point2d q;  
};
```

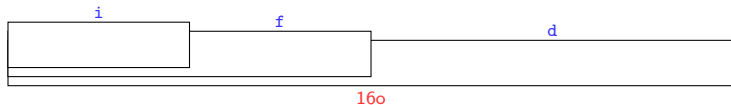
```
struct triangle {  
    struct point2d points[3];  
};
```

```
struct carre {  
    struct point2d points[4];  
};
```

# Unions

- Contient des données de **types différents**
- Une seule donnée peut exister à un **instant donné**
- Même **syntaxe** que pour les structures
- Mémoire réservée:  $\geq$  sizeof du type le **plus volumineux**
- **Utilité**: factorisation de code, économie d'espace
- **Difficulté**: il faut savoir quel type est le bon

```
// Un nombre entier ou flottant
union nombre {
    int    i;
    float  f;
    double d;
};
```



# Example

```
#include <stdio.h>

int main(void) {
    union nombre {
        int    i;
        float  f;
        double d;
    };
    union nombre n;
    n.i = 152;
    printf("%d %f %f\n", n.i, n.f, n.d);
    n.f = 87.31;
    printf("%d %f %f\n", n.i, n.f, n.d);
    n.d = -999.999;
    printf("%d %f %f\n", n.i, n.f, n.d);
}
```

**Résultat:** (peut varier)

```
152 0.000000 0.000000
1118740152 87.309998 0.000000
-206158430 -28882151778513119643386622509056.000000 -999.999000
```

# Structures, unions et enum anonymes

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

struct choice {
    bool is_number;                // marqueur de type de donnée
    union {                       // union anonyme
        float number;
        enum {YES, NO, MAYBE} answer; // enum anonyme
    };
};

void print_choice(struct choice c) {
    if (c.is_number) {
        printf("%lf\n", c.number);
    } else {
        switch (c.answer) {
            case YES:    printf("yes");    break;
            case NO:     printf("no");     break;
            case MAYBE:  printf("maybe");
        }
        printf("\n");
    }
};

int main() {
    struct choice c = {false, .answer = YES}; print_choice(c);
    c = (struct choice){true, 3.14};         print_choice(c);
}
```

## Résultat:

```
yes
3.140000
```

# Utilisation de typedef

- On peut éviter de réécrire struct, union et enum
- En utilisant l'instruction typedef
- **Avantage**: syntaxe plus proche de Java et C++

```
#include <stdio.h>

typedef struct {
    double x;
    double y;
} point2d;

int main(void) {
    point2d p = {2.0, -1.5};
    printf("point(%f,%f)\n", p.x, p.y);
    return 0;
}
```

- Mais considéré **abusif** (voir [discussion](#), en particulier [cette réponse](#))
- Donc à **éviter** dans le cours

# Fonctions

# Utilité des fonctions

- **Unité de base** d'un programme (avec les types)
- Effectue une tâche **précise**
- Doit préférablement être **courte**
- Permet de **diviser** un problème complexe
- À la base de la **réutilisation**
- Et de la **factorisation** de code
- Fondamentales pour la **maintenance**
- Doivent être **bien nommées**
- Avec une syntaxe et une logique **uniforme**
- Devraient être **documentées**

## Fonction `main`

- Fonction spéciale, point d'entrée du programme
- Communique à l'aide des **paramètres** `argc` et `argv`
- Et d'une **valeur de retour** de type `int`
- On va y revenir

# Fonctions pures et effets de bord

## Fonction pure

- Le résultat ne **dépend** que des arguments
- Retourne le **même résultat** si appelée avec les **mêmes arguments**
- Pas d'**effet** de bord
- Par exemple, les fonctions **mathématiques**
- Ou les fonctions de **lecture seule**

## Fonction non pure ou à effets de bord

- Le résultat dépend de l'**environnement** ou le modifie
- Peut retourner un **résultat différent** même lorsque appelée avec les **mêmes arguments**
- **Exemples**: écriture sur `stdout`, lecture sur `stdin`, lecture/écriture fichier, allocation dynamique, chargement de données, ...



# Arguments et paramètres

- **Paramètre**: nom de la variable dans la fonction
- **Argument**: valeur passée lors de l'appel
- Valeurs passées par **copie**
- Aussi possible par **adresse** (on va y revenir)

## La valeur void

- void si pas de valeur retournée
- `f(void)`: fonction sans paramètre
- `f()`: fonction avec nombre inconnu de paramètres

## Nombre variable de paramètres

- `f(int, ...)`: paramètre entier, suivi de paramètres optionnels
- **Exemple**: `printf` et `scanf`

# Exemples

```
#include <stdio.h>

// Aucun paramètre, aucune valeur retournée
void f1(void) {
    printf("f1()");
};

// Un paramètre, aucune valeur retournée
void f2(int a) {
    printf("f2(%d)", a);
}

// Aucun paramètre, une valeur retournée
int f3() {
    return 42;
}

// Un paramètre, une valeur retournée
int f4(int a) {
    return -a;
}

int main(void) {
    f1(); // Pas d'argument
    f2(4); // Argument obligatoire
    f3(); // Pas obligatoire de récupérer la valeur
    printf("%d\n", f4(4));
    return 0;
}
```

## Résultat:

f1()f2(4)-4

## Déclaration et définition (1/2)

- Attention à l'ordre de **déclaration** des fonctions
- Car la compilation se fait en **une** passe

```
#include <stdio.h>

int a(int x) {
    return 42;
}

int b(int x) {
    return c(x - 1) + 1;
}

int c(int x) {
    return a(x + 1);
}
```

### Avertissement à la compilation:

```
fonctiondd.c: In function 'b':
fonctiondd.c:8:12: warning: implicit declaration of function 'c'
    [-Wimplicit-function-declaration]
    return c(x - 1) + 1;
```

## Déclaration et définition (2/2)

**Solution:** séparer les déclarations et les définitions

```
#include <stdio.h>

// Déclarations
int a(int x);
int b(int x);
int c(int x);

// Définitions
int a(int x) {
    return 42;
}

int b(int x) {
    return c(x - 1) + 1;
}

int c(int x) {
    return a(x + 1);
}

int main(void) {
    return 0;
}
```

# Fonction récursive

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

void print_reverse(char s[], int n) {
    // Cas de base: si n < 0, on ne fait rien
    if (n >= 0) {
        // Cas général: 1) on affiche la dernière lettre
        printf("%c", s[n]);
        // 2) puis on diminue la longueur de la chaîne
        print_reverse(s, n - 1);
    }
}

int main(void) {
    char s[] = "linux";
    char t[] = "esoperesteicietserepose";
    print_reverse(s, strlen(s) - 1); printf("\n");
    print_reverse(t, strlen(t) - 1); printf("\n");
    return 0;
}
```

## Résultat:

```
xunil
esoperesteicietserepose
```

## La fonction printf (1/3)

- Affichage (print) **formaté** (f)
- Disponible dans la **bibliothèque** `stdio.h`
- Affichage des **types** de base:

Code	Description
%c	caractère
%d	entier sous forme décimale
%hd	entier court sous forme décimale
%ld	entier long sous forme décimale
%u	entier non signé
%o	entier sous forme octale
%x	entier sous forme hexadécimale
%e	flottant en notation scientifique
%f	flottant en notation décimale
%g	flottant de façon compacte
%lf	double en notation décimale
%L	long double en notation décimale
%s	chaîne de caractères
%p	pointeur

## La fonction printf (2/3)

- Options d'affichage:

Code	Description
%-	Alignement à gauche (par défaut à droite)
%+	Ajoute le symbole + aux nombres positifs
%	Ajoute un espace aux nombres positifs
%#	Ajoute un préfixe 0 ou 0X si octal ou hexadécimal
%<n>	Largeur d'affichage d'au moins <n>
%*	Largeur d'affichage variable
%.<n>f	Précision numérique ou remplissage de <n>

- Très pratique pour afficher des **tableaux** et des **grilles**
- Autres **options** disponibles (voir la documentation)

## La fonction printf (3/3)

```
#include <stdio.h>

int main(void) {
    // Fixe
    printf("|%11d|%11f|\n",      32, -1.4);
    printf("|%-11d|%-11f|\n",   32, -1.4);
    printf("|%.5d|%.5f|\n",      32, -1.4);
    printf("|%11.5d|%11.5f|\n",  32, -1.4);
    // Variable
    printf("|%*d|*f|\n",        11,      32, 11,      -1.4);
    printf("|%-*d|%-*f|\n",     11,      32, 11,      -1.4);
    printf("|%.*d|%.5f|\n",     5,       32, 5,        -1.4);
    printf("|%*.*d|%*.*f|\n",  11, 5,    32, 11, 5,    -1.4);
}
```

### Résultat:

```
|          32|   -1.400000|
|32          |-1.400000  |
|00032|-1.40000|
|          00032|   -1.40000|
|          32|   -1.400000|
|32          |-1.400000  |
|00032|-1.40000|
|          00032|   -1.40000|
```



# La fonction main

- Point d'**entrée** d'un programme C
- Trois **signatures** possibles

```
// Aucun argument permis
```

```
int main(void);
```

```
// Avec arguments
```

```
// argc: nombre d'arguments, incluant le nom du programme
```

```
// argv: tableaux d'arguments
```

```
//     argv[0]: nom du programme
```

```
//     argv[1]: premier argument
```

```
//     argv[2]: deuxième argument
```

```
//     ...
```

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

```
// Nombre d'arguments indéfinis (à éviter)
```

```
int main();
```

- char\* argv[]: tableau de pointeurs vers char (on va y revenir)

## Exemple avec arguments

- argv[0]: nom du programme **tel qu'invoqué**
- **Protection** avec " ou '
- Variables d'**environnement** disponibles

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    printf("argc = %d\n", argc);
    for (unsigned int i = 0; i < argc; ++i)
        printf("argv[%d] = %s\n", i, argv[i]);
    return 0;
}
```

### Résultat:

```
$ ./prog alpha 1 "un deux" 'trois quatre' "$LANG"
argc = 6
argv[0] = ./prog
argv[1] = alpha
argv[2] = 1
argv[3] = un deux
argv[4] = trois quatre
argv[5] = en_CA.UTF-8
```

# Fonction et variables statiques

## Variables statiques

- On peut attacher des variables **statiques** à une fonction
- À l'aide du mot réservé `static`
- Mémoire réservée au **début** du programme
- Et **libéré** à la fin du programme
- **Initialisée** à 0 par défaut

## Mémoïsation

- On considère n'importe quelle **fonction pure**
- On « **mémorise** » le résultat de la fonction
- Pour certains **arguments** donnés
- Lors d'un appel de fonction avec les **mêmes** arguments
- On n'a pas à **refaire** le calcul

## Exemple: fibo.c

```
#include <stdio.h>

unsigned long long fibonacci(unsigned int n) {
    if (n == 0 || n == 1)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}

int main(void) {
    // %lld: long long decimal
    for (unsigned int n = 0; n < 40; ++n) {
        printf("%lld ", fibonacci(n));
    }
    printf("\n");
    return 0;
}
```

### Problème?

On recalcule plusieurs fois les **mêmes** valeurs

# Chronomètre

```
$ gcc fibo.c -o fibo
$ time ./fibo
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
2584 4181 6765 10946 17711 28657 46368 75025 121393
196418 317811 514229 832040 1346269 2178309 3524578
5702887 9227465 14930352 24157817 39088169 63245986
102334155
real    0m1.452s
user    0m1.451s
sys     0m0.000s
```

- **Solution 1:** « dérécursifier » le programme
- Mais pas **facile** en général
- **Solution 2:** « mémoriser » les valeurs déjà calculées
- À l'aide d'une variable **statique**

## Example: fibofast.c

```
#include <stdio.h>

#define MAX_N 1000

unsigned long long fibonacci(unsigned int n) {
    // Le tableau f contient les MAX_N premières valeurs
    // On l'initialise avec {1,1,0,0,0,0,...}
    static unsigned long long f[MAX_N] = {1, 1};
    if (n >= MAX_N)
        // Tableau pas assez grand, on y va naïvement
        return fibonacci(n - 1) + fibonacci(n - 2);
    if (f[n] == 0) {
        // Tableau assez grand, on mémorise
        f[n] = fibonacci(n - 1) + fibonacci(n - 2);
    }
    return f[n];
}

int main(void) {
    for (unsigned int n = 0; n < 40; ++n) {
        printf("%lld ", fibonacci(n));
    }
    printf("\n");
    return 0;
}
```

# Chronomètre

```
$ gcc fibofast.c -o fibofast
$ time ./fibofast
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
2584 4181 6765 10946 17711 28657 46368 75025 121393
196418 317811 514229 832040 1346269 2178309 3524578
5702887 9227465 14930352 24157817 39088169 63245986
102334155
real    0m0.002s
user    0m0.002s
sys     0m0.000s
```

# Compilation et Makefiles



# Compilateur

- GCC = *GNU Compiler Collection*
- Cygwin GCC: chaîne d'outils Linux pour Windows
- Mingw GCC = *Minimalist GNU for Windows*
- Clang LLVM = *Low Level Virtual Machine*
- Plusieurs autres...

## Remarque

Sur **MacOS**, gcc est un alias pour clang:

```
$ gcc --version
[...]  
Apple LLVM version 10.0.0 (clang-1000.10.44.4)  
Target: x86_64-apple-darwin17.7.0  
Thread model: posix  
[...]
```

# GCC

- **Ensemble** de compilateurs
- Sous une **interface** commune
- Développée par **GNU**
- Plusieurs **langages** supportés:

C, C++, Objective-C, Fortran, Ada, Go, D

## Plusieurs options

- `-c`: compilation seulement
- `-o FICHIER`: spécifier le nom du fichier en sortie
- `-Wall`: afficher tous les avertissements
- `-Wextra`: afficher encore plus d'avertissements
- `-std=STD`: spécifier le standard (c99, c11, etc.)
- Commande `man gcc` pour toutes les options

# Cycle de compilation

## Édition du programme source (.c)

À l'aide d'un éditeur de texte ou d'un EDD

## Compilation (.c $\rightarrow$ .o)

- Vérification **syntaxique**
- Produit des fichiers (binaires) **objets**

## Édition de liens (.o $\rightarrow$ .out) (*linking*)

- Fichiers .o assemblés pour former un binaire **exécutable**
- Extension .out par **défaut**

# Simplifier la compilation

On a vu un peu plus tôt la compilation en deux étapes pour créer un exécutable

- On compile le fichier `.c` en un fichier `.o`

```
$ gcc -c maj.c
```

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

```
$ gcc -o maj maj.o
```

- **Problème:** pénible de saisir la commande de compilation chaque fois qu'on apporte une modification au fichier source
- Encore plus avec les **options:**

```
$ gcc -c -Wall -Wextra maj.c
```

```
$ gcc maj maj.o
```

- **Solution:** utiliser un Makefile

# Makefiles

- Existent depuis la fin des années '70
- Fichier **texte**
- Décrit les **dépendances** entre composantes d'un programme
- Automatisent la **compilation** en minimisant le nombre d'étapes
- Malgré qu'ils soient archaïques, ils sont encore **très utilisés**
- Certaines **limitations** corrigées par des outils comme Autoconf et CMake (on va y revenir)
- Préférer le nom `Makefile` (avec une majuscule) à `makefile`

## Contenu d'un Makefile

- Des règles **explicites**
- Des règles **implicites** (on va y revenir)
- Des définitions de **variables**
- Des **directives** spécifiques à `make` (on va y revenir)
- Des commentaires, préfixés par `#`

# Syntaxe d'une règle explicite

```
<cible>: <prérequis>  
<tab><recette>
```

- <cible>: nom de **fichier** ou nom **personnalisé** ou nom **spécial**
- <prérequis>: noms de **fichier** ou autres **cibles**
- séparés par des **espaces**
- <tab>: caractère de **tabulation**, pas d'espaces
- <recette>: suite de **commandes** permettant de générer <cible>

## Exemple

```
maj: maj.o  
    gcc -o maj maj.o  
  
maj.o: maj.c  
    gcc -c -Wall -Wextra maj.c
```

# Invocation d'un Makefile

- **Invocation** avec la commande `make`:

```
$ make  
gcc -c -Wall -Wextra maj.c  
gcc -o maj maj.o
```

- La commande `make` ne regarde que la **première** règle
- Et ses **dépendances**, si elles doivent être mises à jour
- Par défaut, les commandes sont **affichées**
- Possible de les faire taire avec `@` ou `-s|--silent`

## Astuce Vim

- Associer les touches `,m` à l'invocation `!make`
- Voir **fichier `vimrc` en labo**

# Variables

## Syntaxe

- **Déclaration:** `<nom variable> = <valeur>`
- **Utilisation:** `$(<nom variable>)`
- Variables **textuelles**

## Exemple

```
exec = maj
CFLAGS = -Wall -Wextra

$(exec): $(exec).o
    gcc -o $(exec) $(exec).o

$(exec).o: $(exec).c
    gcc -c $(CFLAGS) $(exec).c
```



# Cibles spéciales

- **Interprétées** de façon particulière par make
- Voir **documentation**
- En **majuscules**
- **Préfixées** par un point .

## Exemples

```
.PHONY, .SUFFIXES, .DEFAULT, .PRECIOUS, .INTERMEDIATE,  
.SECONDARY, .SECONDEXPANSION, .DELETE_ON_ERROR, .IGNORE,  
.SILENT, .POSIX, ...
```

## .PHONY

- Permet de déclarer des cibles **personnalisées**
- Indique que ce ne sont pas des **noms** de fichier

## Exemple avec .PHONY

```
exec = maj
readme = README
CFLAGS = -Wall -Wextra -std=c11
```

```
$(exec): $(exec).o
    gcc -o $(exec) $(exec).o
```

```
$(exec).o: $(exec).c
    gcc -c $(exec).c
```

```
.PHONY: clean html
```

```
clean:
    rm -f *.o
    rm -f $(exec)
```

```
html:
    pandoc -o $(readme).html -sc pandoc.css $(readme).md
```

# Invocation d'une cible personnalisée

## Syntaxe

```
$ make <cible>
```

## Exemple

```
$ make  
gcc -c -Wall -Wextra -std=c11 maj.c  
gcc -o maj maj.o  
$ make clean  
rm -f *.o  
rm -f maj  
$ make html  
pandoc -o README.html -sc pandoc.css README.md
```

# Préprocesseur

# Directives au préprocesseur

## Précompilation

- **Interprétées** par le préprocesseur **avant** la compilation
- Symbole # au **début** de la ligne
- On peut insérer des **espaces** entre # et la directive

## Directives permises

- #include: **inclusion** d'un fichier externe
- #define: définition d'un **symbole** ou d'une **macro**
- #undef: **annulation** d'un symbole ou d'une macro
- #ifdef/#ifndef: **vérifie** si une macro est définie ou non définie
- #if/#else/#elif/#endif: structure **conditionnelle**
- #error: indique une **erreur fatale**
- #pragma: pour des traitements plus **spécifiques**

# La directive `#include`

- Permet d'**inclure** un fichier source externe
- Généralement, on inclut le fichier d'**en-tête** (extension `.h`)
- Inclusion d'un fichier `.c` possible, mais à **éviter**

## Exemples

```
// Bibliothèques standards
#include <stdio.h>
#include <stdbool.h>
#include <math.h>

// Modules locaux
#include "utils.h"
#include "constants.h"

// Bibliothèques tierces
#include <jansson.h>
#include <cairo.h>
#include <sdl2.h>
```

# La directive #define

- **Syntaxe:** #define <symbole> <valeur>
- **Remplace** toutes les occurrences de <symbole> par <valeur>
- La valeur est donnée par **le reste de la ligne**
- Pour une valeur **multiligne**, utiliser le caractère \

## Exemples

```
#define BUFFER_SIZE 100
#define ERROR_MSG "Error: wrong number of arguments"
#define USAGE "\
Usage: %s [-h|--help] [-n|--num-iterations VALUE]\n\
\n\
Simulates the propagation of a rumour.\n\
\n\
Optional arguments:\n\
    -h, --help           Shows this help message and exit.\n\
    -n, --num-iterations VALUE The number of iterations.\n\
                           The default value is 100.\n\
"
```

## Autres directives

- #undef: **annulation** d'un symbole ou d'une macro
- #ifdef/#ifndef: **vérifie** si une macro est définie ou non définie
- #if/#else/#elif/#endif: structure **conditionnelle**

### Utilité

- Empêcher les **inclusions** multiples de modules
- **Portabilité** du code

### Plus tard

- Quand on va parler de **modules**
- Puis de **bibliothèques** (*libraries*)