Chapitre 2: Introduction au langage C

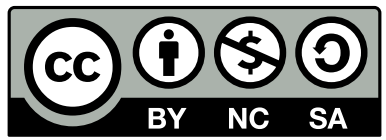
INF3135

Construction et maintenance de logiciels

Alexandre Blondin Massé

Université du Québec à Montréal

v251



Plan

Compilation

Exemple

Fichier `maj.c`:

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

int main() {
  char c;
  while ((c = getchar()) != EOF) {
    putchar(toupper(c));
  }
  return 0;
}
```

Question

Comment **compile**-t-on ce programme?

Compilation

- **Édition du programme source**: à l'aide d'un éditeur de texte ou d'un environnement de développement

→ L'extension du fichier est `.c`

- **Compilation**: traduction du langage C en langage machine

→ Indique les erreurs de syntaxe

→ Ignore les fonctions et les bibliothèques invoquées

→ Génère un fichier avec l'extension `.o`

- **Édition de liens** (*linking*): Le code machine de différents fichiers `.o` est assemblé pour former un fichier binaire

→ Par défaut, l'extension est `.out` sous Unix

→ L'extension est `.exe` sous Windows

→ L'exécutable a les droits `ugo+x` par défaut

- **Exécution du programme**: en ligne de commande ou en double-cliquant sur l'icône du fichier binaire.

Compilation « directe »

- On peut combiner la **compilation** et l'**édition des liens** en une seule commande:
- Produit (par défaut) le fichier exécutable `a.out`

```
$ gcc maj.c
$ ls -l a.out
-rwxr-xr-x 1 ablondin ablondin 8392 Sep  3 07:33 a.out
```

- Puis on l'exécute:

```
$ ./a.out
Langage de balisage léger
LANGAGE DE BALISAGE LéGER
^C
```

- Pour nommer l'exécutable (par exemple `maj`):

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

Compilation et édition des liens

- Parfois, il est préférable de **séparer** la compilation et l'édition des liens

→ projets de plus grande envergure

- On compile d'abord en langage machine chaque fichier

→ par défaut l'extension `.c` devient `.o`

```
$ gcc -c maj.c
$ gcc -o maj.o -c maj.c  # Équivalent
```

- Produit le fichier **objet** `maj.o`
- Ensuite:

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

- Produit un fichier exécutable nommé `maj`
- Exécution:

```
$ ./maj
```

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 relancer la compilation chaque fois qu'on apporte une modification au fichier source
- **Solution**: utiliser 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 existent depuis longtemps, ils sont encore **très utilisés** (et le seront encore pour très longtemps)
- Certaines **limitations** des Makefiles sont corrigées par des outils comme Autoconf et CMake (vu vers la fin du trimestre)

Question

On préfére le nom `Makefile` (avec une majuscule) à `makefile`.
Pourquoi?

Exemple

- Exemple de `Makefile` minimal pour un programme C d'un seul fichier:

```
maj: maj.o
    gcc -o maj maj.o

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

- La **syntaxe** est de la forme

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

- Le caractère `<tab>` est très important!

Invocation d'un Makefile

- Pour invoquer un `Makefile`, il suffit d'entrer

```
$ make
gcc -c exemple.c
gcc -o exemple exemple.o
```

- Par défaut, les commandes sont **affichées** (possible de les faire taire)

Astuce Vim

On peut associer les caractères `<leader>m` (par exemple `,m`) à la commande `make` en l'indiquant dans le fichier `vimrc`

Variables et constantes

Types de base

Types numériques

- `char` (signé ou pas): 1 octet
- `short` (signé ou pas): 2 octets
- `int` (signé ou pas): 2 ou 4 octets (selon l'architecture)
- `long` (signé ou pas): 4 octets
- `float`: 4 octets
- `double`: 8 octets
- `long double`: 16 octets
- `size_t`: entier non signé

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)

Type booléen

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

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

int main() {
  int i = 4;
  unsigned int u = 0;
  char c = 'A';
  bool b = false;
  if (i) printf("i ");
  if (u) printf("u ");
  if (c) printf("c ");
  if (b) printf("b ");
  return 0;
}
```

Déclaration des variables

Une variable

- doit être **déclarée** avant son utilisation, en début de bloc (accolades)
- est **visible** seulement dans le bloc où elle est déclarée
- peut être **initialisée** lors de la déclaration
- **non initialisée** a une valeur indéterminée

```
// Avec initialisation
char c = 'e';
// a n'est pas initialisée, b l'est
int a, b = 4;
// Ni x ni y ne sont initialisées
float x, y;
// d est initialisée en invoquant une fonction
unsigned int d = fibonacci(10);
```

Constantes

- À l'aide de la directive `#define` (**macro**, on va y revenir)

```
#define PI 3.141592654
```

- Avec le mot réservé `const`

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

- À l'aide d'un type énumératif

```
enum Status {
  OK = 0,
  WRONG_FORMAT = 1,
  WRONG_VALUE  = 2
};
```

Valeurs numériques littérales

- Le suffixe `u` ou `U` indique une valeur **non signée**
- Le suffixe `l` ou `L` indique une valeur **longue**
- Le préfixe `0` indique une valeur **octale**

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

- Le préfixe `0x` indique une valeur **hexadécimale**

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

- Un caractère est un nombre

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

```
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 contre-oblique
- `\'`: l'apostrophe
- `\"`: les guillemets

```
#include <stdio.h>

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

$ gcc special.c && ./a.out
"\   '"
```

Types énumératifs

- Déclaration

```
enum Jour {LUN, MAR, MER, JEU,
           VEN, SAM, DIM};
```

- Une des façons de définir des **constantes**
- La première valeur vaut `0`, la seconde vaut `1`, etc.
- Seules des valeurs entières (`int`) sont permises:

```
// Ne fonctionne pas !!!
enum ConstanteMath {PI = 3.141592654,
                    E = 2.7182818};
```

Types énumératifs et entiers

- Une variable de type `enum` est traitée comme un `int`
- Aucune vérification n'est faite

```c
#include <stdio.h>

enum State {
  OK = 0,
  ERROR = 1
};

int main() {
  enum State s = 3;
  int i = ERROR;
  printf("%d %d\n", s, i);
}
```

Affiche: `3 1`

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 chaque tour de boucle, avant d'exécuter le corps de la boucle
- `<incrémentation>` est évaluée lors de chaque tour de boucle, après avoir exécuté le corps de la boucle.

Initialisation dans les anciens standards

- **Attention**: on ne peut déclarer le type de l'itérateur dans l'initialisation qu'à partir du standard **C99**
- Le fragment de code suivant ne compile pas avec le standard ANSI:

```
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` et `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**

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

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

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.
- S'il y a correspondance, alors l'instruction correspondante est exécutée, ainsi que toutes les instructions suivantes, tant que le mot réservé `break` n'est pas rencontré
- Le cas `default` est **optionnel**

Question sur `switch`

Quel est le résultat du programme suivant?

```c
#include <stdio.h>

int main() {
  char c = 'B';
  switch (c) {
    case 'A': printf("A");
    case 'B': printf("B");
    case 'C': printf("C");
              break;
    default: printf("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

Opérateurs et conversions

Opérateurs arithmétiques

- `+`: addition
- `-`: soustraction
- `*`: multiplication
- `/`: division
- `%`: modulo

Division entière

- Lorsque les deux opérandes de la division sont des types **entiers**, alors la division est entière également
- Si un des opérandes est un nombre **flottant**, alors la division est entre nombres flottants

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** (*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>
```

- 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

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

Utilité?

- Pour optimiser certains calculs
- 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;
}
[...]
```

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, éviter de mélanger les types dans une même opération ou montrer les conversions de façon explicite

Conversions implicites

Attention aux conversions implicites entre types signés et non signés:

```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
}
```

Conversion explicites

```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	$\rightarrow$	`( )`, `[ ]`
2	$\rightarrow$	`->`, `.`
1	$\leftarrow$	`!`, `++`, `--`, `+`, `-`, `(int)`, `*`, `&`, `sizeof`
2	$\rightarrow$	`*`, `/`, `%`
2	$\rightarrow$	`+`, `-`
2	$\rightarrow$	`<`, `<=`, `>`, `>=`
2	$\rightarrow$	`==`, `!=`
2	$\rightarrow$	`&&`
2	$\rightarrow$	`||`
3	$\rightarrow$	`?` `:`
1	$\leftarrow$	`=`, `+=`, `-=`, `*=`, `/=`, `%=`
2	$\rightarrow$	`,`

Pointeurs et tableaux

Définition

- Un pointeur est l'**adresse** d'une donnée en mémoire
- On déclare un pointeur en utilisant le symbole *

```
int* p1; // Un pointeur vers un entier
int* p2 = NULL; // Un pointeur initialisé
```

- L'opérateur & retourne l'adresse d'une donnée en mémoire

```
#include <stdio.h>

int main() {
  int x = 210;
  printf("x = %d\n", x);
  printf("&x = %p\n", &x);
}
```

Exemple

```c
#include <stdio.h>

int main() {
  int* pi, x = 104;
  pi = &x;
  printf("x = %d, &x = %p\n", x, &x);
  printf("pi = %p, *pi = %d\n", pi, *pi);

  *pi = 350;
  printf("x = %d, &x = %p\n", x, &x);
  printf("pi = %p, *pi = %d\n", pi, *pi);
}
```

Résultat (peut varier):

```
x = 104, &x = 0x7ffe109e64ac
pi = 0x7ffe109e64ac, *pi = 104
x = 350, &x = 0x7ffe109e64ac
pi = 0x7ffe109e64ac, *pi = 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 pas souhaitable */
```

- 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;
```

Tableaux

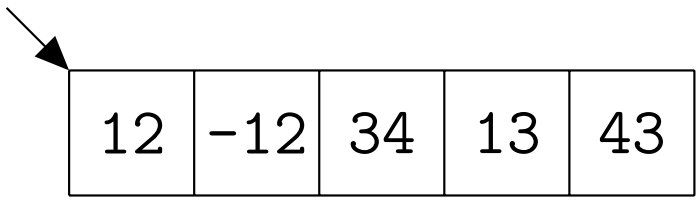
- Collection de données de **même type**
- Déclaration:

```
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



Accès

- À l'aide de l'opérateur `[]`

```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 b ? */
}
```

- Le premier élément est à l'indice `0`
- Comportement **indéfini** s'il y a dépassement de bornes
- Source fréquente d'erreur de segmentation (*segfault*)

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;
  printf("%p %p %d %d %d %d\n",
         a, pi, a[0], a[1], a[2], *pi);
}
```

- **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]`

Autres opérations

- 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

```c
#include <stdio.h>

int main() {
  int a[3] = {1, -1, 2}, *pi, *pi2;
  pi = a;
  pi2 = &a[2];
  printf("%ld ", pi - pi2);
  printf("%d ", *(--pi2));
  printf("%d\n", *(pi + 1));
  if (pi + 1 == pi2)
      printf("pi + 1 == pi2\n");
}
```

```
2 -1 -1
pi + 1 == pi2
```

Gestion de la mémoire

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

- La déclaration d'un tableau réserve l'espace mémoire nécessaire pour stocker le **tableau complet**
- La déclaration de `*pi` ne réserve que l'espace mémoire pour stocker l'**adresse pointée**
- Si l'espace mémoire pointé par `pi` n'a pas été réservé, les expressions suivantes compilent, mais ont un **comportement indéterminé**:

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

- Le programme pourrait éventuellement utiliser cet espace
- Source fréquente d'erreurs de **segmentation** (*segfault*)

Chaînes de caractères

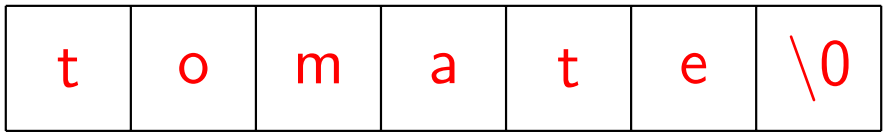
- **Chaîne** de caractères = **tableau** de caractères
- Les chaînes **littérales** sont délimitées par des guillemets
- Les deux déclarations suivantes sont équivalentes:

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

- Se termine par le caractère `\0`

→ Longueur de la chaîne `"tomate"` : `6`

→ Taille du tableau de la chaîne `"tomate"` : `7`



- Des fonctions élémentaires sur les caractères se trouvent dans la bibliothèque standard `ctype.h`
- La bibliothèque standard `string.h` fournit plusieurs fonctions permettant de manipuler les chaînes de caractères

La bibliothèque `ctype.h`

- `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`

La bibliothèque `string.h`

La fonction

```
size_t strlen(const char* s)
```

retourne la longueur d'une chaîne de caractères

La fonction

```
int strcmp(const char* s, const char* t)
```

retourne

- une valeur `< 0` si `s` apparaît avant `t` dans un dictionnaire (ordre lexicographique)
- une valeur `> 0` si `s` apparaît après `t`
- la valeur `0` si `s` et `t` sont les mêmes chaînes de caractères

Question

Quelle est la différence entre `s == t` et `strcmp(s, t) == 0`?

Exemple

```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;
}
```

Résultat:

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

Concaténation

Les fonctions

```
char* strcat(char* s, const char* t);
char* strncat(char* s, const char* t, size_t n);
```

permettent de **concaténer** deux chaînes de caractères

- 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

Question

Qu'est-ce qui est affiché sur stdout par le programme suivant?

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

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

Copie

Les fonctions

```
char* strcpy(char* s, const char *t);
char* strncpy(char* s, const char *t, size_t n);
```

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

- La chaîne `t` est copiée dans la chaîne `s`, incluant le caractère `\0`
- 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 de `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 caractèreres donnés

- Le paramètre `s` correspond à la chaîne qu'on souhaite **segmenter**,
- Le paramètre `delim` donne la liste des caractères **séparateurs**

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`, pour des fins d'optimisation

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);
  }
}
```

Exemple (2/2)

Résultat:

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

Fonctions

Utilité des fonctions

- Elles forment les **unités** de base de programmation
- Chaque fonction doit effectuer **une tâche** bien précise
- Elles doivent être **courtes**
- 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.

Deux types de fonctions

Les fonctions **pures**:

- Le résultat ne **dépend** que des arguments
- Pas d'**effet** de bord
- Par exemple, les fonctions **mathématiques**
- Ou les fonctions de **lecture seule**

Les fonctions **non pures**

- Le résultat dépend de l'**environnement** ou le modifie
- Par exemple, les fonctions d'allocation dynamique, les fonctions utilisant des variables globales

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 (exemple: `x` et `y`)
- Un **argument** de fonction est une valeur passée à une fonction lors de son appel (exemple: `3` et `4`)
- Les fonctions ont un, aucun ou plusieurs paramètres en **entrée**
- Elles renvoient **au plus** un résultat en **sortie**

Documentation d'une fonction

- Bien qu'il n'y ait pas de **standard** de documentation en C, on utilise souvent le standard Javadoc (obligatoire dans les **TP**)
- Aussi, si la déclaration (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);
```

Passage par valeur

- 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.

```
#include <stdio.h>

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

int main() {
  int x = 2;
  printf("%d %d\n", carre(x), x);
}
```

Affiche `4 2`

Exercice

Quelles sont les valeurs affichées par ce programme?

```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);
}
```

Passage par adresse

Correction du programme:

```c
#include <stdio.h>

void echanger(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main() {
  int a = 3, b = 2;
  echanger(&a, &b);
  printf("a = %d, b = %d\n", a, b);
  return 0;
}
```

Passage d'un tableau

- Les tableaux peuvent être **arguments** d'une fonction

```
float produit_scalaire(const float a[],
                       const float b[],
                       unsigned 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 important d'utiliser le mot réservé `const`

Exemple

```c
#include <stdio.h>

float produit_scalaire(const float a[],
                       const float b[],
                       unsigned int taille) {
  int i;
  float p = 0.0;
  for (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));
}
```

Affiche: `-3.000000`

Fonction retournant un tableau

- On ne peut retourner un pointeur créé dans une fonction
- Sauf s'il y a eu **allocation dynamique** (plus tard)
- **Solution**: le tableau doit être un des **arguments**

```
#include <stdio.h>

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

int main() {
  int tableau[8];
  initialise_tableau(tableau, 4);
  printf("%d\n", tableau[0]);
}
```

Déclaration et définition des fonctions

- C'est une bonne pratique de déclarer les **prototypes** (signatures) des fonctions au début du fichier où elles sont définies
- Pas nécessaire, mais encouragé de donner un **nom** aux variables
- Contrairement à C++ et Java, la **surcharge** (*overloading*) de fonctions est interdite:

```
unsigned int nombreOccurrences(const char *s,
                               char c);
unsigned int nombreOccurrences(const char *s,
                               const char *t);
```

Affiche:

```
occ.c:2: error: conflicting types for
         'nombreOccurrences'
occ.c:1: error: previous declaration of
         'nombreOccurrences' was here
```

Visibilité des variables et des fonctions

- La visibilité des variables en C est plus complexe lorsqu'on manipule **plusieurs fichiers**
- D'abord, il y a les variables **locales** (appelées **automatiques** en C), dont la portée est limitée à un bloc donné
- Ensuite, il y a les variables **globales** qui sont visibles dès leur déclaration et ce, jusqu'à la fin du fichier
- 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, mais ont une **durée** de vie plus longue

Variables locales (automatiques)

- **Visibles** dès leur déclaration jusqu'à la fin du bloc où elles sont définies
- **Accessibles** uniquement dans leur bloc
- **Supprimées** lorsqu'on sort du bloc
- Aucune garantie sur leur valeur lorsqu'elles ne sont pas initialisées

Quelles valeurs sont affichées par le programme suivant?

```
int main() {
  int i = 0, j = 0;
  if (i == 0) {
    int i = 5;
    printf("%d ", i);
  }
  printf("%d\n", i);
}
```

Variables et fonctions globales

- **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

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));
}
```

Fichier `math.c`

```
const float PI = 3.141592654;

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

Variables et fonctions statiques

```
#define SIZE 100
[...]
static char buffer[SIZE];
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 `main`

- Point d'**entrée** de tout programme C
- C'est cette fonction que le **compilateur** recherche pour exécuter le programme
- La fonctin `main` d'un programme n'acceptant aucun argument est

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

Code d'erreur

Par convention, la valeur de **retour** de la fonction `main` est `0` si tout s'est bien déroulé et un entier indiquant un **code d'erreur** 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`: nombre d'arguments, incluant le nom du programme
- `argv`: tableau de chaînes de caractères

→ `argv[0]`: nom du programme
→ `argv[1]`: premier argument
→ `argv[2]`: deuxième argument
→ ...

Conversion de chaînes en nombres

- 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 à traiter
- `fin`: ce qui reste de la chaîne après traitement
- `base`: base dans laquelle le nombre est exprimé
- Les fonctions `atof`, `atoi`, `atol`, sont plus limitées si on souhaite valider si la conversion s'est bien déroulée

Types composés

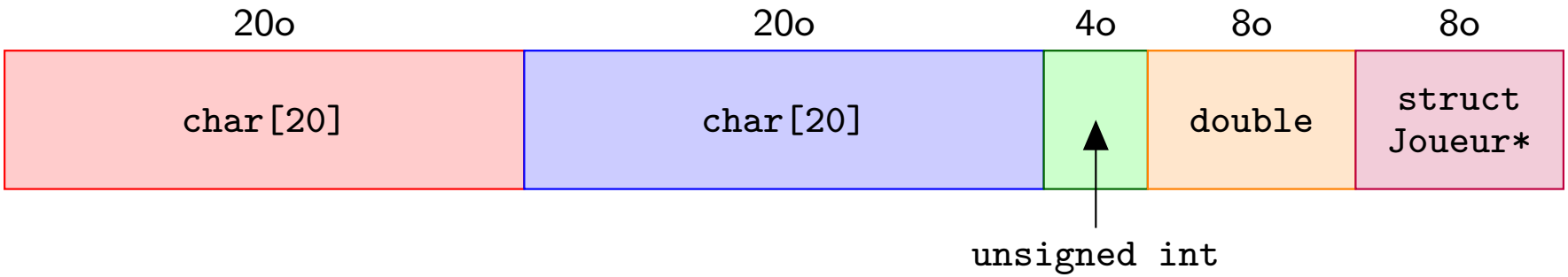
Les structures

- Parfois appelées **enregistrements**
- Permet de regrouper sous un même bloc des données **hétérogènes**
- Définissent un **nouveau type** de données
- Déclarées à l'aide du mot réservé `struct`

```
struct Joueur {
  char prenom[20];
  char nom[20];
  unsigned int classement;
  double taux_victoire;
  struct Joueur *derniers_adversaires[5];
};
```

Représentation en mémoire

```
struct Joueur {
  char prenom[20];
  char nom[20];
  unsigned int classement;
  double taux_victoire;
  struct Joueur *derniers_adversaires[5];
};
```



o = octet

Déclaration et initialisation

```
struct Point2d {
  double x;
  double y;
};
```

- Déclaration d'une variable de type `struct Point2d`:

```
struct Point2d p;
```

- Attention de ne pas oublier le mot `struct` dans la déclaration:
- On peut combiner déclaration, initialisation et définition.

```
struct Point2d p = {2.0, -1.2};
```

Affectation (*compound literal*)

- On peut initialiser une structure en spécifiant les **champs**
- On peut aussi faire une affectation en bloc

```c
#include <stdio.h>

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

int main() {
  struct Rectangle r = {1.0, 2.0, 5.0, 6.0};
  // r = {3.0, 8.0, 9.0, 7.0}; Syntaxe non valide
  r = (struct Rectangle) {3.0, 8.0, 9.0, 7.0};
  float a = 0.0, b = 0.0, c = 1.0, d = 2.0;
  r = (struct Rectangle) {.x      = a,
                          .y      = d,
                          .width  = b,
                          .height = c};
  return 0;
}
```

Manipulation des structures

```
struct Point2d p1 = {-1.2, 2.1};
struct Point2d p2 = p1;
```

- L'affectation `p2 = p1` copie les champs des structures
- Les structures sont passées par **valeurs** aux fonctions
- Pour accéder aux différents membres d'une structure, il faut utiliser l'opérateur point (.):

```
void affichePoint(struct Point2d p) {
  printf("(%f, %f)", p.x, p.y);
}

int main() {
  struct Point2d p = {2.0, -1.2};
  affichePoint(p);
}
```

Résultat: `(2.000000, -1.200000)`

Pointeur sur une structure

- Lorsqu'on a un **pointeur** vers une structure, on doit utiliser l'opérateur ->
- La plupart du temps, il est préférable de passer les structures par **adresse** aux fonctions
- C'est plus **efficace**, en particulier lorsque les structures sont de taille importante
- Par exemple, si on compare deux points:

```
int ptcmp(const struct Point2d *p,
          const struct Point2d *q) {
  if (p->x != q->x) return p->x - q->x;
  else return p->y - q->y;
}
```

- L'expression `p->x` est un sucre syntaxique pour `(*p).x`

Types composés

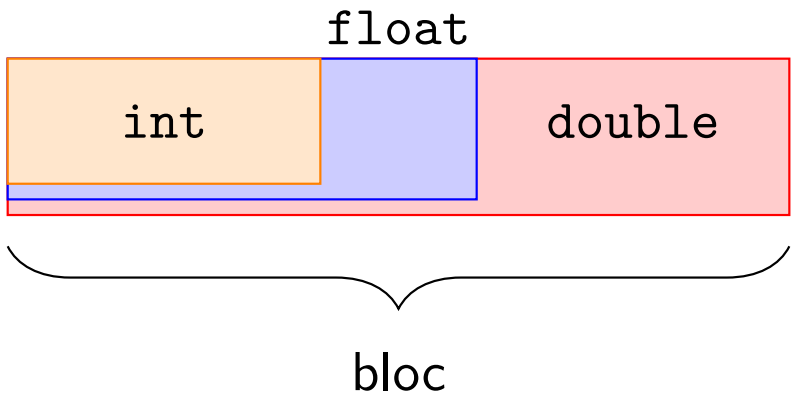
- Les structures peuvent être **imbriquées**
- Elles peuvent aussi être composées avec des **pointeurs** et des **tableaux**

```
struct Segment {
  struct Point2d p;
  struct Point2d q;
};

struct Carre {
  struct Point2d points[4];
};

// Alternative possible
struct Carre {
  struct Point2d *points[4];
};
```

Unions

- Variables dont le contenu diffère selon le contexte
- La variable sera créée avec une taille suffisamment grande pour contenir le type le **plus volumineux**
- La syntaxe est la même que pour les structures

```
union Nombre {
  int    i;
  float  f;
  double d;
};
```



Exemple

```c
#include <stdio.h>

int main() {
  union Nombre {
    int    i;
    float  f;
    double d;
  };
  union Nombre n;
  n.i = 3;
  printf("%d %f %lf\n", n.i, n.f, n.d);
  n.f = 2.0;
  printf("%d %f %lf\n", n.i, n.f, n.d);
  n.d = 3.0;
  printf("%d %f %lf\n", n.i, n.f, n.d);
}
```

Affiche:

```
3 0.000000 0.000000
1073741824 2.000000 0.000000
0 0.000000 3.000000
```

Initialisation des unions

- Comme les structures, les unions peuvent être initialisées en bloc
- Par contre, seul le premier membre peut être initialisé.

```
#include <stdio.h>

int main() {
    union Nombre {
        int    i;
        float  f;
        double d;
    };
    union Nombre n1 = {3};
    printf("%d %f %lf\n", n1.i, n1.f, n1.d);
    union Nombre n2 = {2.1};
    printf("%d %f %lf\n", n2.i, n2.f, n2.d);
}
```

Résultat:

```
3 0.000000 0.000000
2 0.000000 0.000000
```

Structures et unions anonymes (1/2)

Les structures et les unions peuvent être **anonymes**:

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

struct Choix {
  bool estNombre;
  union { float nombre; char *chaine; };
};

void afficherChoix(struct Choix *choix) {
  if (choix->estNombre)
    printf("%lf\n", choix->nombre);
  else
    printf("%s\n", choix->chaine);
};

int main() {
  struct Choix choix = {false, .chaine = "oui"};
  afficherChoix(&choix);
  choix = (struct Choix){true, 3.14};
  afficherChoix(&choix);
}
```

L'instruction `typedef`

- Permet de définir de **nouveaux types**

```
typedef char NAS[9];
typedef char *String;
typedef struct {
  float x;
  float y;
} Point2d;

NAS nas;
String s;
Point2d p;
```

- Améliore la **lisibilité** du code
- Les types sont seulement des **synonymes**
- **Exemple**: `char *` et `String` sont interchangeables

Utilisation de `typedef`

- Des programmeurs expérimentés considèrent que l'instruction `typedef` est utilisée de façon **abusive**
- Voir une discussion intéressante, en particulier cette réponse
- En tant que programmeurs, cependant, si vous avez à lire du code écrit en C, il est probable que vous rencontriez les deux pratiques
- Il est donc important d'être familier avec les instructions `typedef`

Portée de `struct`, `union`, `enum` et `typedef`

- Même portée que les variables et les fonctions
- Si déclaré **localement**, alors limité au bloc dans lequel ils sont déclarés
- Si déclaré **globalement**, alors accessible jusqu'à la fin du fichier
- Par contre, impossible de les déclarer **externes**
- Pour rendre des structures, des unions, des enums et des types accessibles dans n'importe quel fichier, il faut alors les déclarer dans une interface (fichier `.h`) qu'on inclut à l'aide de l'instruction `#include` dans le préambule.

L'opérateur `sizeof`

Retourne le nombre d'octets utilisés par

- un type de données: `sizeof(int)`
- une valeur constante: `sizeof("bonjour")`
- le nom d'une variable: `sizeof(matrice)`

Très pratique:

- L'expression est évaluée à la **compilation**
- Permet de produire du code **plus portable**
- Apparaît souvent lors d'**allocation dynamique**

Exemple

```
#include <stdio.h>

int main() {
  typedef struct {
    int quantite;
    float poids;
  } Fruit;
  int a[5];

  printf("%lu %lu %lu %lu %lu\n", sizeof(int),
         sizeof(float), sizeof(Fruit), sizeof a,
         sizeof "bonjour");
}
```

Affiche: `4 4 8 20 8`

Précompilation

Directives au préprocesseur

Préfixées par le symbole `#`

Exemple de directives:

- `#include`
- `#define`
- `#if`
- `#endif`
- `#ifndef`, etc.

Les directives sont **lues et interprétées** par le préprocesseur avant même de procéder à la **compilation** des différents fichiers

Symboles

- Pour définir un **symbole** ou une macro, on utilise la directive

```
#define <identificateur> <valeur>
```

- Le préprocesseur remplace toutes les occurrences de `<identificateur>` (comme mot) par `<valeur>`
- La valeur est donnée par **le reste de la ligne**
- Pour affecter une valeur sur **plusieurs lignes**, il faut utiliser le caractère `\`
- La **portée** du symbole s'étend jusqu'à la **fin du fichier** dans lequel il est défini
- Sauf si on trouve une commande

```
#undef <identificateur>
```

Exemple

Fichier `preproc.c` :

```c
#include <stdio.h>

#define i x

/*
 * Commentaire quelconque
 */
int main() {
  int i = 6, j;
  if (i) {
#undef i
    j = i * 2;
  }
  return 0;
}
```

Après `gcc -E preproc.c` :

```
# 1 "preproc.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 325 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "preproc.c" 2

# 1 "/usr/include/stdio.h" 1 3 4
# 64 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/sys/cdefs.h" 1 3 4
# 506 "/usr/include/sys/cdefs.h" 3 4
# 1 "/usr/include/sys/_symbol_aliasing.h" 1 3 4
# 507 "/usr/include/sys/cdefs.h" 2 3 4
# 572 "/usr/include/sys/cdefs.h" 3 4
# 1 "/usr/include/sys/_posix_availability.h" 1 3 4
# 573 "/usr/include/sys/cdefs.h" 2 3 4
# 65 "/usr/include/stdio.h" 2 3 4
# 1 "/usr/include/Availability.h" 1 3 4
# 153 "/usr/include/Availability.h" 3 4
# 1 "/usr/include/AvailabilityInternal.h" 1 3 4
# 154 "/usr/include/Availability.h" 2 3 4
# 66 "/usr/include/stdio.h" 2 3 4

# 1 "/usr/include/_types.h" 1 3 4

⋮
```

Symboles prédéfinis

Fichier `predefini.c`:

```c
#include <stdio.h>

int main() {
  printf("Nom du fichier source courant: %s\n", __FILE__);
  printf("Numéro de la ligne courante: %d\n", __LINE__);
  printf("Date de compilation: %s\n", __DATE__);
  printf("Heure de compilation: %s\n", __TIME__);
  printf("Compilateur conforme à la norme ISO? %s\n",
         __STDC__ == 1 ? "oui" : "non");
  return 0;
}
```

Affiche:

```
Nom du fichier source courant: predefini.c
Numéro de la ligne courante: 5
Date de compilation: Nov  5 2019
Heure de compilation: 09:32:14
Compilateur conforme à la norme ISO? oui
```

Constantes

- Dans certains cas, il est **nécessaire** d'utiliser des symboles pour définir des constantes

```
#include <stdio.h>

int main() {
  const int nbLig = 2;
  int a[nbLig] = {1,2};
}
```

```
preproc/tableau.c: In function 'main':
preproc/tableau.c:5: error: variable-sized object
may not be initialized
```

- Pour le compilateur, les variables constantes sont des **variables** qu'on ne peut modifier, mais pas des **constantes**