IUT de Montpellier, Dép. Informatique
M2101 - Archi 2 V. Poupet

# TD n° 1 - Introduction à la programmation en C

## 1 Organisation d'un programme en C

### 1.1 Premier exemple

Voici un exemple simple[1] de programme en C :

```
#include <stdio.h>
int main() {
  printf("Youpi !\n");
}
```

**1** Ouvrez un éditeur de texte (par exemple `geany`), et recopiez le programme. Sauvegardez-le sous le nom « `simple.c` ».

Ce programme ne peut pas être exécuté directement par l'ordinateur. Pour pouvoir être exécuté, il doit être *compilé* c'est-à-dire traduit en *langage machine*. Il existe plusieurs compilateurs, mais le plus couramment utilisé est `gcc` (*GNU C Compiler*).

**2** Compilez le programme en tapant la commande « `gcc simple.c` » dans un terminal (il faut vous placer dans le répertoire contenant le fichier `simple.c`).

Le compilateur produit alors un nouveau fichier qui est exécutable. Par défaut, l'exécutable produit par `gcc` s'appelle `a.out`, mais on peut choisir un autre nom avec l'option « `-o <nom de fichier>` » au moment de la compilation.

**3** Essayez de regarder le contenu du fichier `a.out` créé par `gcc` à la question précédente (c'est très moche, mais c'est normal).

**4** Quels sont les droits d'accès sur le fichier `a.out` ? Exécutez-le.

**Remarque :** Par défaut, les programmes en C ne peuvent utiliser aucune fonction pré-définie. Pour utiliser des fonctions existantes, il faut le préciser au début du programme. Dans l'exemple, on veut afficher du texte à l'écran, en utilisant la fonction `printf`. Cette fonction se trouve dans la *bibliothèque* `stdio.h` (*standard input/output*), d'où la première ligne.

Il existe de nombreuses bibliothèques disponibles en C, avec énormément de fonctions déjà écrites qui peuvent être utilisées dans les programmes. Les deux plus couramment utilisées sont `stdio.h` et `stdlib.h` qui contiennent la plupart des fonctions de base, mais vous pouvez être amenés à en utiliser d'autres selon les besoins spécifiques de vos programmes (`math.h`, `time.h`, `string.h`, etc.).

1. On peut faire encore plus court, par exemple « `int main(){}` » mais le programme ne fait rien du tout...

### 1.2 Structure générale

Un programme en C commence généralement par l'inclusion d'autres fichiers nécessaires (bibliothèques définissant des fonctions ou des types) :

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

La différence entre les inclusions avec « `< >` » et « `" "` » est que la première syntaxe recherche le fichier à inclure dans les répertoires standards du système (que le compilateur connaît) tandis que la seconde recherche dans le répertoire courant (ou le répertoire explicitement mentionné entre les guillemets).

On peut également définir (en général en début de programme) des constantes pour le pré-compilateur :

```
#define NBCASES 100
#define TEXTE "Tralala"
```

La syntaxe de ces définitions est « `#define <nom> <valeur>` ». Cela ne définit pas de variables, mais indique au compilateur qu'il faut remplacer toutes les occurrences de `<nom>` par `<valeur>` dans le programme. Ce remplacement est fait syntaxiquement (chercher-remplacer) sur le code source avant la compilation (donc en phase de *pré-compilation*).

**5** Modifiez le programme `simple.c` en définissant une constante `MESSAGE` ayant pour valeur « `"Super\n"` », et en passant cette constante comme argument à la fonction `printf`. Compilez et exécutez le programme.

On peut également définir des *variables globales* en les déclarant en dehors de toute fonction (typiquement au début du programme, après les inclusions et les définitions de constantes) :

```
float x;
char carac, *mot;
int entier, tab[10];
```

Les variable en C sont déclarées en indiquant leur *type*, suivi du nom de la variable. Si on veut déclarer plusieurs variable ayant le même type, on peut séparer les noms par des virgules (sans avoir à répéter le type).

**Remarque :** Dans l'exemple, « `int entier, tab[10]` » définit un entier `nombre` ainsi qu'un tableau contenant 10 entiers nommé `tab`. La notation « `*mot` » de la deuxième ligne est également spéciale, nous y reviendrons par la suite.

On peut bien sûr définir des fonctions :

```
int mult(int a, int b) {
  return a*b;
}
```

La syntaxe générale d'une fonction est :

```
<type du résultat> <nom de fonction> (<type1> <nom1>, <type2> <nom2>, ...) {
...
}
```

Une fonction peut éventuellement ne rien renvoyer (son type de retour est alors `void`) et peut prendre n'importe quel nombre d'arguments (éventuellement aucun, comme la fonction `main` du premier exemple).

Enfin, tout programme en C doit contenir une fonction appelée « `main` » dont le type de retour est `int`. C'est cette fonction qui est appelée lorsque le programme est exécuté.

**Remarque :** Dans le premier exemple, la fonction `main` est déclarée avec le type de retour `int` bien qu'elle semble ne rien renvoyer. Ce n'est pas un problème car si le code de la fonction ne renvoie rien (aucune instruction `return`) le programme considère qu'elle renvoie `0`.

**6** Définissez une fonction `dbl` qui prend en argument un entier et renvoie le double de cet entier. Modifiez la fonction `main` pour qu'elle affiche le résultat de `dbl(4)` (ça devrait être 8...).

**Indication :** Pour afficher un entier, il faut utiliser le code « `%d` » dans la fonction `printf` puis donner l'entier à afficher. Par exemple :

```
printf("résultat: %d\n", a);
```

si `a` est un entier.

## 2 Les types

Les types de base en C sont :

**`char` :** un octet représentant un caractère. Pour décrire la valeur d'un caractère, on le place entre guillemets simples (attention, le nom de la variable n'est pas entre guillemets, mais le caractère qu'il faut mettre comme valeur l'est) :

```
char c = 'A';
```

**`int` :** un nombre entier dont la taille correspond à celle des entiers du système d'exploitation (en général 4 octets) ;

**`float` :** un nombre en virgule flottante en simple précision (4 octets) ;

**`double` :** un nombre en virgule flottante en double précision (8 octets) ;

Pour déclarer un tableau de valeurs, on ajoute la taille du tableau entre crochets après le nom de la variable :

```
int tab[12];
float autre_tab[20];
```

On peut accéder aux éléments du tableau en indiquant l'indice entre crochets également :

```
tab[0] = 15;
tab[1] = tab[0] + 2;
printf("Valeur : %d\n", tab[1]);
```

**Remarque :** Dans un tableau de taille $n$, les indices varient de 0 à $(n-1)$.

**7** Écrivez un programme qui déclare un tableau contenant 3 entiers, affecte les valeurs 1 et 2 dans les deux premières cases du tableau, puis affiche à l'écran les valeurs dans le tableau aux indices 0, 1, 2 et 3.

Que remarquez-vous ? (les résultats peuvent varier d'un ordinateur à un autre et même d'une exécution à l'autre)

**Remarque :** Vous pouvez utiliser plusieurs fois le code `%d` dans un appel de `printf`, en lui donnant alors autant d'arguments supplémentaires que de codes spéciaux :

```
printf("a: %d, b: %d, c: %d\n", a, b, c);
```

Les chaînes de caractères sont des tableaux de caractères dont le dernier caractère est `\0` (ce qui correspond au caractère dont le numéro dans la table ASCII est 0, et non pas le caractère qui affiche le chiffre 0 à l'écran, qui serait `'0'` en C et correspond au numéro 48).

**8** Combien d'espace mémoire occupe la chaîne de caractères `"Bonjour"` ?

# 3 Pointeurs et adresses

Tous les objets utilisés par un programme en cours d'exécution sont écrits dans la mémoire RAM de l'ordinateur. La mémoire est divisée en *cases* pouvant chacune contenir un octet et l'on associe à chacune de ces cases un numéro : son *adresse*.

Ainsi, pour retrouver un objet en mémoire, il faut savoir l'adresse à laquelle il est enregistré. Il se peut que l'objet soit écrit sur plusieurs cases s'il occupe plus d'un octet, auquel cas il faut pouvoir retrouver toutes les cases. Les objets simples (`int`, `float`, etc.) sont écrits sur des cases consécutives et il suffit donc de trouver la première case, mais il arrive que l'on manipule des objets plus complexes qui sont répartis à différents endroits.

## 3.1 Pointeurs

Un *pointeur* sur un objet est une variable dont la valeur est l'adresse mémoire où se trouve l'objet. Si `ptr` est un pointeur, on désigne par « `*ptr` » l'objet pointé par `ptr`. La notation inverse de `*` est `&`. Ainsi, si `obj` est un objet (un entier, une chaîne de caractères, ou n'importe quoi d'autre), `&obj` désigne son adresse mémoire.

Lorsque l'on déclare une variable par :

```
int *tab;
```

on déclare que `tab` est un pointeur, et que l'objet sur lequel il pointe (`*tab`) est un entier. `tab` est donc un pointeur sur un entier.

En C, on ne manipule directement que les objets très simples (types primitifs `int`, `char`, etc.). La plupart des autres objets (tableaux, ou structures plus complexes) sont contrôlés par l'intermédiaire de pointeurs.

**9** Soit le programme suivant :

```
#include <stdio.h>

int main() {
  int a, *b;
  a = 12;
  b = &a;
  *b = a + 1;
  printf("a = %d, b = %d\n", a, b);
}
```

Expliquez ce que fait chacune des lignes. Que valent les variables `a` et `b` à la fin de l'exécution ? Recopiez, compilez et exécutez le programme pour vérifier vos réponses.

**10** Il se peut que le compilateur se soit plaint lors de la compilation du programme (s'il ne s'est pas plaint, recompilez en utilisant l'option `-Wall` qui active tous les *warnings*[2]). Essayez de comprendre le message d'alerte.

**Indication :** On rappelle que le code `%d` dans la fonction `printf` sert à afficher des entiers, donc des objets de type `int`.

**11** Modifiez la dernière ligne du programme en

2. Les *warnings* sont des alertes que le compilateur envoie lorsqu'il détecte du code qui effectue des actions qui ne devraient pas être faites. Ce ne sont pas des erreurs car le programme peut tout de même s'exécuter, mais ils méritent en général d'être étudiés et dénotent souvent une instruction mal programmée. Vous devriez toujours utiliser l'option `-Wall` et écrire des programmes qui ne génèrent pas d'alertes.

```
printf("a = %d, b = %lu\n", a, (long unsigned) b);
```

Recompilez avec l'option `-Wall` et vérifiez que le *warning* a disparu.

**Explication :** Beaucoup de choses en C sont fondamentalement représentées par des entiers. Il est alors possible de considérer qu'une valeur d'un certain type est en fait une valeur d'un autre type. Par exemple le caractère `'0'`, qui est le numéro 48 de la table ASCII est représenté par la valeur numérique 48, sur un octet (donc une valeur entre 0 et 255). Ce n'est pas un `int` car il n'est représenté que sur un octet, mais on pourrait très bien décider de le considérer comme un `int`.

Il suffit pour cela dans le programme de préfixer la valeur de la notation « `(int)` » pour indiquer au programme que l'on veut qu'il considère la valeur de la variable comme si c'était un objet de type `int`.

Il en va de même pour les pointeurs qui sont des entiers représentant des adresses mémoire. Une adresse mémoire est fondamentalement un entier positif, codé sur 64 bits. Bien que ce ne soit pas réellement un entier, si l'on préfixe le nom de variable par `(long unsigned)`, on peut demander au programme de faire comme si c'en était un (si l'on ne fait pas la conversion explicite dans le code, le compilateur la fait tout seul mais émet un *warning* pour prévenir que ce comportement n'est peut-être pas voulu).

Comme vous l'aurez deviné, le code `%lu` sert à afficher des entiers de type `long unsigned`.

### 3.2 Tableaux

Les tableaux sont une utilisation très courante des pointeurs. Les éléments d'un tableau sont stockés consécutivement dans la mémoire. Ainsi, pour retrouver les valeurs d'un tableau il suffit de connaître l'adresse du premier élément. Si les objets dans le tableau sont inscrits sur 4 cases mémoire (par exemple des entiers sur 32 bits), et que la variable `tab` est un pointeur vers la première valeur du tableau, alors la seconde valeur du tableau se trouve en position (`tab + 4`), la troisième en position (`tab + 8`), etc.

Ainsi, si le programme sait que le tableau contient des entiers, il peut convertir la notation `tab[2]` en `*(tab + 8)` (l'objet se trouvant 8 cases après le début du tableau).

La déclaration `int tab[10];` demande au programme de réserver l'espace mémoire pour enregistrer 10 entiers consécutifs, et de placer dans la variable `tab` l'adresse du premier emplacement de cette zone réservée.

**12** Si l'on dispose d'un tableau d'entiers déclaré par l'instruction « `int tab[3];` », comment peut on désigner l'adresse où se trouve la deuxième valeur du tableau (celle à l'indice 1) ?

**13** Écrivez un programme qui déclare un tableau d'entiers contenant trois valeurs et qui affiche les adresses de la première et la seconde valeur du tableau. Déduisez-en la taille d'un `int` en mémoire.

**14** En modifiant le programme de la question précédente, déterminez quelle est la taille d'un `float`.

**15** Si l'on exécute le programme suivant, quelle devrait être la différence entre les deux valeurs affichées ?

```
#include <stdio.h>
int main() {
  int t[2];
  printf("Premiere adresse: %d, seconde adresse: %d\n", (int)t, (int)(t+1));
}
```

Copiez, compilez et exécutez le programme. Qu'observez-vous ?

**Arithmétique pointeur**

Le programme de la question 15 montre que si `tab` est un pointeur sur un entier, `(tab+1)` désigne l'adresse qui se trouve 4 octets après `tab`.

Ceci est dû au fait que les opérations arithmétiques (addition et soustraction) sur les pointeurs se font par multiples de la taille prise par le type désigné par le pointeur (si on avait fait un tableau contenant des `double` le décalage serait de 8 octets, et si on prenait des `char` il serait d'un seul octet).

Bien que cela puisse paraître étrange de prime abord, cela permet d'éviter de tomber accidentellement « au milieu » d'une valeur du tableau, et également de trouver la valeur à l'indice $i$ d'un tableau en allant à l'adresse `(tab + i)`. Ainsi, grâce à l'*arithmétique pointeur*, la notation « `tab[i]` » est en réalité un synonyme de « `*(tab+i)` ».

# 4 Expressions

## 4.1 Opérateurs

| | | | |
|---|---|---|---|
| `=` | assignation | `++` | incrémentation |
| `+` | addition | `--` | décrémentation |
| `-` | soustraction | `<op>=` | opération `<op>` et assignation |
| `*` | multiplication | `%` | reste de la division euclidienne (modulo) |
| `/` | division | `()` | parenthèses pour priorités des opérations |

## 4.2 Relations

| | | | |
|---|---|---|---|
| `<` | strictement inférieur | `==` | égal |
| `>` | strictement supérieur | `!=` | différent |
| `<=` | inférieur ou égal | `!` | négation |
| `>=` | supérieur ou égal | `\|\|` | ou logique |
| | | `&&` | et logique |

## 4.3 Opérateurs sur les bits

| | |
|---|---|
| `>>` | décalage à droite |
| `<<` | décalage à gauche |
| `&` | et logique |
| `^` | ou exclusif |
| `\|` | ou logique |
| `~` | négation |

# 5 Instructions

Une instruction doit être terminée par un point-virgule.

On peut délimiter un bloc d'instructions entre accolades (par exemple pour les instructions conditionnelles).

## 5.1 Conditionnelles

```
if (<expression>)
  <instruction>
else
  <instruction>
```

```
switch (<expression>) {
  case <valeur 1>: <instruction>; break;
  case <valeur 2>: <instruction>; break;
  default : <instruction>
}
```

Si l'expression évaluée au début du bloc est égale à l'une des valeurs indiquées par les `case`, alors l'instruction est exécutée. L'instruction `break` permet de sortir du bloc lorsqu'une instruction est exécutée. Si on omet le `break`, l'expression est comparée aux autres valeurs, y compris le cas `default` qui correspond à n'importe quelle valeur.

## 5.2 Les boucles

### 5.2.1 `while`

```
while (<expression>) <instruction>
```

Tant que l'expression est vraie (différente de 0), exécuter l'instruction.

### 5.2.2 `do... while`

```
do <instruction> while (<expression>);
```

Exécuter l'instruction jusqu'à ce que l'expression ne soit plus vraie (c'est très proche du `while` précédent, mais l'instruction est exécutée au moins une fois.

### 5.2.3 `for`

```
for (<instr. initiale>; <test>; <instr. fin>) <instruction>
```

Initialiser la boucle en exécutant `<instr. initiale`, puis tant que `<test>` est vrai, exécuter `<instruction>` suivi de `<instr. fin>`.

Cette boucle est équivalente à

```
<instr. initiale>;
while (<test>) {
  <instruction>
  <instr. fin>
}
```

L'exemple type d'une boucle `for` est :

```
for (i=0; i<10; i++) {
  ...
}
```

qui exécute le bloc d'instructions pour toutes les valeurs de `i` de 0 à 9.

**16** Écrivez un programme qui affiche tous les nombres de 1 à 100.

**17** Écrivez un programme qui affiche tous les nombres entre 1 et 100 qui ne sont ni des multiples de 3 ni des multiples de 7.

**18** Écrivez un programme pour trouver le plus petit entier strictement positif qui est à la fois divisible par 2262 et 13195.

Le programme suivant permet de lire une chaîne de caractères entrée par l'utilisateur au clavier et de l'afficher à l'écran. Pour plus d'information concernant les fonctions utilisées, reportez-vous au manuel (`man fgets` par exemple) :

```
#include <stdio.h>
int main() {
  char chaine[80];
  printf("Entrez une chaîne: ");
  fgets(chaine, 80, stdin);
  fputs(chaine, stdout);
}
```

**19** Sachant qu'une chaîne de caractères se termine par le caractère `'\0'`, complétez le programme pour qu'il calcule la longueur de la chaîne entrée par l'utilisateur et qu'il affiche cette valeur.

**20** Écrivez une fonction qui prend en argument une chaîne de caractères et teste si cette chaîne est un palindrome (la fonction renvoie un entier qui vaut 1 si le mot est un palindrome et 0 sinon).

**Indication :** La fonction renvoie un entier et prend en argument un pointeur vers un caractère (le début d'un tableau de caractères). Son *prototype* est donc

```
int palindrome(char *s)
```

Le programme suivant parcourt un fichier texte nommé « `message.txt` » et affiche son contenu caractère par caractère :

```
#include <stdio.h>

int main() {
  int i;
  FILE *f;
  f = (FILE *) fopen("message.txt","r");
  while (1) {
    i = fgetc(f);
    if (feof(f)) {
      break;
    }
    fputc(i,stdout);
  }
  printf("\n");
  fclose(f);
}
```

**21** Écrivez un message quelconque dans un fichier nommé « `message.txt` », puis testez le programme.

**22** Modifiez le programme pour qu'il effectue une permutation cyclique de toutes les lettres de l'alphabet sur le contenu du fichier `message.txt` : chaque lettre (majuscule et minuscule) est remplacée par la suivante dans l'ordre alphabétique (`'z'` et `'Z'` deviennent `'a'` et `'A'` respectivement).

**Indication :** Souvenez-vous que les caractères sont représentés par l'entier qui les identifie dans la table ASCII. Dans cette table, les lettres sont enregistrées consécutivement dans l'ordre alphabétique.

## 6 Définition de types et structures

Il est possible de définir en C des objets appelés *structures* contenant un certain nombre de *champs* nommés contenant chacun un objet. En un sens, les structures ressemblent à des objets dans un langage orienté objets (par exemple Java) mais n'ayant pas de méthodes (uniquement des attributs).

Par exemple, on peut définir une structure `Personne` contenant une chaîne de caractères `nom`, un entier `âge` et un nombre à virgule flottante `taille` comme ceci :

```
struct Personne {
  char *nom;
  int age;
  float taille;
};
```

La définition précédente permet d'utiliser le type `struct Personne`, c'est-à-dire qu'on peut créer des variables ayant cette spécification. On accède aux différents éléments de la structure par la notation pointée (comme les attributs d'objets en Java) :

```
struct Personne p;
p.nom = "Guybrush";
p.age = 22;
p.taille = 1.75;
```

En mémoire, une structure contient toutes les valeurs de ses champs l'une après l'autre. Dans l'exemple, `p` est une adresse mémoire qui indique où est stocké la structure. Les premiers octets à cette adresse sont un pointeur vers un caractères (l'adresse du premier caractère de la chaîne contenant le nom de la personne). Les 4 octets suivants contiennent l'entier représentant l'âge de la personne, et les 4 derniers octets contiennent le nombre à virgule flottante représentant la taille.

Ainsi, dans l'exemple, la taille complète d'un objet de type `struct Personne` est $8+4+4 = 16$ octets (sur un système en 64 bits, les pointeurs occupent 8 octets).

Vous pouvez vérifier la taille occupée en mémoire par un objet en utilisant la fonction `sizeof` [3] :

```
printf("chaine: %d, entier: %d, flottant: %d, personne: %d\n" %
   sizeof(char*), sizeof(int), sizeof(float), sizeof(struct Personne));
```

3. Il est possible que ce code produise des *warnings* car le type de retour de la fonction `sizeof` est un type nommé `size_t` qui peut être soit un entier non signé soit un entier long non signé. Pour éviter les *warnings* il faudrait *caster* toutes les valeurs renvoyées par `sizeof` en `int`, mais c'est un peu lourd pour les besoins de l'exercice.

**23** À la suite de la définition de la strcture `Personne`, définissez une structure `Famille` contenant 5 champs :

**`nom :`** une chaîne de caractères ;

**`pere :`** une personne ;

**`mere :`** une personne ;

**`nb_enfants :`** un entier ;

**`enfants :`** un tableau de personnes.

**24** Quelle est la taille en mémoire d'un objet de type `struct Famille` ? Vérifiez en utilisant la fonction `sizeof`.

**25** Écrivez une fonction `age_total` qui prend en argument un objet de type `struct Famille` et renvoie un entier correspondant à la somme des âges de tous les membres de la famille.

**Indication :** Il n'existe pas de moyen en C d'obtenir la longueur d'un tableau en ne connaissant que l'adresse du premier élément. Il faut donc mémoriser la taille des tableaux dans des variables. C'est à ça que sert le champ `nb_enfants`, et on peut donc supposer que le tableau `enfants` a exactement `nb_enfants` cases.

**26** Créez une variable contenant une famille ayant deux enfants. Vérifiez que la fonction `age_total` marche bien.