

Année académique 2012–2013



# Informatique appliquée aux sciences et aux technologies

## Bases de la programmation

## Table des matières

<b>Avant-propos</b>	<b>5</b>
<b>1 Introduction</b>	<b>6</b>
<b>2 Types de données, variables et opérateurs</b>	<b>7</b>
2.1 Types de données et variables . . . . .	7
2.1.1 Nombres entiers . . . . .	7
2.1.2 Nombres flottants . . . . .	8
2.1.3 Caractères . . . . .	9
2.1.4 Affichage avec <code>printf</code> . . . . .	9
2.2 Opérateurs . . . . .	10
2.2.1 Opérateurs arithmétiques . . . . .	10
2.2.2 Opérateurs de comparaison . . . . .	10
2.2.3 Opérateurs logiques . . . . .	11
<b>3 Structures de contrôle</b>	<b>12</b>
3.1 Instructions conditionnelles . . . . .	12
3.1.1 Enchaînement de plusieurs conditions . . . . .	12
3.2 Instructions répétitives . . . . .	13
3.2.1 La boucle <code>while</code> . . . . .	13
3.2.2 La boucle <code>for</code> . . . . .	14
<b>4 Tableaux</b>	<b>15</b>
4.1 Déclaration d'un tableau . . . . .	15

4.2	Accès aux cellules d'un tableau . . . . .	15
4.3	Parcours de tableau . . . . .	15
4.4	Initialisation d'un tableau . . . . .	16
4.5	Affichage d'un tableau . . . . .	16
<b>5</b>	<b>Procédure et fonction</b>	<b>17</b>
5.1	Procédures . . . . .	17
5.1.1	Procédures avec paramètres . . . . .	18
5.2	Fonctions . . . . .	18
5.3	Organisation de la mémoire . . . . .	19
<b>6</b>	<b>Pointeur et mémoire dynamique</b>	<b>23</b>
6.1	Introduction aux pointeurs . . . . .	23
6.2	Tas ou mémoire dynamique . . . . .	24
6.3	Tableaux dynamiques . . . . .	25
6.4	Fonction pour créer un tableau . . . . .	26
<b>7</b>	<b>Structure</b>	<b>27</b>
7.1	Définir une nouvelle structure . . . . .	28
7.2	Structure en mémoire dynamique . . . . .	30
7.3	Structure représentant un tableau . . . . .	31
7.4	Structure de structure . . . . .	32
<b>8</b>	<b>Organisation d'un programme en fichiers et compilation</b>	<b>34</b>
8.1	Prototype de fonction . . . . .	34
8.2	Fichier d'entête et librairie . . . . .	35

8.3	Chaine de compilation . . . . .	37
8.3.1	Compilation . . . . .	37
8.3.2	Linking . . . . .	38
8.4	Utiliser le compilateur <code>gcc</code> . . . . .	38

## Avant-propos

Ce syllabus couvre la matière vue au cours de bases de la programmation dispensé aux étudiants en deuxième année du bachelier en informatique et systèmes : réseaux et télécoms et du bachelier en électronique appliquée. Ce cours aborde les bases de la programmation en utilisant le langage de programmation C.

Ce syllabus est distribué sous licence Creative Commons BY-NC-SA, c'est-à-dire que vous êtes libre de le distribuer et de l'adapter pour autant que l'auteur original soit clairement mentionné, que vous le redistribuiez sous la même licence (ou sous une autre licence similaire) et que vous ne pouvez pas l'utiliser dans un but commercial.

Pour toute question relative au présent syllabus, ou pour signaler toute erreur ayant pu s'y glisser, vous pouvez contacter directement Sébastien Combéfis par e-mail à l'adresse suivante : [sebastien@combefis.be](mailto:sebastien@combefis.be).

# 1 Introduction

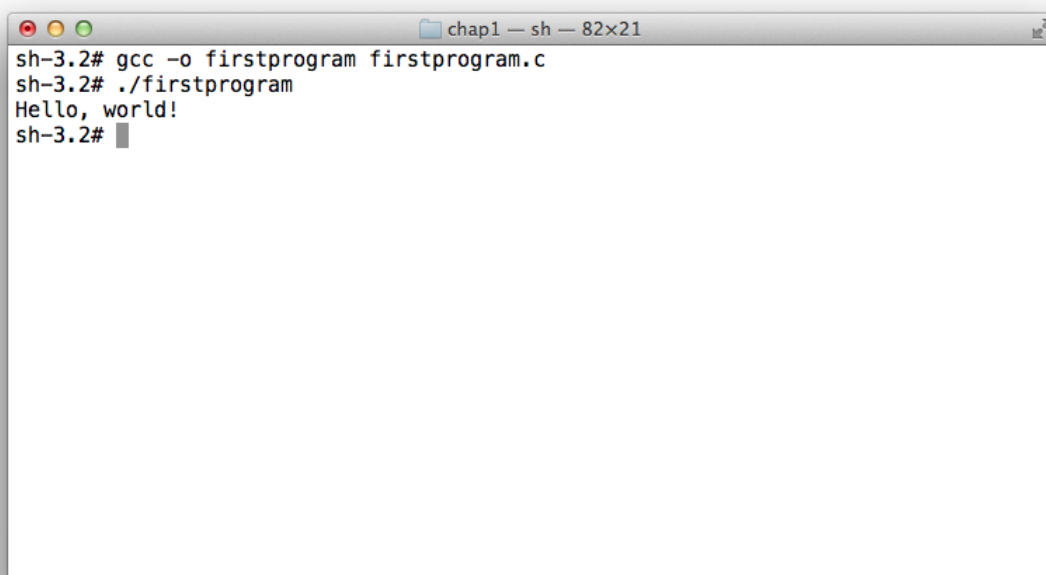
Commençons tout de suite par un exemple concret de programme complet. L'exemple suivant est le classique programme *Hello, world!* utilisé pour montrer la syntaxe d'un langage de programmation.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf ("Hello, world!");
6
7     return 0;
8 }
```

Passons en revue les différentes parties de ce programme :

- Tout d'abord la première ligne permet d'inclure une librairie existante. Inclure une librairie permet d'utiliser toute une série de fonctionnalités déjà existantes, sans devoir les recoder.
- Les lignes 3 à 8 correspondent à la définition de la fonction principale du programme (appelée **main**). Il s'agit du point d'entrée du programme, l'exécution de ce dernier commençant donc à la ligne 5.
- La ligne 3 déclare le début de la fonction **main** et vient ensuite le corps de la fonction, placé entre accolades, des lignes 4 à 8.
- La ligne 5 fait appel à la fonction **printf** (qui se trouve dans la librairie **stdio**), dont le but est d'afficher une phrase à l'écran, celle écrite entre guillemets, le tout entre parenthèses.
- Enfin, la ligne 7 indique la fin de la fonction principale.

La figure 1 montre la compilation et l'exécution de ce premier programme d'exemple. Ne vous inquiétez pas si vous ne comprenez pas tout pour l'instant, on aura l'occasion de revenir en détails sur cela plus tard. L'important est que vous voyiez que la phrase **Hello, world!** a bien été affichée à l'écran.



```
sh-3.2# gcc -o firstprogram firstprogram.c
sh-3.2# ./firstprogram
Hello, world!
sh-3.2#
```

FIGURE 1. Compilation et exécution du programme *Hello, world!*.

## 2 Types de données, variables et opérateurs

Lorsqu'on programme, une chose que l'on doit faire à tout moment consiste à manipuler des données. Ces données vont devoir être stockées dans la mémoire de l'ordinateur. Chaque donnée possède un certain type et en fonction de ce dernier, différentes opérations vont pouvoir être faites.

Par exemple, supposez que vous êtes en train d'écrire un programme pour gérer les stocks d'un magasin. Pour ce faire, il vous faudra notamment stocker la quantité restante de sachets de chips paprika 700g. Cette valeur est de type « *un nombre entier positif* ». Les opérations que vous pourrez faire sur cette valeur est de lui ajouter ou retirer une certaine quantité, par exemple ajouter dix sachets de chips ou en retirer trois.

### 2.1 Types de données et variables

Il existe de nombreux types de données, mais on va ici se limiter à quelques types de base. On retrouve essentiellement trois types de données :

- Nombres entiers
- Nombres réels
- Caractères

Pour manipuler des données, il faut donc les stocker dans la mémoire de l'ordinateur. Pour pouvoir interagir avec une zone mémoire, on passe par les *variables*. Une variable représente une zone dans la mémoire de l'ordinateur. Pour pouvoir l'utiliser, on doit lui donner un nom et un type.

Prenons un exemple :

```
int age;  
age = 28;
```

Deux choses sont faites dans cet extrait de code. Tout d'abord, une nouvelle variable est déclarée : son nom est `age` et son type est `int` (nombre entier). La seconde ligne de code stocke la valeur 28 dans la variable `age`. Une fois qu'une variable est initialisée, c'est-à-dire qu'elle a reçu sa première valeur, on peut modifier sa valeur à tout moment, par exemple :

```
age = 42;
```

#### 2.1.1 Nombres entiers

En ce qui concerne les nombres entiers, il y a trois types de base : `int`, `short` et `long`. La différence entre ces trois types est la place qui est utilisée en mémoire pour stocker leurs valeurs. Afin de connaître cette taille occupée en mémoire, on peut utiliser l'opérateur `sizeof`. Voici un exemple de programme qui affiche à l'écran le nombre d'octets occupés pour chacun des trois types :

```
printf ("Un int occupe %ld octets en mémoire\n", sizeof (int));  
printf ("Un short occupe %ld octets en mémoire\n", sizeof (short));  
printf ("Un long occupe %ld octets en mémoire\n", sizeof (long));
```

Ne faites pas attention pour l'instant à la syntaxe du `printf`, on y reviendra à la fin de ce chapitre. L'exécution de ce programme affiche par exemple :

```
Un int occupe 4 octets en mémoire
Un short occupe 2 octets en mémoire
Un long occupe 8 octets en mémoire
```

Prenons par exemple le type `short`. Il est encodé sur deux octets, à savoir seize bits. Ainsi, avec le type `short`, on peut représenter  $2^{16} = 65536$  valeurs différentes. Il y a maintenant deux possibilités :

- On se limite aux valeurs positives et on peut représenter les entiers compris entre 0 et 65535 ;
- On autorise des valeurs négatives et on se retrouve alors entre  $-32768$  et  $32767$ .

Par défaut, les valeurs négatives sont autorisées et on dit que le type est *signé*. On peut explicitement le signaler si on le souhaite, grâce au mot réservé `signed`, par exemple :

```
signed int price;
```

Si on ne désire pas autoriser les valeurs négatives, il suffit d'utiliser un type *non signé* qui se déclare avec le mot réservé `unsigned`, par exemple :

```
unsigned int age;
```

Le tableau 1 reprend les valeurs possibles pour les trois types nombres entiers. Notez évidemment que les tailles réservées en mémoire pour ces différents types peuvent dépendre de la machine. Ici, les tests ont été effectués sur un MacBook Pro 64 bits.

Type	Taille	Signé	Non signé
<code>short</code>	2 octets	0 à 65 535	$-32\,768$ à $32\,767$
<code>int</code>	4 octets	0 à 4 294 967 295	$-2\,147\,483\,648$ à $2\,147\,483\,647$
<code>long</code>	8 octets	0 à 18 446 744 073 709 551 616	$-9\,223\,372\,036\,854\,775\,808$ à $9\,223\,372\,036\,854\,775\,807$

TABLE 1. Valeurs possibles pour les trois types de données nombres entiers.

### 2.1.2 Nombres flottants

Il n'est pas possible de représenter tous les nombres réels en C. Ce qu'on peut faire, c'est représenter un sous-ensemble de tous les nombres réels. On appelle ces nombres les *nombres flottants*. Il y a deux types de données en C pour ces nombres : `float` et `double`. Le premier type occupe quatre octets en mémoire tandis que le second en occupe huit. On pourrait par exemple déclarer :

```
float pi = 3.14;
```

Contrairement aux nombres entiers, les nombres flottants sont d'office signés. Le tableau 2 reprend les valeurs possibles pour chacun des deux types de nombres flottants.

Type	Taille	Valeurs possibles
<code>float</code>	4 octets	$3,4 \cdot 10^{-38}$ à $3,4 \cdot 10^{38}$
<code>double</code>	8 octets	$1,7 \cdot 10^{-308}$ à $1,7 \cdot 10^{308}$

TABLE 2. Valeurs possibles pour les trois types de données nombres flottants.



### 2.1.3 Caractères

Les caractères sont représentés par le type `char` qui occupe un octet en mémoire. Cela signifie qu'on peut stocker 256 caractères différents dans une variable de type `char`. Le principe est que chaque caractère est associé à une valeur entière. Le tableau 3 montre quelques exemples de caractères avec la valeur entière associée.

Caractère	!	/	5	D	g
Valeur associée	33	47	53	68	103

**TABLE 3.** Quelques exemples de caractères avec la valeur entière leur étant associée.

Les caractères s'écrivent entre guillemets simples. Par exemple, voici une variable qui contient le caractère parenthèse ouvrante.

```
char c = '(';
```

### 2.1.4 Affichage avec printf

Comme on l'a déjà vu dans des exemples précédents, on peut utiliser `printf` pour afficher du texte à l'écran. Pour faire un affichage, on utilise une suite de caractères placée entre guillemets doubles. Dans cette suite de caractères, vous pouvez utiliser des marqueurs spéciaux qui sont constitués du caractère `%` suivi d'une série de caractères. Il suffit ensuite de placer les valeurs à la suite. Prenons tout de suite un exemple :

```
printf ("Un entier : %d, un flottant : %f et enfin un caractère : %c\n", 12, 0.99, 'k');
```

L'exécution de cette instruction affiche à l'écran :

```
Un entier : 12, un flottant : 0.990000 et enfin un caractère : k
```

On peut donc voir qu'il y a trois marqueur dans la chaîne de caractères : `%d`, `%f` et `%c` et on a également fourni trois valeurs : 12, 0.99 et 'k'. Il faut évidemment toujours autant de valeurs qu'il y a de marqueurs. De plus, chaque marqueur correspond à un type de données particulier et il faut également que ces types correspondent. Le tableau 4 reprend quelques marqueurs avec le type de données correspondant.

Marqueur	Type de données
<code>%d</code>	short ou int
<code>%ld</code>	long
<code>%f</code>	float ou double
<code>%c</code>	char

**TABLE 4.** Valeurs possibles pour les trois types de données nombres flottants.

## 2.2 Opérateurs

Maintenant qu'on est capable de stocker des valeurs, on va voir comment faire des opérations entre plusieurs valeurs. Ceci nous permettra notamment de faire des calculs. On peut classer les opérateurs en plusieurs catégories.

### 2.2.1 Opérateurs arithmétiques

Commençons avec les opérateurs arithmétiques qui sont au nombre de cinq. Ils sont repris dans le tableau 5.

Opérateur	Description	Exemple
-	Changement de signe	-44
+	Addition	44 + 12
-	Soustraction	44 - 12
*	Multiplication	44 * 12
/	Division	44 / 12
%	Reste de la division entière	44 % 12

**TABLE 5.** Opérateurs arithmétiques.

Les premiers opérateurs sont assez classiques. Il faut porter une attention particulière aux deux derniers dont l'effet sera différent selon les valeurs qui sont fournies.

Commençons avec le cas où les deux valeurs sont des entiers. Dans ce cas, l'opérateur / calcule la division entière et % calcule le reste de cette division. Par exemple, si on divise 17 par 5, ça donne 3 avec un reste de 2 puisque  $17 = 3 \times 5 + 2$ . Donc, la valeur de  $17 / 5$  est de 3 et celle de  $17 \% 5$  est de 2.

Si les deux valeurs ne sont pas entières, alors l'opérateur / calcule la division classique dont le résultat est un flottant. Ainsi, si on prend  $17 / 3.0$  ou  $17.0 / 3$  ou encore  $17.0 / 3.0$ , la valeur sera de 5.666666... Vous remarquerez qu'il suffit qu'au moins une des deux valeurs soit un flottant pour que le résultat de la division soit un flottant.

### 2.2.2 Opérateurs de comparaison

On vient de voir comment faire des calculs avec des nombres. Il est également possible de comparer différentes valeurs. Le tableau 6 reprend les six opérateurs de comparaison.

Opérateur	Description	Exemple
>	Strictement plus grand	44 > 12
>=	Plus grand ou égal	44 >= 12
<	Strictement plus petit	44 < 12
<=	Plus petit ou égal	44 <= 12
==	Égal	44 == 12
!=	Différent	44 != 12

**TABLE 6.** Opérateurs de comparaison.

Ces opérateurs de comparaison permettent en fait de construire des expressions booléennes. C'est-à-dire des expressions dont la valeur est soit vraie, soit fausse. Comme on le verra dans le chapitre suivant, on va les utiliser comme conditions dans les structures de contrôle.

### 2.2.3 Opérateurs logiques

On vient de voir les opérations de comparaison grâce auxquelles on peut maintenant construire des conditions basiques. Il est possible de construire des conditions plus complexes grâce aux opérateurs logiques. Ces opérateurs sont au nombre de trois et sont repris dans le tableau 7.

Opérateur	Description	Exemple
!	NON logique	! (x > 8)
&&	ET logique	(x > 8) && (x < 10)
	OU logique	(x == 8)    (x == 10)

**TABLE 7.** Opérateurs logiques.

Le premier opérateur permet d'inverser une condition. Le deuxième opérateur permet de s'assurer que les deux conditions liées doivent toutes les deux être vraies. Enfin, le troisième opérateur permet de s'assurer qu'au moins une des deux conditions liées est vraie.

Imaginons par exemple qu'on a une variable `temperature` représentant la température. On veut écrire une condition qui soit vraie si la température est comprise entre 10 et 20 degrés (bornes incluses) :

```
(temperature >= 10) && (temperature <= 20)
```

## 3 Structures de contrôle

Grâce au chapitre précédent, on est maintenant capable de stocker des valeurs dans des variables et de faire des opérations avec ces dernières. Tant qu'à présent, un programme se limite à une suite d'instructions qui s'exécutent les unes après les autres. Dans ce chapitre, on va voir les *structure de contrôle* dont le but est de « *casser* » cette exécution linéaire.

### 3.1 Instructions conditionnelles

Parfois, on souhaiterait qu'une partie de code ne soit exécutée que si une certaine condition est remplie. Pour ce faire, on peut utiliser une instruction conditionnelle. Une telle instruction s'introduit avec le mot réservé `if`. Ne tardons pas et examinons tout de suite un exemple :

```
if (temperature >= 37.6)
{
    printf ("Attention, vous avez de la fièvre !\n");
}
```

On commence donc avec le mot réservé `if` suivi d'une condition entre parenthèse. Dans notre exemple, la condition stipule que la valeur de la variable `temperature` est plus grande ou égale à 37.6. Si la condition est satisfaite, alors le code situé entre accolades sera exécuté. Dans notre exemple, un message d'alerte indique que vous avez de la fièvre.

On peut également vouloir exécuter une partie de programme si une condition est satisfaite, et une autre partie de programme si la condition n'est pas satisfaite. Pour ce faire, on va utiliser un `if-else`. Voyons toute de suite ça en exemple :

```
if (grade >= 10)
{
    printf ("Examen réussi, bravo !\n");
}
else
{
    printf ("Examen raté, il va falloir se reprendre !\n");
}
```

Dans cet exemple, la condition testée est que la valeur de la variable `grade` doit être plus grande ou égale à 10. Si la condition est vraie, le code attaché au `if` sera exécuté. Par contre, si la condition n'est pas vraie, ce sera le code attaché au `else` qui sera exécuté.

#### 3.1.1 Enchaînement de plusieurs conditions

Si l'on souhaite enchaîner plusieurs conditions, il faut utiliser un `if-else if-else`. Prenons un exemple où on veut calculer le prix d'entrée d'un parc d'attractions, ce prix dépendant de l'âge de la personne. Le tableau ci-dessous reprend la grille de prix en fonction de l'âge :

Tranche d'âge	< 4 ans	Entre 4 ans et 12 ans	Entre 13 et 25 ans	> 25 ans
Prix	gratuit	7 euros	10 euros	15 euros

Voici donc le programme qui va permettre de calculer le prix qu'il faut payer en fonction de l'âge :

```

if (age < 4)
{
    price = 0;
}
else if (age >= 4 && age <= 12)
{
    price = 7;
}
else if (age >= 13 && age <= 25)
{
    price = 10;
}
else
{
    price = 15;
}

```

Notez que le dernier **else** n'est évidemment pas obligatoire.

## 3.2 Instructions répétitives

Un autre comportement intéressant est celui de la répétition de code. Soit on désire répéter un code tant qu'une certaine condition est vraie, ou alors on peut également vouloir répéter un code un certain nombre précis de fois.

### 3.2.1 La boucle while

Commençons avec un exemple simple qui va diviser un nombre entier par deux, jusqu'à atteindre 1. Il faut compter le nombre de fois qu'on a fait la division. Voici le code :

```

1 int counter = 0;
2 while (n != 1)
3 {
4     n = n / 2;
5     counter++;
6 }
7 printf ("Le nombre a été divisé %d fois par 2", counter);

```

Décortiquons cet exemple. La première ligne déclare une variable **counter** initialisée à zéro. Cette variable va permettre de compter combien de fois on a fait la division par deux. Ensuite, on attaque la boucle grâce au mot réservé **while** suivi d'une condition. Tant que la condition est vraie, on va répéter le code de la boucle. Donc, dans notre cas, on veut continuer tant que **n** est différent de 1.

Que fait-on à chaque boucle ? Et bien, à la ligne 4 on met à jour la valeur de **n** en la divisant par deux et à la ligne 5, on augmente la valeur du compteur de une unité. L'instruction **counter++**; est en effet équivalente à **counter = counter + 1**;

### 3.2.2 La boucle for

Lorsqu'on veut faire un nombre précis de boucles, on va préférer utiliser la boucle **for**. Son fonctionnement est similaire à celui de la boucle **while**, mais se base sur une variable qu'on peut appeler « *compteur* ». Prenons un exemple :

```
1 int i;  
2 for (i = 0; i < 10; i++)  
3 {  
4     // ...  
5 }
```

La première ligne déclare une nouvelle variable **i** de type **int**. Il s'agit du compteur. Ensuite, vient le début de la boucle avec le mot réservé **for**. Trois éléments sont à définir entre parenthèses, et séparés par des point-virgules. Le premier élément (**i = 0**) est l'initialisation exécutée une fois tout au début. La seconde partie (**i < 10**) est la condition et fonctionne de manière similaire aux boucles **while**. Enfin, la dernière partie (**i++**) est exécutée à chaque fois après exécution du corps de la boucle. Pour mieux comprendre, voici la même boucle réécrite avec l'instruction **while**.

```
int i;  
i = 0;  
while (i < 10)  
{  
    // ...  
  
    i++;  
}
```

La variable **i** est appelée compteur étant donné que sa valeur va démarrer à zéro et augmenter de 1 à chaque boucle, et cela tant que sa valeur reste strictement inférieure à 10. Donc, la variable **i** va valoir 0, 1, 2, ..., 9. On aurait évidemment pu écrire la condition de la boucle comme **i <= 9**. Une autre solution aurait pu être de commencer à 1 (**i = 1**) et finir à 10 (**i <= 10**). On verra dans le chapitre suivant pourquoi on préfère commencer le compteur à zéro.

## 4 Tableaux

Jusqu'à présent, on s'est contenté d'utiliser des variables pour stocker des valeurs. Mais dans une variable, on ne peut stocker qu'une seule valeur. Imaginez par exemple qu'on souhaite stocker plusieurs valeurs, il faudrait alors déclarer autant de variables que de valeurs qu'on souhaite stocker. Ceci n'est évidemment pas pratique du tout.

### 4.1 Déclaration d'un tableau

Prenons un exemple concret. Imaginons que l'on souhaite stocker la moyenne des températures pour chacun des mois de l'année 2012. On aurait besoin de douze variables pour cela. Mais une autre solution est possible, on va utiliser ce qu'on appelle un tableau.

```
float temperature[12];
```

Cette ligne de code déclare un nouveau tableau de `float`, que l'on peut utiliser avec la variable `temperature`. Le tableau possède 12 cases, on dit que sa taille est de 12. C'est comme si on avait déclaré 12 nouvelles variables `float`.

### 4.2 Accès aux cellules d'un tableau

On va pouvoir accéder à chacune des cellules du tableau grâce à une notation particulière. On utilise le nom de la variable, suivi de l'indice de la cellule qui nous intéresse, écrit entre crochets. Attention, il faut savoir que la première cellule porte l'indice 0.

Voici par exemple comment mettre la valeur 88 dans la première cellule du tableau :

```
temperature[0] = 88;
```

Voici un autre exemple où on affiche la valeur de la 4<sup>e</sup> cellule (celle d'indice 3 donc) :

```
printf ("La valeur est : %d\n", temperature[3]);
```

### 4.3 Parcours de tableau

On peut maintenant faire ce qu'on veut avec les éléments d'un tableau. Typiquement, lorsqu'on veut parcourir les différents éléments d'un tableau, on va utiliser une boucle `for`. Voyons par exemple comment faire la somme des éléments d'un tableau.

```
float sum = 0;
int i;
for (i = 0; i < 12; i++)
{
    sum = sum + temperature[i];
}
```

## 4.4 Initialisation d'un tableau

Par défaut, lorsqu'on crée un nouveau tableau, chacune de ses cellules ne sont pas initialisées. On ne sait donc pas prédire la valeur qu'elles contiennent. Donc, avant d'utiliser un tableau, il va falloir initialiser toutes ses cases. Une autre manière de faire est de donner directement les valeurs du tableau, comme sur l'exemple suivant :

```
int tab[5] = {4, 9, -2, 0, 1};
```

## 4.5 Affichage d'un tableau

Contrairement aux types de données qu'on a vu jusqu'à présent, il n'y a pas moyen d'afficher les éléments d'un tableau facilement. Pour ce faire, on est obligé de faire une boucle, pour afficher les éléments un à un. Voici le genre d'affichage qu'on aimerait avoir :

```
[]           // pour un tableau vide
[7]          // pour un tableau à un seul élément
[7, 3, -2]   // pour un tableau contenant plus d'un élément
```

Si on vous donne une variable de type tableau, il n'y a pas moyen de connaître sa taille, de manière générale. Donc, ce qu'on fait souvent, c'est associer une variable de type `int` contenant la taille du tableau. Voyons maintenant comment faire l'affichage d'un tableau `tab` de taille `N`.

```
1  if (N == 0)
2  {
3      printf ("[]\n");
4  }
5  else
6  {
7      printf ("[%d", tab[0]);
8
9      int i;
10     for (i = 1; i < N; i++)
11     {
12         printf (" , %d", tab[i]);
13     }
14
15     printf ("]\n");
16 }
```

Ce code est assez simple à comprendre. Premièrement, si le tableau est vide, on affiche directement `[]`. Ensuite, si on arrive dans le `else` de la ligne 5, on est sûr que le tableau contient au moins un élément. La ligne 7 affiche le crochet ouvrant suivi de la première valeur du tableau.



## 5 Procédure et fonction

Jusqu'à présent, nos programmes se limitaient à une fonction `main` et l'intégralité du programme s'y trouvait. Ceci n'est évidemment pas pratique pour diverses raisons dont la complexité de lecture et compréhension du code, sa maintenabilité difficile et enfin la duplication de code et la difficulté le réutiliser.

### 5.1 Procédures

Commençons avec le cas le plus simple : les procédures. Une procédure est une série d'instructions que l'on rassemble et à qui on donne un nom. Grâce à cet unique nom, il va être possible à tout moment d'appeler la procédure, à savoir exécuter intégralement ses instructions. Prenons un exemple :

```
void countdown()
{
    int i;
    for (i = 3; i > 0; i--)
    {
        printf ("%d... ", i);
    }
    printf ("0\n");
}
```

Une procédure se déclare donc en commençant avec le mot réservé `void` suivi d'un nom et de deux parenthèses. Viennent ensuite les instructions de la procédure. Dans notre exemple, il s'agit d'un compteur qui va afficher : « 3... 2... 1... 0 ». Voyons maintenant comment organiser tout cela dans un programme complet :

```
#include <stdio.h>

void countdown()
{
    int i;
    for (i = 3; i > 0; i--)
    {
        printf ("%d... ", i);
    }
    printf ("0\n");
}

int main()
{
    countdown();
    countdown();

    return 0;
}
```

On voit donc qu'on déclare d'abord la procédure `countdown` avant la fonction `main`. La règle à respecter est qu'il faut toujours déclarer une procédure avant de l'utiliser. Et comme ici on fait appel, deux fois, à la procédure `countdown` dans la fonction `main`, il a fallu la déclarer avant. L'exécution de ce programme affiche à l'écran :

```
3... 2... 1... 0
3... 2... 1... 0
```

### 5.1.1 Procédures avec paramètres

On voit donc tout doucement apparaître le grand avantage des procédures. On évite la répétition de code et on définit un ensemble d'instructions à qui on donne un nom et que l'on peut réutiliser partout où on le souhaite. Maintenant, la forme qu'on vient de voir reste assez limitée. On va voir qu'il est possible de passer des paramètres à une procédure. Pour ce faire, on les déclare simplement entre les parenthèses de la déclaration de la procédure.

Revoyons notre exemple pour permettre de démarrer le décompte à n'importe quelle valeur choisie, et pas seulement à la valeur trois.

```
void countdown (int startValue)
{
    int i;
    for (i = startValue; i > 0; i--)
    {
        printf ("%d... ", i);
    }
    printf ("0\n");
}
```

On voit donc que la seule différence est qu'on a introduit un paramètre de type `int` nommé `startValue` (c'est en réalité une simple variable) et qu'on a changé la boucle `for` afin qu'elle démarre à `startValue` au lieu de commencer à trois comme précédemment. Maintenant, lorsqu'on souhaite appeler cette procédure, il faudra évidemment fournir une valeur au paramètre `startValue`. Cela se fait tout simplement lors de l'appel de la procédure :

```
int main()
{
    countdown (5);
    countdown (3);
    countdown (10);

    return 0;
}
```

L'exécution de ce programme affiche à l'écran :

```
5... 4... 3... 2... 1... 0
3... 2... 1... 0
10... 9... 8... 7... 6... 5... 4... 3... 2... 1... 0
```

## 5.2 Fonctions

Les procédures permettent d'exécuter une séquence d'instructions, mais il n'y a aucun retour vers l'appelant. En effet, si on reprend notre exemple, la fonction `main` appelle la procédure `countdown`, mais une fois que `countdown` a fini de s'exécuter, l'exécution revient simplement dans `main`.

Les fonctions sont comme les procédures, à part qu'elles vont pouvoir renvoyer une valeur, une fois leur exécution terminée. Voyons un exemple concret. On désire pouvoir retrouver la plus grande valeur parmi deux nombres entiers donnés.

Pour cela on définit donc la fonction suivante :

```
int getMax (int a, int b)
{
    if (a > b)
    {
        return a;
    }
    else
    {
        return b;
    }
}
```

La première ligne est similaire aux procédures, si ce n'est qu'on n'utilise plus `void` mais à la place, on met un type de données. Ce type correspond au type de la valeur de retour qui sera renvoyée par la fonction, dans notre cas, ce sera un `int`. Vient ensuite le corps de fonction où on va tester avec un `if` si `a` est plus grand que `b`. Si c'est le cas, la fonction s'arrête et sa valeur de retour est `a`, ce qu'on indique grâce au `return a;`. Si `a` n'est pas plus grand que `b`, on se retrouve dans le `else` et dans ce cas, la valeur renvoyée sera `b`, ce qu'on indique avec `return b;`.

Lorsqu'on appelle une fonction, contrairement aux procédures, il va falloir récupérer la valeur de retour qui est renvoyée par la fonction. Pour ce faire, il suffit de déclarer une variable du bon type de d'y stocker l'appel à la fonction. Voyons par exemple comment faire pour retrouver la plus grande valeur entre `-12` et `34` :

```
int main()
{
    int max = getMax (-12, 34);
    printf ("La plus grande valeur est : %d\n", max);

    return 0;
}
```

Comme vous avez pu le remarquer, on peut donc également définir des fonctions qui prennent des paramètres. En réalité, la seule différence entre une procédure et une fonction est que les fonctions renvoient une valeur de retour lorsqu'elles ont fini de s'exécuter. Souvent on décrit les procédures comme des routines dont le but est d'exécuter une action et les fonctions comme étant des routines dont le but est de calculer le résultat d'une opération.

Comme on le verra plus loin, il existe déjà de nombreuses procédures et fonctions prédéfinies et qu'on va pouvoir utiliser. D'ailleurs, on en utilise déjà une depuis le tout début de ce syllabus, à savoir la procédure `printf`. De plus, depuis le début de ce syllabus, on a déjà également défini une fonction, à savoir la fonction `main`.

### 5.3 Organisation de la mémoire

Pour mieux comprendre le fonctionnement des procédures et des fonctions, et surtout pour introduire le chapitre suivant, il nous faut nous intéresser de plus près à l'organisation en mémoire des différentes variables, et de ce qui se passe lors d'un appel de procédure ou de fonction.

Partons de l'exemple suivant :

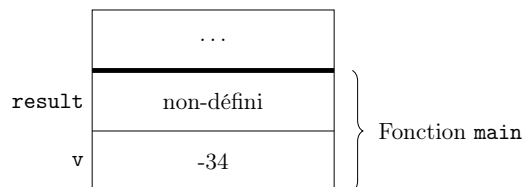
```

1 #include <stdio.h>
2
3 int getAbsValue (int value)
4 {
5     int absValue = value;
6     if (value < 0)
7     {
8         absValue = -value;
9     }
10    return absValue;
11 }
12
13 int main()
14 {
15     int v = -34;
16     int result = getAbsValue (v);
17     printf ("La valeur absolue de -34 vaut : %d\n", result);
18
19     return 0;
20 }

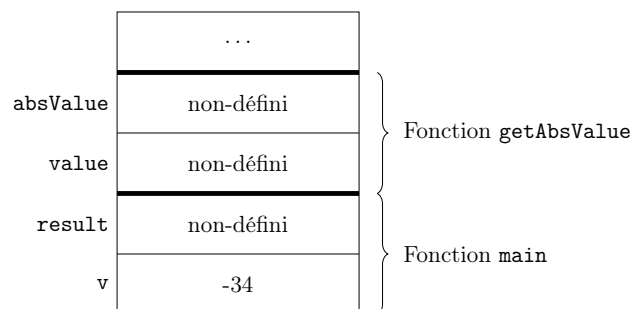
```

À tout moment dans la mémoire de l'ordinateur, un bloc de mémoire est réservé à chaque fois qu'on rentre dans une nouvelle procédure ou fonction. Cette zone de mémoire sert à accueillir toutes les variables déclarées dans la procédure ou fonction. Cette mémoire est appelée la *pile* et chaque zone réservée pour une procédure ou fonction est appelée *environnement*.

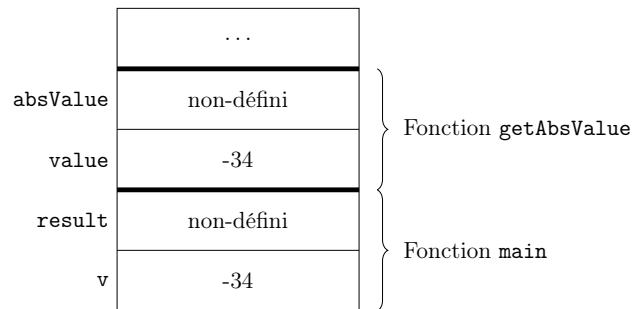
Initialement, on entre donc dans la fonction `main` qui comporte deux variables `v` et `result`, comme le montre le dessin suivant :



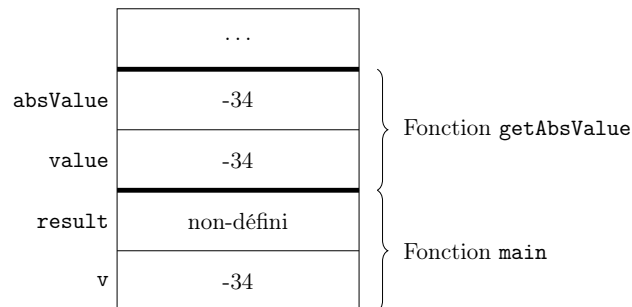
Une fois arrivé à la ligne 16, un appel de fonction va démarrer. Un nouvel environnement est créé dans la mémoire, pour préparer l'exécution de la fonction `getAbsValue`. Deux variables sont nécessaires pour cette fonction : le paramètre `value` et la variable `absValue`. La mémoire est donc modifiée et ressemble maintenant à :



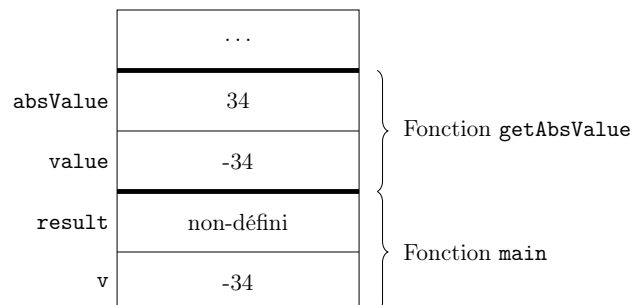
Lors de l'appel, les valeurs données aux paramètres lors de l'appel vont être copiées dans les paramètres de la méthode. Donc, comme l'appel est `getAbsValue (v)`, la valeur de la variable `v` de la fonction `main` va être copiée dans la variable `value`, paramètre de la fonction `getAbsValue`. La mémoire devient donc :



On en arrive ensuite à l'instruction de la ligne 5 qui copie la valeur de la variable `value` dans la nouvelle variable `absValue`. On se retrouve donc avec :



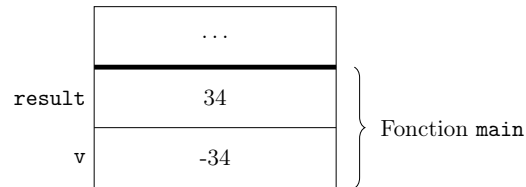
Ensuite, on arrive à l'instruction `if` qui teste si la valeur de `value` est strictement négative. Comme c'est bien le cas, on rentre dans le `if` et on exécute donc `absValue = -value;`. La variable `absValue` est donc mise à jour en mémoire et on obtient :



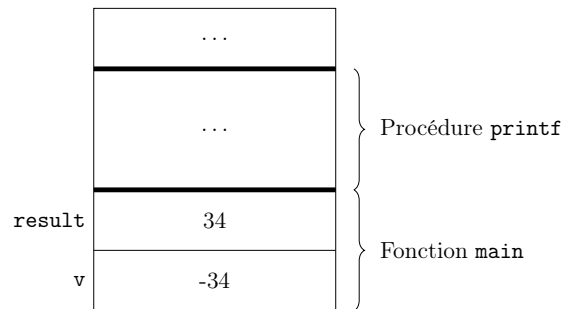
La fonction `getAbsValue` a maintenant fini de s'exécuter et renvoie comme résultat la valeur de la variable `absValue` (à savoir 34). Notez que toute la mémoire déclarée pour la fonction est maintenant supprimée et donc, les variables `value` et `absValue` de la fonction n'existent plus. On revient donc dans

la fonction `main` et la valeur renvoyée par la fonction `getAbsValue` est copiée dans la variable `result` de la fonction `main`.

La mémoire se trouve donc maintenant dans l'état suivant :



Pour la suite de l'exécution, on fait appel à la procédure `printf` et donc, un nouvel environnement est créé et on recommence...



## 6 Pointeur et mémoire dynamique

On vient de voir, à la fin du chapitre précédent, que lorsqu'une procédure ou fonction est appelée, une fois qu'elle a fini de s'exécuter, toute la mémoire qui lui était réservée est simplement supprimée. Cela peut être déroutant dans certaines situations. Par exemple, supposez que l'on souhaite écrire une fonction dont le but est de créer un nouveau tableau d'une certaine taille fixée, et de le remplir avec des zéros. On pourrait se dire qu'il suffit d'écrire le code suivant :

```
int[] newTab (int N)
{
    int tab[N];
    int i;
    for (i = 0; i < N; i++)
    {
        tab[i] = 0;
    }
    return tab;
}
```

Mais ceci ne fonctionnera pas. En fait, le code ne compilera même pas. La raison est simple, le tableau `tab` créé dans cette fonction sera supprimé de la mémoire une fois la fonction terminée. Donc, faire un `return tab;` n'a pas de sens, puisqu'on renverrait quelque chose qui n'existe plus. Pour résoudre notre problème, on ne va donc pas pouvoir compter sur la mémoire se trouvant sur la pile.

### 6.1 Introduction aux pointeurs

Avant de voir comment résoudre le souci qui nous préoccupe, on doit rentrer un peu plus en détails dans le mécanisme de la mémoire. Pour accéder à un élément en mémoire, tant qu'à présent on utilise des variables. En réalité, chaque case mémoire possède une adresse mémoire (un nombre entier) et une variable n'est rien d'autre qu'un nom raccourci pour une adresse.

En C, il est possible de manipuler ces adresses. Grâce à l'opérateur `&`, il est possible de récupérer l'adresse d'une variable. Cette adresse, qui est une valeur, on va pouvoir la stocker elle-même dans une variable. Les variables qui permettent de stocker des adresses sont appelées *pointeur*. Il s'agit d'un nouveau type de données que l'on indique en ajoutant une `*`. Prenons un exemple :

```
int a = 12;
int* p = &a;
printf ("L'adresse de la variable a est : %p\n", p);
```

Cet exemple déclare donc une variable `a` de type `int`. Cette variable contient la valeur 12. Ensuite, on déclare une nouvelle variable `p` de type `int*` (pointeur vers une variable de type `int`). La valeur que l'on met dans cette variable, c'est l'adresse de la variable `a`. On affiche ensuite cette adresse avec `printf` et le marqueur `%p` qui indique le type pointeur. Voici une illustration de ce qui pourrait se trouver en mémoire :

p : 2000	2008
a : 2008	12

Sur ce dessin, outre les noms des variables, on a également indiqué leurs adresses (2000 pour `p` et 2008 pour `a`). La raison pour laquelle la variable `a` se trouve à l'adresse 2008 est parce que le type `int*` occupe 8 octets en mémoire. On voit clairement sur le dessin que le contenu de la variable `p` est 2008, à savoir l'adresse de la variable `a`.

À partir du moment où on dispose d'une variable qui est un pointeur, il est possible de retrouver la valeur stockée à l'adresse pointée. Pour cela, on utilise l'opérateur `*`. Voici par exemple comment retrouver la valeur stockée dans la zone mémoire pointée par `p`.

```
int b = *p;
printf ("La valeur pointée par p est : %d\n", b);
```

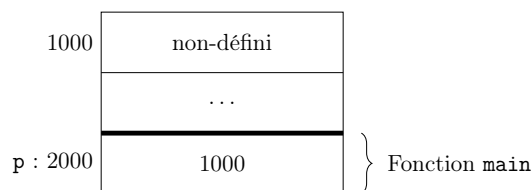
Notez que lorsqu'on déclare une variable de type pointeur, deux écritures sont possibles. On peut écrire « `int* p;` » ou « `int *p;` ». La seconde notation est assez souvent préférée puisqu'on peut la lire comme : « la valeur pointée par `p` (`*p`) est de type `int` ».

## 6.2 Tas ou mémoire dynamique

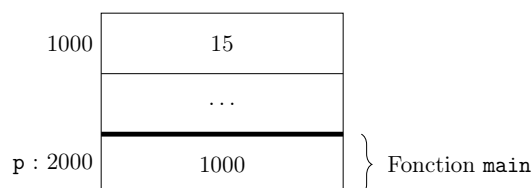
Outre la pile où se trouvent la mémoire réservée pour les variables déclarées dans les procédures et fonctions, il existe également *le tas*. Cette zone de la mémoire, une fois allouée, le reste malgré la fin des procédures et fonctions. On peut allouer de la mémoire dans cette zone en utilisant la fonction `malloc` disponible en chargeant la librairie `stdlib`. Lorsqu'une zone a été réservée dans le tas, vous recevez en retour un pointeur vers cette zone. Prenons un exemple :

```
1 int *p = malloc (sizeof (int));
2 *p = 15;
3
4 printf ("La valeur de variable int pointée par p vaut : %d\n", *p);
```

À la ligne 1, on demande à `malloc` de réserver de la place en mémoire pour stocker un `int` (grâce à `sizeof`). On stocke un pointeur vers cette zone mémoire dans la variable `p`.



Ensuite, la ligne 2 va modifier la variable qui est pointée par `p`. On y stocke la valeur 15.





La mémoire dynamique n'étant pas automatiquement libérée, comme c'est le cas pour les variables locales se trouvant sur la pile, il faut manuellement gérer la mémoire dynamique. Lorsque vous n'avez plus besoin d'une zone mémoire qui a été allouée avec `malloc`, il faut la libérer en utilisant la procédure `free`. Si on reprend notre exemple, lorsqu'on n'a plus besoin de la zone mémoire pointée par `p`, il suffira d'écrire :

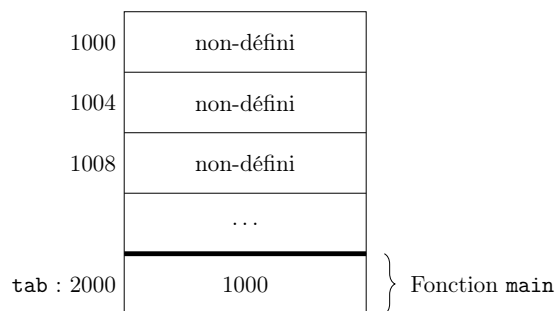
```
free (p);
```

### 6.3 Tableaux dynamiques

La fonction `malloc` permet de créer autant de mémoire que l'on souhaite. On va donc notamment pouvoir l'utiliser pour obtenir des tableaux dynamiques. Voyons un exemple :

```
int *tab = malloc (3 * sizeof (int));
```

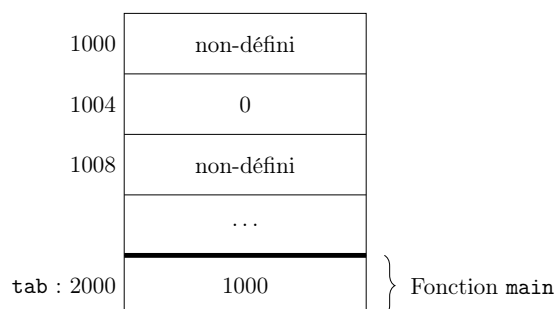
Avec cet exemple, on demande la réservation d'une zone mémoire qui permet de stocker trois variables `int`. Et on obtient en retour un pointeur qui point au début de cette zone.



Le pointeur que l'on reçoit ne permet d'accéder qu'à la première zone mémoire. On va pouvoir accéder à n'importe laquelle des zones en utilisant une notation que l'on connaît déjà, à savoir les crochets avec un indice. Ainsi, on pourrait par exemple écrire :

```
tab[1] = 0;
```

Ce qui mettrait la valeur 0 dans la deuxième zone, à savoir celle d'adresse 1004 :



Pour libérer la mémoire qui a été allouée, il suffit d'utiliser une seule fois **free** pour libérer toute la zone. De manière générale, pour être certain d'avoir bien libéré toute la mémoire, vous devriez avoir autant de **free** que de **malloc** dans vos programmes.

## 6.4 Fonction pour créer un tableau

Revenons-en maintenant à ce qu'on tentait de faire au début de ce chapitre, à savoir écrire une fonction qui crée et renvoie un nouveau tableau avec une taille reçue en paramètre, et dont les éléments sont tous initialisés à zéro.

```
int* newTab (int N)
{
    int *tab = malloc (N * sizeof (int));
    int i;
    for (i = 0; i < N; i++)
    {
        tab[i] = 0;
    }
    return tab;
}
```

On crée donc un tableau dynamique avec N cases que l'on remplit ensuite avec des 0 grâce à une boucle **for** et on termine en renvoyant un pointeur vers ce nouveau tableau dynamique.

## 7 Structure

Jusqu'à présent, toutes les variables qu'on a déclarées pouvaient être des simples variables pouvant contenir une seule valeur, que ce soit des variables comme `int` ou `float`, ou alors des pointeurs comme `int*`. Si on souhaite par contre stocker plusieurs valeurs, on peut alors utiliser des tableaux. Pour rappel, un tableau permet de stocker plusieurs valeurs, mais elles doivent toutes être du même type.

```

1 // Exemples de variables simples
2 int i = 12;
3 char c = 'X';
4 float f = 12.5;
5
6 // Exemples de pointeurs
7 int *p = &i;
8 char *str = "Hello";
9
10 // Exemples de tableaux
11 int tab1[] = {5, 8, -2};
12 int *tab2 = malloc (12 * sizeof (int));

```

On va maintenant s'intéresser à des types plus complexes qui permettent de stocker et de manipuler aisément plusieurs variables, de types pouvant être différents. Le principe général consiste à définir un nouveau type de données qui est composé d'une séquence d'autres types.

Prenons un exemple, on désire écrire un programme qui représente des personnes. Une personne, dans le cadre de notre programme, c'est un prénom, un nom et un âge. Il nous faut donc trois variables pour représenter une personne. Par exemple, supposons que l'on souhaite avoir un programme qui utilise les deux personnes suivantes :

- Pierre Dupont, 42 ans
- Bernadette Picard, 37 ans

Si on s'en tient à tout ce qu'on a vu jusqu'à présent, on pourrait par exemple écrire le programme présenté ci-dessous.

```

1 int main()
2 {
3     // Pierre
4     char *pFirstname = "Pierre";
5     char *pLastname = "Dupont";
6     int pAge = 42;
7
8     // Bernadette
9     char *mFirstname = "Bernadette";
10    char *mLastname = "Picard";
11    int mAge = 37;
12
13    // ...
14
15    return 0;
16 }

```

Ceci n'est pas acceptable dans le sens où on va multiplier inutilement des variables, qui en fait forment un tout. Ce qu'on veut, ce n'est pas devoir à chaque fois déclarer trois variables lorsqu'on veut définir et manipuler une personne, mais pouvoir le faire avec une seule variable.

## 7.1 Définir une nouvelle structure

Pour cela, on peut utiliser les *structures*. Il s'agit d'une construction particulière permettant de définir un nouveau type de données qui est composé d'autres types déjà existants. Pour notre exemple, ce qui nous intéresserait serait d'avoir un type permettant de représenter une personne. Pour cela, on va définir une nouvelle structure **person**.

```
struct person
{
    char *firstname;
    char *lastname;
    int age;
};
```

On déclare donc une structure en utilisant le mot réservé **struct** suivi du nom de la structure. Viennent ensuite les composants de la structure, qui sont simplement des déclarations de variables, le tout entre accolades. Ces variables sont appelées *champs* de la structure. Enfin, il ne faut pas oublier de terminer par un point-virgule. On déclare les structures en dehors de toute routine, par convention au début du fichier.

Dans notre exemple, une personne est donc composée d'un prénom et d'un nom qui sont des chaînes de caractères (**char\***) et d'un âge qui est un entier (**int**). Le fait d'avoir créé cette nouvelle structure introduit le nouveau type de données **struct person** qui peut donc être utilisé pour déclarer des variables. On peut par exemple écrire :

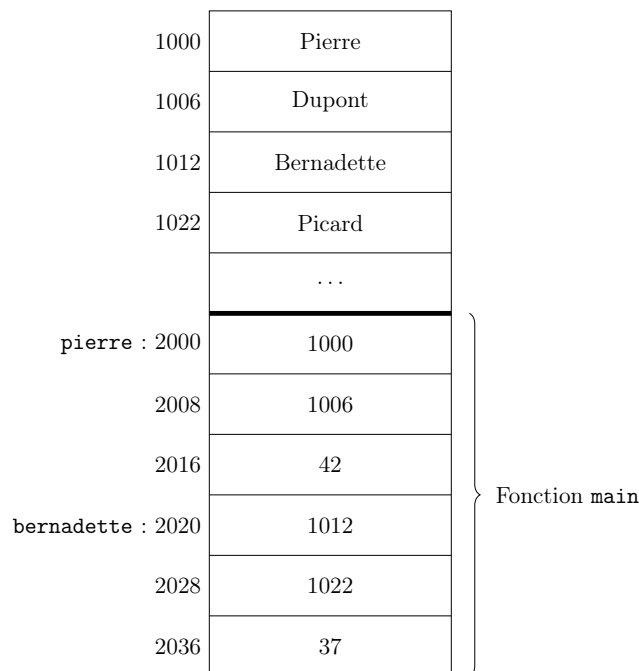
```
struct person pierre;
```

Cette instruction déclare une nouvelle variable dont le nom est **pierre** et dont le type est **struct person**. On peut accéder aux champs d'une structure, à partir d'une variable de type structure, en utilisant l'opérateur d'accès qui est le point (**.**). Par exemple, on peut réécrire la fonction **main** présentée plus haut comme montré ci-dessous.

```
1 int main()
2 {
3     struct person pierre;
4     pierre.firstname = "Pierre";
5     pierre.lastname = "Dupont";
6     pierre.age = 42;
7
8     struct person bernadette;
9     bernadette.firstname = "Bernadette";
10    bernadette.lastname = "Picard";
11    bernadette.age = 37;
12
13    // ...
14
15    return 0;
16 }
```

L'initialisation des champs de la structure reste lourde, mais on verra plus loin dans cette section comment rendre cela plus propre. Si on s'intéresse un moment à ce qui se passe en mémoire, on peut voir une structure comme une suite de blocs mémoires représentant les différents champs de la structure. Par exemple, la figure 2 montre à quoi pourrait ressembler la mémoire créée après exécution des lignes 3 à 11 de l'exemple juste ci-dessus. On y voit clairement les deux variables **pierre** et **bernadette** créée

dans la pile et les chaînes de caractères se retrouvant dans le tas. Pour rappel, un `char` occupe un octet en mémoire, un `int` en occupe quatre et un `char*` en occupe huit.



**FIGURE 2.** État de la mémoire (pile et tas), après création de deux structures de type `struct person`, qui sont créées dans la pile, dans une fonction `main`.

Avant de poursuivre dans la découverte des structures, voyons par exemple à quoi pourrait ressembler une procédure qui affiche proprement à l'écran une donnée de type structure. L'exemple suivant montre une procédure permettant d'afficher les champs d'une variable de type `struct person` sur la sortie standard. La procédure reçoit une variable de type `struct person` en paramètre et affiche ses différents champs.

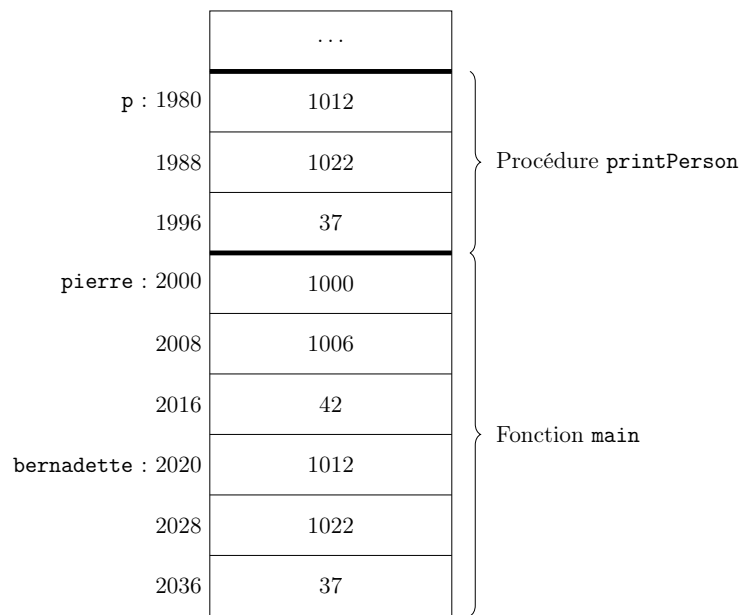
```

1 void printPerson (struct person p)
2 {
3     printf ("%s %s, %d ans", p.firstname, p.lastname, p.age);
4 }
```

Le souci lorsqu'on passe en paramètre une structure de la sorte, c'est le gaspillage de mémoire. En effet, tous les champs de la structure vont être copiés dans la pile, dans l'environnement de la procédure `printPerson` lors d'un appel. Ajoutons par exemple l'appel suivant à notre fonction `main` d'exemple :

```
printPerson (bernadette);
```

Une fois que l'exécution du programme sera rentrée dans la fonction `printPerson`, on se retrouve avec la situation illustrée en figure 3. Vous pouvez clairement y voir que toute la structure a été dupliquée. Ceci provoque évidemment un gaspillage de mémoire et on va voir dans la section suivante comme pallier ce problème grâce à l'utilisation de la mémoire dynamique.



**FIGURE 3.** État de la pile après appel de la procédure `printPerson` prenant en paramètre une structure de type `struct person`, depuis la fonction `main` (voir figure 2 pour les chaînes de caractères).

## 7.2 Structure en mémoire dynamique

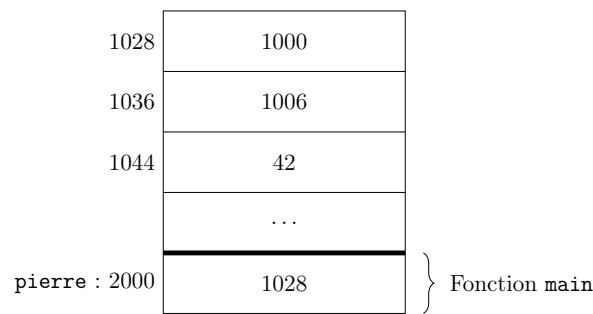
Tout comme pour les tableaux, on peut déclarer des variables de type structure en mémoire dynamique. Pour ce faire, il faut déclarer une variable de type pointeur et initialiser la zone mémoire en utilisant `malloc`. Une autre différence est présente lorsqu'on travaille en mémoire dynamique, il ne faut plus utiliser le point pour accéder aux champs de la structure, mais une flèche (`->`).

```

1 int main()
2 {
3     struct person *pierre = malloc (sizeof (struct person));
4     pierre->firstname = "Pierre";
5     pierre->lastname = "Dupont";
6     pierre->age = 42;
7
8     // ...
9
10    free (pierre);
11
12    return 0;
13 }
```

Évidemment, comme pour toute mémoire allouée avec `malloc`, une fois qu'elle n'est plus utile, il faut la libérer avec `free`.

Cette fois-ci, la situation est donc différente. Les données de la structure sont en effet créées dans le tas et seul un pointeur vers les champs se trouve stocké dans la variable `pierre` déclarée dans la fonction `main`. La figure 4 illustre la situation. On voit clairement qu'il s'agit maintenant d'un pointeur et si on modifie la procédure `printPerson` pour qu'elle prenne un pointeur en paramètre, on n'aura plus la copie intégrale de tous les champs de la structure, comme précédemment.



**FIGURE 4.** État de la mémoire (pile et tas), après création de deux structures de type `struct person`, qui sont créées dans la pile, dans une fonction `main` (voir figure 2 pour les chaînes de caractères).

Comme avec les tableaux, la raison pour laquelle il est intéressant de créer des structures en mémoire dynamique, c'est pour permettre l'écriture de fonction qui créent des nouvelles structures et renvoient des pointeurs vers ces dernières. L'exemple suivante est une fonction permettant de créer et d'initialiser une nouvelle variable de type `struct person`.

```

1 struct person* createPerson (char *first, char *last, int age)
2 {
3     struct person *p = malloc (sizeof (struct person));
4
5     p->firstname = first;
6     p->lastname = last;
7     p->age = age;
8
9     return p;
10 }
```

Supposons qu'on reprenne la dernière fonction `main` présentée et qu'on remplace la ligne 8 par le code suivant, qui permet de créer Bernadette. La figure 4 montre l'état de la mémoire après appel de la fonction `createPerson`, juste avant que l'instruction `return` de la fonction ne soit exécutée.

```
struct person *bernadette = createPerson ("Bernadette", "Picard", 37);
```

### 7.3 Structure représentant un tableau

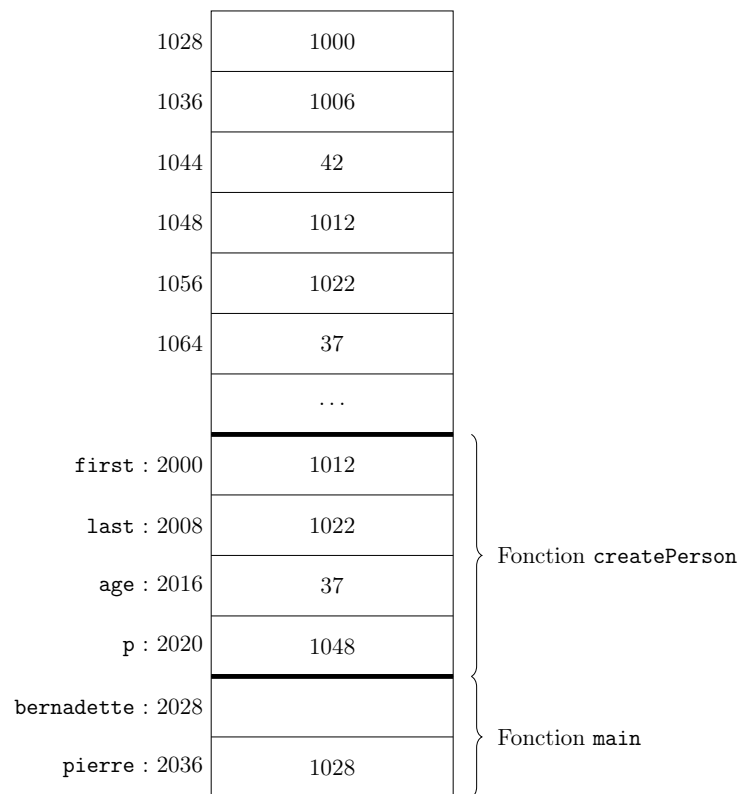
En C, il n'y a pas vraiment de type de données permettant de représenter proprement un tableau. Jusqu'à présent, on a toujours été obligé d'avoir deux variables : une pour stocker les données du tableau et une autre pour stocker sa taille. Grâce aux structures, on peut maintenant définir une nouvelle structure `intarray` représentant un tableau d'entiers `int`.

La structure va contenir deux champs. Le premier sera un `int` représentant sa taille et le second sera un pointeur représentant le tableau à proprement parler.

```

struct intarray
{
    int size;
    int *data;
};
```

Pour initialiser une variable de type `struct intarray`, il suffit d'écrire une fonction à qui on fournit



**FIGURE 5.** État de la mémoire (pile et tas), après appel de la fonction `createPerson`, juste avant exécution de l'instruction `return` de cette fonction (voir figure 2 pour les chaînes de caractères).

une taille en paramètre. Il faut aussi prévoir une fonction pour libérer la mémoire associée à une variable de ce type.

```

1 // Création d'un nouveau tableau d'entiers de taille n
2 struct intarray* createIntArray (int n)
3 {
4     struct intarray *tab = malloc (sizeof (struct intarray));
5
6     tab->size = n;
7     tab->data = malloc (n * sizeof (int));
8
9     return tab;
10 }
11
12 // Libération de la mémoire allouée pour un tableau d'entiers
13 void freeIntArray (struct intarray *tab)
14 {
15     free (tab->data);
16     free (tab);
17 }

```

## 7.4 Structure de structure

Une structure définit un nouveau type de données. On peut donc l'utiliser partout là où on a déjà utilisé des types de données : pour déclarer une variable, pour en faire un tableau, ou comme champ dans une structure. Commençons avec un premier exemple. Supposons qu'on souhaite développer une



application permettant de gérer une classe d'élèves. Pour cela, on souhaite créer une nouvelle structure représentant un étudiant. On va se baser sur la structure `person` et en plus ajoutant une information par rapport à l'identifiant unique de l'étudiant.

```
struct student
{
    struct person *p;
    char *id;
};
```

Une telle structure occupe 16 octets en mémoire, à savoir huit octets pour chacun des pointeurs. On aurait pu ne pas déclarer `p` comme un pointeur, mais dans ce cas, la structure `student` aurait pris plus de place en mémoire. On aurait eu un total de 32 octets : 24 pour la structure `person` (16 pour les deux pointeurs `char*`, 4 pour le `int` et 4 pour compléter pour avoir un multiple de huit<sup>1</sup>) et 8 pour le pointeur `char*`.

Supposons maintenant que l'on souhaite représenter une classe d'étudiants. Une classe porte un nom et représente une liste d'étudiants. Pour ce faire, on définit une nouvelle structure `class`.

```
struct class
{
    char *name;
    int n;
    struct person **students;
};
```

Vous aurez remarqué que `students` est un pointeur de pointeurs. En fait, cette variable est un tableau, dont les éléments sont chacun des pointeurs vers une `struct person`. Voici maintenant un exemple de code qui crée une nouvelle classe, avec deux étudiants :

```
int main()
{
    // Création de la classe
    struct class *c = malloc (sizeof (struct class));
    c->name = "2A";
    c->n = 2;
    c->students = malloc (c->n * sizeof (students*));

    c->students[0] = createPerson ("Pierre", "Dupont", 42);
    c->students[1] = createPerson ("Bernadette", "Picard", 37);
}
```

On voit très bien, grâce à cet exemple, que `c->students` est un tableau de pointeurs puisque la fonction `createPerson` renvoie bel et bien un pointeur `struct person*`.

---

1. On ne va pas entrer dans ces détails qui sont dépendants des différents compilateurs.

## 8 Organisation d'un programme en fichiers et compilation

Dans cette section, on va voir comment organiser un programme C en plusieurs fichiers. On verra également comment créer un exécutable à partir des différents fichiers, en utilisant le compilateur en ligne de commande. On verra également comment on peut utiliser l'outil **make** pour rendre la tâche de création d'un exécutable plus facile.

### 8.1 Prototype de fonction

On va commencer par s'intéresser à des programmes dont tout le code se trouve dans un seul fichier C. Si votre programme est décomposé en plusieurs fonctions, elles ne peuvent pas être déclarées dans n'importe quel ordre dans le fichier. En effet, toutes les fonctions utilisées par une fonction doivent être déclarées au-dessus de cette dernière. Voyons un exemple avec le programme suivant qui calcule la plus grande valeur parmi deux entiers encodés par l'utilisateur.

```

1 #include <stdio.h>
2
3 double max (double a, double b)
4 {
5     double max = a;
6     if (b > a)
7     {
8         max = b;
9     }
10    return max;
11 }
12
13 int main()
14 {
15     double a, b;
16
17     printf ("Entrez les deux nombres à comparer, séparés par une virgule, sans espace : ");
18     scanf ("%lf,%lf", &a, &b);
19     printf ("La plus grande valeur est : %lf\n", max (a, b));
20
21     return 0;
22 }
```

La fonction `main` utilise la fonction `max` et cette dernière doit donc être placée au-dessus de la fonction `main`, sans quoi le programme ne compilera pas. Si on avait fait le contraire, à savoir placer la fonction `max` en-dessous de la fonction `main`, on aurait eu l'erreur suivante lors de la compilation :

```

test.c: In function 'main':
test.c:9: warning: format '%lg' expects type 'double', but argument 2 has type 'int'
test.c:9: warning: format '%lg' expects type 'double', but argument 2 has type 'int'
test.c: At top level:
test.c:15: error: conflicting types for 'max'
test.c:9: error: previous implicit declaration of 'max' was here
```

Décortiquons ce message en commençant par les avertissements. Le compilateur indique que la ligne 9 (ligne 19 du programme ci-dessus) pose problème car on a utilisé `%lg` pour insérer un `double` alors que l'appel `max (a, b);` renvoie un `int`. En effet, lorsqu'une fonction appelée n'est pas trouvée avant dans le code, le compilateur suppose qu'il s'agit d'une fonction qui renvoie un `int` et il ne fait aucune hypothèse quant au nombre de paramètres de la fonction.

Venons-en maintenant à l'erreur de la ligne 9 (ligne 19 du programme ci-dessus) : `previous implicit declaration of 'max' was here`. Cette erreur indique qu'il y a un problème de type avec la fonction `max`. Le compilateur s'attendait en effet à ce que la fonction renvoie un `int`, mais il se rend compte à la ligne 15 (ligne 4 du programme ci-dessus) que la fonction `max` renvoie autre chose, à savoir un `double` : `conflicting types for 'max'`.

Pour régler le problème, il faut soit déclarer la fonction `max` avant la fonction `main` comme on a fait au début, ou alors il faut fournir le *prototype* de la fonction `max` en début de fichier. Un prototype, c'est une déclaration de fonction sans son corps. On déclare le type de retour et le nom de la fonction. Enfin, on peut aussi déclarer le type des paramètres, mais c'est optionnel. Pour la fonction `max`, on peut donc utiliser n'importe lequel des deux prototypes suivants :

```
double max();
double max (double, double);
```

Les prototypes sont placés par convention en tout début de fichier, juste après les `#include` et autres déclarations (`typedef`, `struct`...) et avant la première fonction du fichier. Le programme devient donc (remarquez le prototype en ligne 3) :

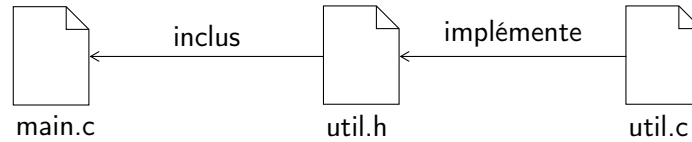
```
1 #include <stdio.h>
2
3 double max (double, double);
4
5 int main()
6 {
7     double a, b;
8
9     printf ("Entrez les deux nombres à comparer, séparés par une virgule, sans espace : ");
10    scanf ("%lf,%lf", &a, &b);
11    printf ("La plus grande valeur est : %lg\n", max (a, b));
12
13    return 0;
14 }
15
16 double max (double a, double b)
17 {
18     double max = a;
19     if (b > a)
20     {
21         max = b;
22     }
23     return max;
24 }
```

## 8.2 Fichier d'entête et librairie

Lorsqu'on définit plusieurs fonctions qui permettent de faire des opérations d'un même type (par exemple toutes des fonctions qui font des opérations mathématiques), on construit ce qu'on appelle une *librairie*. Une librairie (basique) est constituée de deux fichiers : un fichier d'entête (fichier `.h`) avec les prototypes et le fichier avec les définitions des fonctions (fichier `.c`).

La figure 6 montre les différentes relations qui existent dans un programme simple qui utilise une librairie. On a tout d'abord la librairie qui est définie par les deux fichiers `util.h` et `util.c`. On dit par ailleurs que le fichier `util.c` implémente la librairie `util.h`. On a ensuite le fichier `main.c` qui contient

le programme principal qui va utiliser la librairie en incluant le fichier `util.h`. Le fait d'inclure le fichier `util.h` fait en sorte que tous les prototypes se retrouvent inclus dans le fichier `main.c` et qu'on peut dès lors compiler ce fichier en utilisant ces prototypes.



**FIGURE 6.** Différents fichiers pour un programme avec une librairie basique.

Reprenons l'exemple de la section précédente et réorganisons le sous forme de librairie. La première chose à faire est de définir le fichier `util.h` qui va contenir les prototypes. Le fichier `util.h` va donc contenir une seule ligne.

```
double max (double, double);
```

**Listing 1.** Le fichier d'entête `util.h` contenant les prototypes de la librairie.

Une fois le fichier d'entête défini, il faut écrire le code de toutes les procédures et fonctions se trouvant dans le fichier d'entête. Il s'agit de l'implémentation des procédures et fonctions. Cette implémentation se fera dans le fichier `util.c`.

```
double max (double a, double b)
{
    double max = a;
    if (b > a)
    {
        max = b;
    }
    return max;
}
```

**Listing 2.** Le fichier `util.c` contenant le code de la librairie.

Enfin, pour terminer, on peut maintenant utiliser la librairie dans nos programmes. Il suffit pour cela d'inclure le fichier `util.h`. Le fichier `main.c` contient le programme principal qui inclut le fichier `util.h` afin de pouvoir utiliser cette librairie et donc les procédures et fonctions qui y sont définies.

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

int main()
{
    double a, b;

    printf ("Entrez les deux nombres à comparer, séparés par une virgule, sans espace : ");
    scanf ("%lf,%lf", &a, &b);
    printf ("La plus grande valeur est : %lg\n", max (a, b));

    return 0;
}
```

**Listing 3.** Le fichier `main.c` contenant le programme principal.

### 8.3 Chaîne de compilation

Maintenant qu'on a vu comment séparer le code en différents fichiers, voyons les différentes étapes nécessaire pour produire un fichier exécutable pour le programme principal. La plupart des éditeurs intégrés font tout le travail à votre place, mais nous allons ici voir la chaîne de compilation qui décrit toutes les opérations à effectuer. La figure 7 illustre la chaîne de compilation complète pour notre programme d'exemple. On peut voir distinctement deux phases : la compilation et le linking.

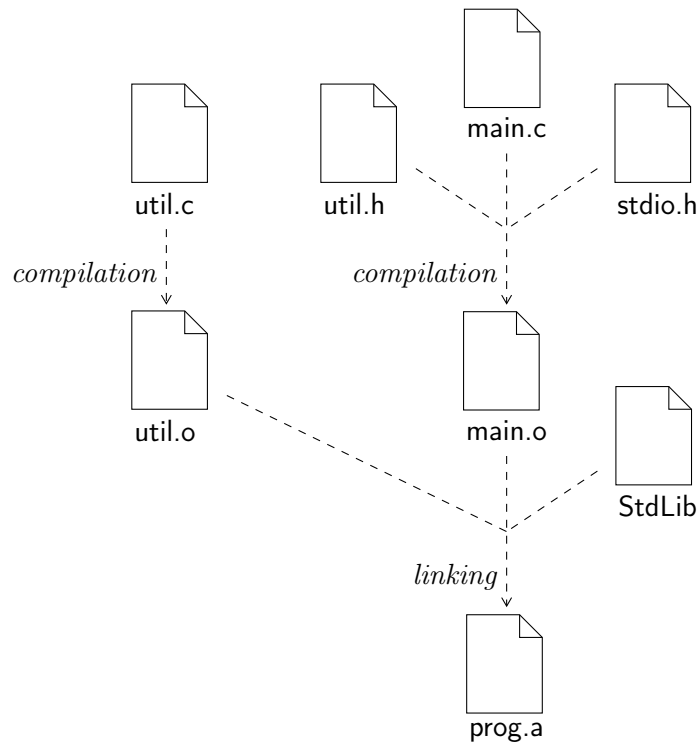


FIGURE 7. Illustration de la chaîne de compilation.

#### 8.3.1 Compilation

La première phase est la compilation. Elle permet de passer du fichier contenant le code source (fichier `.c`) à un fichier contenant le code machine (fichier `.o`). C'est durant cette étape de compilation que les fichiers d'entête seront nécessaires, afin d'inclure les différents prototypes. C'est également durant cette phase que le code source du programme est vérifié et que peuvent se produire des erreurs de compilation.

Chaque fichier `.c` de votre programme sera compilé séparément en un fichier `.o`. Le grand avantage est que si vous modifiez un fichier, il ne faudra recompiler que ce dernier, tous les autres fichiers `.o` restant valables.

### 8.3.2 Linking

La seconde phase est le linking. Elle permet de prendre tous les codes machines (fichier `.o`) nécessaire et d'en faire un seul exécutable (le programme final `prog.a`). C'est durant cette phase que tous les appels de procédures et de fonctions (qui ont pu être compilés grâce aux prototypes) vont effectivement être vérifiés. Si il manque des fichier `.o`, une erreur de linking sera produite.

Sur la figure 7, le fichier noté `StdLib` représente tout le code de la librairie standard fourni avec votre installation C et est nécessaire pour l'exemple étant donné que le fichier `main.c` inclus `stdio.h`.

## 8.4 Utiliser le compilateur gcc

Voyons brièvement comment compiler un programme écrit en C en utilisant le programme `gcc`. Il y a tout d'abord la phase de compilation qui consiste à créer les fichiers `.o` à partir des fichiers `.c`.

Pour compiler les deux fichiers `.c` en fichier `.o`, il suffit donc d'exécuter les deux instructions suivantes :

```
gcc -c util.c
gcc -c main.c
```

Après exécution de ces deux instructions, les fichiers `util.o` et `main.o` auront été créés. Notez qu'il ne faut pas préciser les noms des fichiers `.h` qui sont nécessaires pour la compilation.

Vient ensuite la phase de linking. Il suffit de lister tous les fichiers `.o` que vous souhaitez linker et enfin de choisir le nom du programme final (`prog.a` dans notre exemple).

```
gcc util.o main.o -o prog.a
```