

# Programmation C/C++

© 2017 tv <tvaira@free.fr> v.1.1

## Sommaire

<b>Premier programme</b>	<b>4</b>
Hello world en C++ . . . . .	4
Explications . . . . .	4
Hello world en C . . . . .	6
Structure d'un programme source . . . . .	6
Compilation . . . . .	7
Édition des liens . . . . .	8
Environnement de programmation . . . . .	9
Questions de révision . . . . .	10
Conclusion . . . . .	10
 <b>Un programme informatique</b>	 <b>11</b>
Objectifs du programmeur . . . . .	11
Entrées et sorties . . . . .	11
Objets, types et valeurs . . . . .	12
Nommer une variable . . . . .	13
Instructions . . . . .	14
Expressions . . . . .	14
Conditionner une action . . . . .	15
Les variables booléennes . . . . .	17
Itérer une action . . . . .	19
Questions de révision . . . . .	21
Conclusion . . . . .	21
 <b>Les types dérivés</b>	 <b>22</b>
Les chaînes de caractères . . . . .	22
Déclarations de chaînes de caractères . . . . .	23
Opérations sur les chaînes de caractères . . . . .	24
Lecture de chaînes de caractères . . . . .	24
Affichage de chaînes de caractères . . . . .	25
Les tableaux . . . . .	26
Déclarations de tableaux . . . . .	26

Les tableaux à plusieurs dimensions . . . . .	27
Parcourir un tableau . . . . .	28
Les tableaux et les pointeurs . . . . .	28
Trier un tableau . . . . .	29
Les tableaux dynamiques . . . . .	31
Questions de révision . . . . .	32
Conclusion . . . . .	32
<b>Les fonctions</b>	<b>33</b>
Programmation procédurale . . . . .	33
Fonction vs Procédure . . . . .	35
Déclaration de fonction . . . . .	35
Définition de fonction . . . . .	37
Appel de fonction . . . . .	37
Programmation modulaire . . . . .	38
Passage de paramètre(s) . . . . .	39
Passage par valeur . . . . .	39
Passage par adresse . . . . .	40
Passage par référence . . . . .	42
Valeur de retour . . . . .	43
Nommer une fonction . . . . .	44
La fonction <code>main</code> . . . . .	44
Intérêt des fonctions par l'exemple . . . . .	47
Surcharge de fonctions . . . . .	50
Paramètre par défaut . . . . .	51
Fonctions <code>inline</code> . . . . .	51
Fonctions statiques . . . . .	52
Nombre variable de paramètres . . . . .	52
Pointeur vers une fonction . . . . .	54
Questions de révision . . . . .	54
Conclusion . . . . .	54
<b>Types composés</b>	<b>55</b>
Structures . . . . .	55
Déclaration de structure . . . . .	56
Initialisation d'une structure . . . . .	57
Accès aux membres . . . . .	57
Affectation de structure . . . . .	57

Tableaux de structures . . . . .	58
Liste de structures . . . . .	58
Union . . . . .	59
Champs de bits . . . . .	60
Questions de révision . . . . .	61
Conclusion . . . . .	62
 <b>Annexe 1 : environnement de développement</b>	 <b>62</b>
 <b>Annexe 2 : éditer un fichier texte avec vim</b>	 <b>63</b>
 <b>Annexe 3 : Une liste succincte de commandes de base</b>	 <b>64</b>
 <b>Annexe 4 : printf() et scanf()</b>	 <b>65</b>
 <b>Annexe 5 : erreurs du débutant</b>	 <b>68</b>
 <b>Annexe 6 : opérations sur les nombres réels</b>	 <b>69</b>

# Premier programme

## Hello world en C++

Voici une version du premier programme que l'on étudie habituellement. Il affiche "Hello world !" à l'écran :

```
// Ce programme affiche le message "Hello world !" à l'écran

#include <iostream>

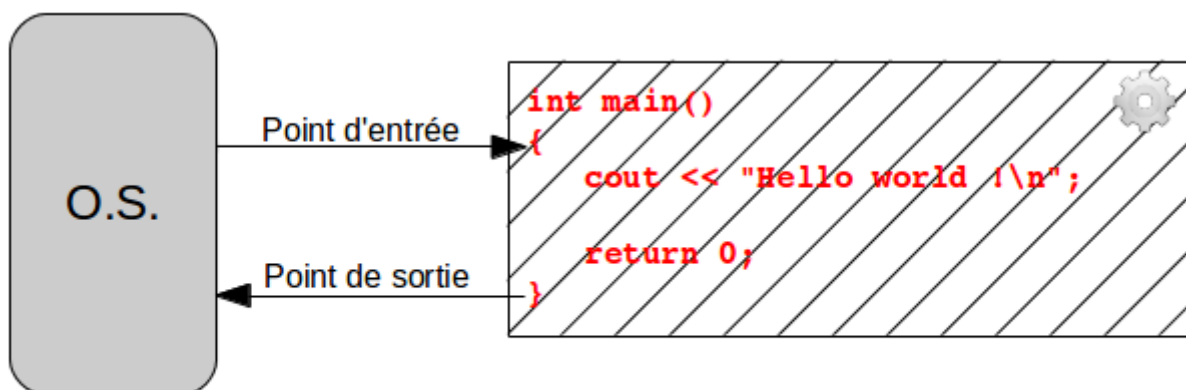
using namespace std;

int main()
{
    cout << "Hello world !\n"; // Affiche "Hello world !"

    return 0;
}
```

*Hello world (version 1) en C++*

## Explications



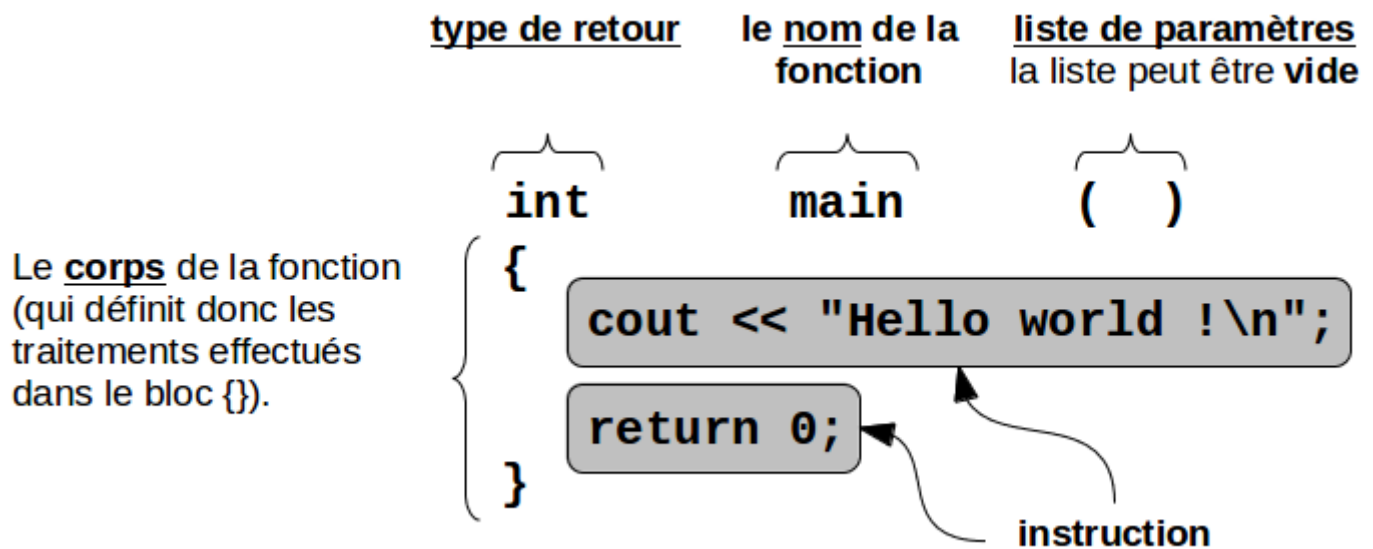
### Exécution d'un programme « binaire » par le système d'exploitation

Tout programme C/C++ doit posséder une (et une seule) **fonction** nommée **main** (dite fonction principale) pour indiquer où commencer l'exécution. Une fonction est essentiellement une **suite d'instructions** que l'ordinateur exécutera dans l'ordre où elles sont écrites.

Une fonction comprend quatre parties :

- un **type de retour** : ici **int** (pour *integer* ou entier) qui spécifie le type de résultat que la fonction retournera lors de son exécution. En C/C++, le mot **int** est un mot réservé (un mot-clé) : il ne peut donc pas être utilisé pour nommer autre chose.
- un **nom** : ici **main**, c'est le nom donné à la fonction (attention **main** est un mot-clé).
- une **liste de paramètres (ou d'arguments)** entre parenthèses (que l'on verra plus tard) : ici la liste de paramètres est vide
- un **corps de fonction** entre accolades (**{...}**) qui énumère les instructions que la fonction doit exécuter

✎ La plupart des instructions C/C++ se terminent par un point-virgule (;).



En C/C++, les **chaînes de caractères** sont délimitées par des guillemets anglais (`"`). `"Hello world !\n"` est donc une chaîne de caractères. Le code `\n` est un "caractère spécial" indiquant le passage à une nouvelle ligne (*newline*).

Le nom `cout` (*character output stream*) désigne le **flux de sortie standard** (l'écran par défaut). Les caractères "placés dans `cout`" au moyen de l'**opérateur** de sortie « `<<` » apparaîtront à l'écran.

`// Affiche "Hello world!"` placé en fin de ligne est un **commentaire**. Tout ce qui est écrit après `//` sera ignoré par le compilateur (la machine). Ce commentaire rend le code plus lisible pour les programmeurs. On écrit des commentaires pour décrire ce que le programme est supposé faire et, d'une manière générale, pour fournir des informations utiles impossibles à exprimer directement dans le code. Les langages C/C++ admettent aussi les commentaires multi-lignes avec `/* ... */`.

La première ligne du programme est un commentaire classique :  
il indique simplement ce que le programme est censé faire (et pas ce que nous avons voulu qu'il fasse!). Prenez donc l'habitude de mettre ce type de commentaire au début d'un programme.

La fonction `main` de ce programme retourne la valeur 0 (`return 0;`) à celui qui l'a appelée. Comme `main()` est appelée par le "système", il recevra cette valeur. Sur certains systèmes (Unix/Linux), elle peut servir à vérifier si le programme s'est exécuté correctement. Un zéro (0) indique alors que le programme s'est terminé avec succès (c'est une convention UNIX). Évidemment, une valeur différente de 0 indiquera que le programme a rencontré une erreur et sa valeur précisera alors le type de l'erreur.

En C/C++, une ligne qui commence par un `#` fait référence à une **directive** du préprocesseur (ou de pré-compilation). Le préprocesseur ne traite pas des instructions C/C++ (donc pas de `;`). Ici, la directive `#include <iostream>` demande à l'ordinateur de rendre accessible (d'"inclure") les fonctionnalités contenues dans un fichier nommé `iostream`. Ce fichier est fourni avec le compilateur et nous permet d'utiliser `cout` et l'opérateur de sortie « `<<` » dans notre programme.

Un fichier inclus au moyen de `#include` porte généralement l'extension `.h` ou `.hpp`. On l'appelle en-tête (*header*) ou **fichier d'en-tête**.

🔗 *En C++, il est maintenant inutile d'ajouter l'extension `.h` pour les fichiers d'en-tête standard.*

La ligne `using namespace std;` indique que l'on va utiliser l'**espace de nom** `std` par défaut.

🔗 *`cout` (et `cin`) existe dans cet espace de nom mais pourrait exister dans d'autres espaces de noms. Le nom complet pour y accéder est normalement `std::cout`. L'opérateur `::` permet la résolution de portée en C++ (un peu comme le `/` dans un chemin!).*

Pour éviter de donner systématiquement le nom complet, on peut écrire le code ci-dessous. Comme on utilise quasiment tout le temps des fonctions de la bibliothèque standard, on utilise presque tout le temps `"using namespace std;"` pour se simplifier la vie!

## Hello world en C

Voici la version du programme précédent pour le langage C :

```
// Ce programme affiche le message "Hello world !" à l'écran

#include <stdio.h> /* pour printf */

int main()
{
    printf("Hello world !\n"); // Affiche "Hello world !"

    return 0;
}
```

*Hello world (version 1) en C*

🔗 *`cout` (et `cin`) n'étant pas disponible en C, on utilisera `printf` (et `scanf`) pour afficher sur le flux de sortie standard (et lire sur le flux d'entrée). Les fonctions `printf` (et `scanf`) sont bien évidemment utilisables en C++. Il est même parfois conseillé de les utiliser car `cin` et `cout` peuvent être jusqu'à 10 fois plus lents.*

## Structure d'un programme source

Pour l'instant, la structure d'un programme source dans un fichier unique est donc la suivante :

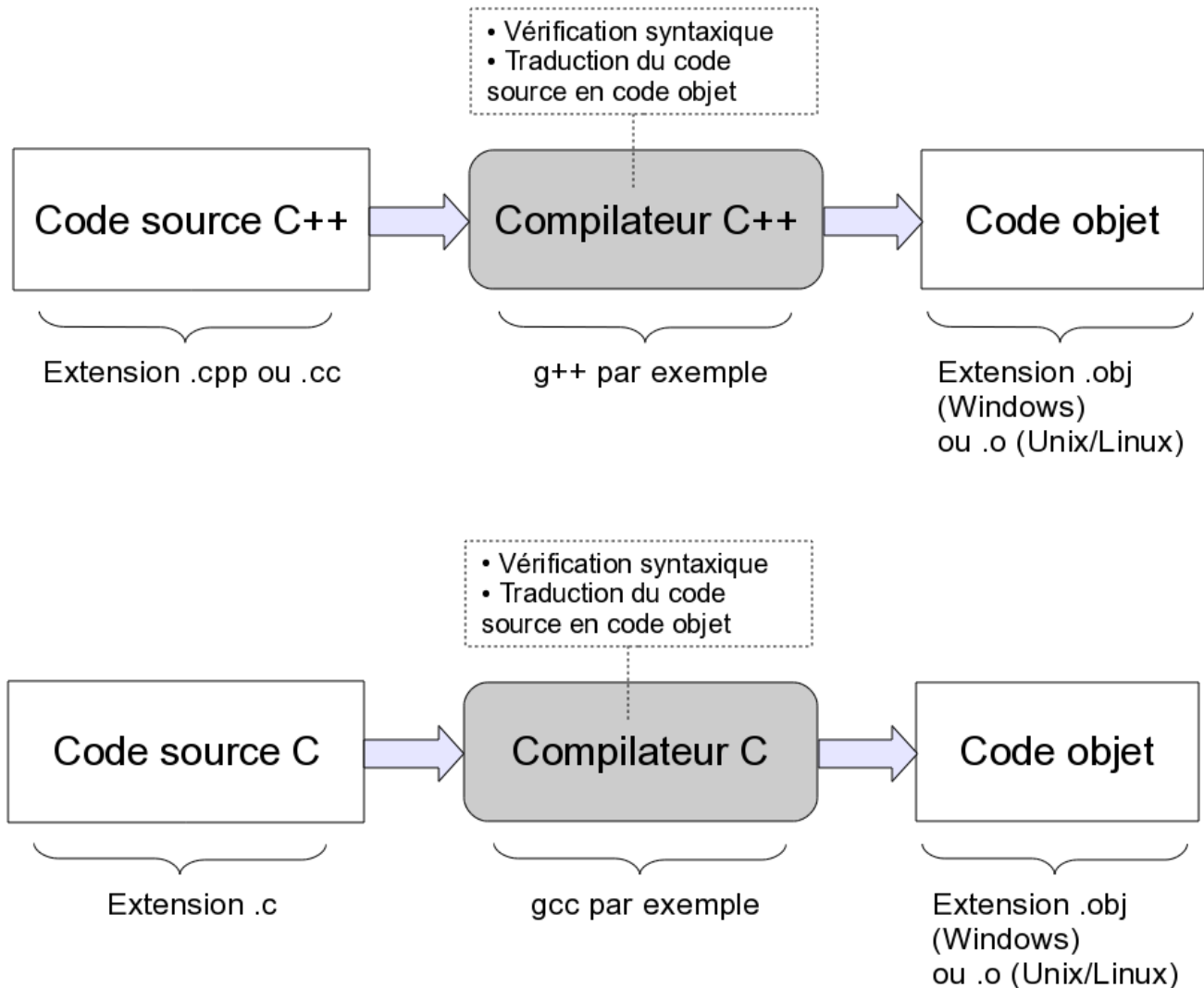
1. inclure les fichiers d'en-tête contenant les déclarations de fonctions externes à utiliser (`#include`)
2. définir la fonction principale (`main`)

🔗 *Par la suite, on y ajoutera des déclarations de constantes et de structures de données, des fonctions et on le décomposera même en plusieurs fichiers! Chaque chose en son temps ...*

## Compilation

C++ (ou C) est un langage compilé. Cela signifie que, pour pouvoir exécuter un programme, vous devez d'abord traduire sa forme lisible par un être humain (code source) en quelque chose qu'une machine peut "comprendre" (code machine). Cette traduction est effectuée par un programme appelé **compilateur**.

Ce que le programmeur écrit est le **code source** (ou programme source) et ce que l'ordinateur exécute s'appelle **exécutable**, **code objet** ou **code machine**.



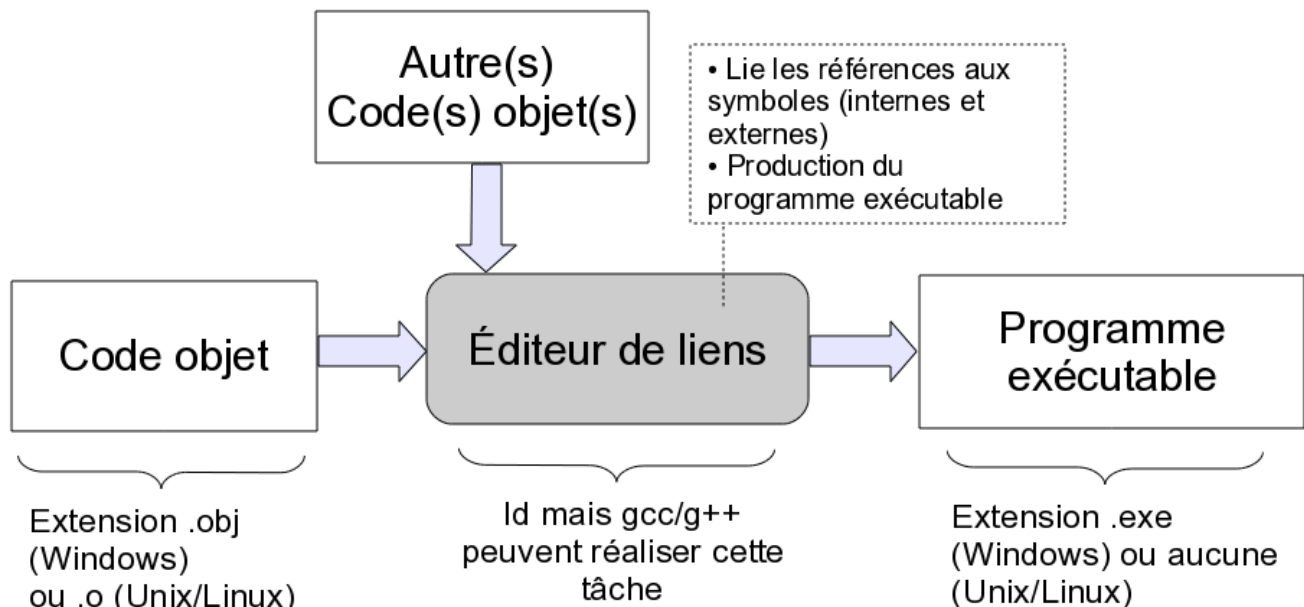
Vous allez constater que le compilateur est plutôt pointilleux sur la syntaxe ! Comme tous les programmeurs, vous passerez beaucoup de temps à chercher des erreurs dans du code source. Et la plupart de temps, le code contient des erreurs ! Lorsque vous coderez, le compilateur risque parfois de vous agacer. Toutefois, il a généralement raison car vous avez certainement écrit quelque chose qui n'est pas défini précisément par la norme C++ et qu'il empêche de produire du code objet.

☞ *Le compilateur est dénué de bon sens et d'intelligence (il n'est pas humain) et il est donc très pointilleux. Prenez en compte les messages d'erreur et analysez les bien car souvenez-vous en bien le compilateur est "votre ami", et peut-être le meilleur que vous ayez lorsque vous programmez.*

## Édition des liens

Un programme contient généralement plusieurs parties distinctes, souvent développées par des personnes différentes. Par exemple, le programme “Hello world!” est constitué de la partie que nous avons écrite, plus d’autres qui proviennent de la **bibliothèque standard** de C++ (cout par exemple).

Ces parties distinctes doivent être liées ensemble pour former un programme exécutable. Le programme qui lie ces parties distinctes s’appelle un **éditeur de liens** (*linker*).



⚠ Notez que le code objet et les exécutables ne sont pas portables entre systèmes. Par exemple, si vous compilez pour une machine Windows, vous obtiendrez un code objet qui ne fonctionnera pas sur une machine Linux.

Une bibliothèque n’est rien d’autre que du code (qui ne contient pas de fonction `main` évidemment) auquel nous accédons au moyen de **déclarations** se trouvant dans un fichier d’en-tête. Une déclaration est une suite d’instruction qui indique comment une portion de code (qui se trouve dans une bibliothèque) peut être utilisée. Le débutant a tendance à confondre bibliothèques et fichiers d’en-tête.

⚠ Une **bibliothèque dynamique** est une bibliothèque qui contient du code qui sera intégré au moment de l’exécution du programme. Les avantages sont que le programme est de taille plus petite et qu’il sera à jour vis-à-vis de la mise à jour des bibliothèques. L’inconvénient est que l’exécution dépend de l’existence de la bibliothèque sur le système cible. Une bibliothèque dynamique, *Dynamic Link Library (.dll)* pour Windows et *shared object (.so)* sous UNIX/Linux, est un fichier de bibliothèque logicielle utilisé par un programme exécutable, mais n’en faisant pas partie.

Les erreurs détectées :

- par le compilateur sont des erreurs de compilation (souvent dues à des problèmes de déclaration)
- celles que trouvent l’éditeur de liens sont des erreurs de liaisons ou erreurs d’édition de liens (souvent dues à des problèmes de définition)
- Et celles qui se produiront à l’exécution seront des erreurs d’exécutions ou de “logique” (communément appelées *bugs*).

Généralement, les erreurs de compilation sont plus faciles à comprendre et à corriger que les erreurs de liaison, et les erreurs de liaison sont plus faciles à comprendre et à corriger que les erreurs d’exécution et les erreurs de logique.



## Environnement de programmation

Pour programmer, nous utilisons un langage de programmation. Nous utilisons aussi un compilateur pour traduire le code source en code objet et un éditeur de liens pour lier les différentes portions de code objet et en faire un programme exécutable. De plus, il nous faut un programme pour saisir le texte du code source (un **éditeur de texte** à ne pas confondre avec un traitement de texte) et le modifier si nécessaire. Ce sont là les premiers éléments essentiels de ce qui constitue la **boîte à outils** du programmeur que l'on appelle aussi **environnement de développement**.

Si vous travaillez dans une fenêtre en mode ligne de commande (appelée parfois “mode console”), comme c’est le cas de nombreux programmeurs professionnels, vous devez taper vous-mêmes les différentes commandes pour produire un exécutable et le lancer.

```
[tv@alias iteration-2]$ vim node.cc
[tv@alias iteration-2]$ make
g++ -c -o graphviz.o graphviz.cc
g++ -c -o main.o main.cc
g++ -c -o node.o node.cc
g++ -o main graphviz.o main.o node.o
[tv@alias iteration-2]$ ./main
```

FIGURE 1 – Exemple de développement sur la console

Si vous travaillez dans un Environnement de Développement Intégré ou **EDI**, comme c’est aussi le cas de nombreux programmeurs professionnels, un simple clic sur le bouton approprié suffira.

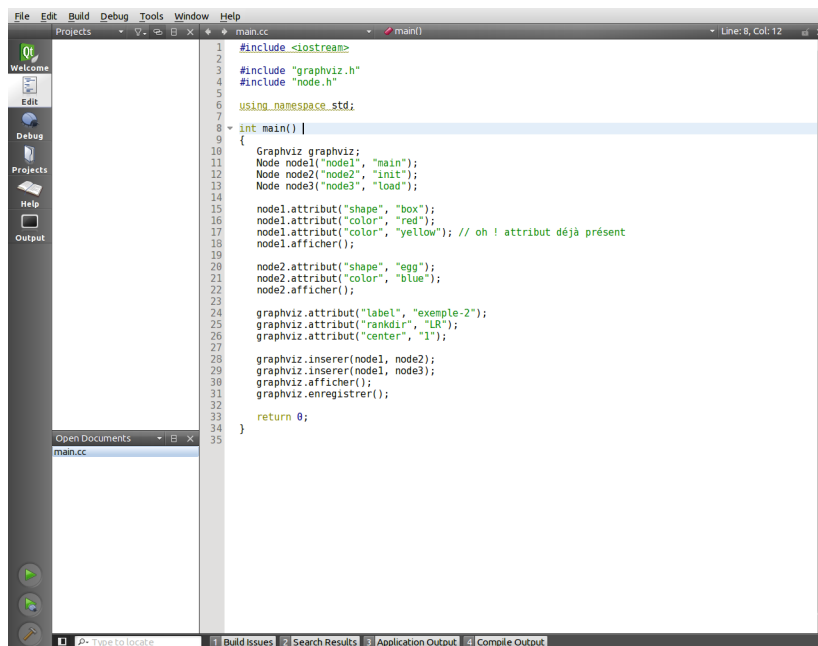


FIGURE 2 – Exemple de développement avec l’EDI Qt Creator

Un EDI (ou IDE pour *Integrated Development Environment*) peut contenir de nombreux outils comme : la documentation en ligne, la gestion de version et surtout un **débogueur** (*debugger*) qui permet de trouver des erreurs et de les éliminer.

Il existe de nombreux EDI (ou IDE) pour le langage C/C++ et on en utilisera certains notamment en projet. On peut citer : Visual C++, Builder, Qt Creator, Code::Blocks, devcpp, eclipse, etc ... Ils peuvent être très pratique mais ce ne sont que des outils et l’apprentissage du C/C++ ne nécessite pas forcément d’utiliser un EDI. Ils améliorent surtout la productivité dans un cadre professionnel.

☞ Voir l’Annexe n°1 (page 62) sur les environnements de développement.

## Questions de révision

L'idée de base des questions de révision est de vous donner une chance de voir si vous avez identifié et compris les points clés de cette partie.

**Question 1.** Citez une fonction qui doit apparaître dans tout programme C ou C++.

**Question 2.** À quoi sert la directive `#include` ?

**Question 3.** En C/C++, que signifie l'extension `.h` à la fin d'un nom de fichier ?

**Question 4.** Quel est le rôle du compilateur ?

**Question 5.** Que fait l'éditeur de liens pour votre programme ?

**Question 6.** Que permet de faire `cin` ?

**Question 7.** Que permet de faire `cout` ?

**Question 8.** Que contient un fichier source ?

**Question 9.** Que contient un fichier objet ?

**Question 10.** Qu'est-ce qu'un environnement de développement intégré (EDI) ?

## Conclusion

Qu'y a-t-il de si important à propos du programme "Hello world !" ? Son objectif était de vous familiariser avec les outils de base utilisés en programmation.

Retenez cette règle : il faut toujours prendre un exemple extrêmement simple (comme "Hello world") à chaque fois que l'on découvre un nouvel outil. Cela permet de diviser l'apprentissage en deux parties : on commence par apprendre le fonctionnement de base de nos outils avec un programme élémentaire puis on peut passer à des programmes plus compliqués sans être distraits par ces outils. Découvrir les outils et le langage simultanément est beaucoup plus difficile que de le faire un après l'autre.

*Conclusion : cette approche consistant à simplifier l'apprentissage d'une tâche complexe en la décomposant en une suite d'étapes plus petites (et donc plus faciles à gérer) ne s'applique pas uniquement à la programmation et aux ordinateurs. Elle est courante et utile dans la plupart des domaines de l'existence, notamment dans ceux qui impliquent une compétence pratique.*

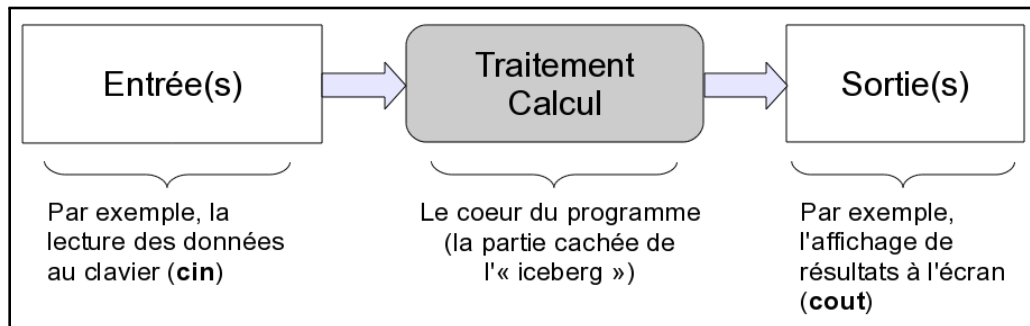
**Descartes** (mathématicien, physicien et philosophe français) dans le *Discours de la méthode* :

« diviser chacune des difficultés que j'examinerais, en autant de parcelles qu'il se pourrait, et qu'il serait requis pour les mieux résoudre. »

« conduire par ordre mes pensées, en commençant par les objets les plus simples et les plus aisés à connaître, pour monter peu à peu comme par degrés jusques à la connaissance des plus composés ... »

## Un programme informatique

Un programme informatique (simple) est souvent structuré de la manière suivante :



Un **traitement** est tout simplement l'action de produire des sorties à partir d'entrées : les vrais programmes ont donc tendance à produire des résultats en fonction de l'entrée qu'on leur fournit.

## Objectifs du programmeur

Le métier de programmeur consiste à écrire des programmes qui :

- donnent des résultats corrects
- sont simples
- sont efficaces

L'ordre donné ici est très important : peu importe qu'un programme soit rapide si ses résultats sont faux. De même, un programme correct et efficace peut être si compliqué et mal écrit qu'il faudra le jeter ou le récrire complètement pour en produire une nouvelle version. N'oubliez pas que les programmes utiles seront toujours modifiés pour répondre à de nouveaux besoins.

**Un programme (ou une fonction) doit s'acquitter de sa tâche de façon aussi simple que possible.**

Nous acceptons ces principes quand nous décidons de devenir des professionnels. En termes pratiques, cela signifie que nous ne pouvons pas nous contenter d'aligner du code jusqu'à ce qu'il ait l'air de fonctionner : nous devons nous soucier de sa structure. Paradoxalement, le fait de s'intéresser à la structure et à la "qualité du code" est souvent le moyen le plus facile de faire fonctionner un programme.

En programmation, les principes à respecter s'expriment par des **règles de codage** et des **bonnes pratiques**.

☞ Exemples :

- **Règle de codage** : les valeurs (comme 10 pour un `MAXIMUM` ou 3.14 pour `PI`) dans une expression doivent être déclarées comme des `CONSTANTES`. En `C`, les `CONSTANTES` s'écrivent en `MAJUSCULES`.
- **Bonne pratique** : Un programme (ou une fonction) ne doit pas dépasser 15 lignes de `C/C++` (accolades exclues).

## Entrées et sorties

Pour l'instant, nos programmes se limiteront à lire des entrées en provenance du **clavier** de l'utilisateur et de produire des résultats en sortie sur l'**écran** de celui-ci.

Pour lire des entrées saisies au clavier (l'entrée standard `stdin`), on utilisera :

- `scanf()` en C ou C++
- `cin` en C++

⚠ *Ne vous préoccuper pas des saisies invalides. Pour écrire des programmes qui fonctionnent et qui sont simples, il est plus prudent pour l'instant de supposer que l'on fait face à un utilisateur « parfait ». Par exemple, lorsque on souhaite lire un entier, l'utilisateur « parfait » saisit un entier !*

Pour afficher des résultats à l'écran (la sortie standard `stdout`), on utilisera :

- `printf()` en C ou C++
- `cout` en C++

⚠ *Evidemment, il existe d'autres types d'entrées/sorties (fichier, réseau, base de données, ...) pour les programmes. Nous les verrons plus tard.*

## Objets, types et valeurs

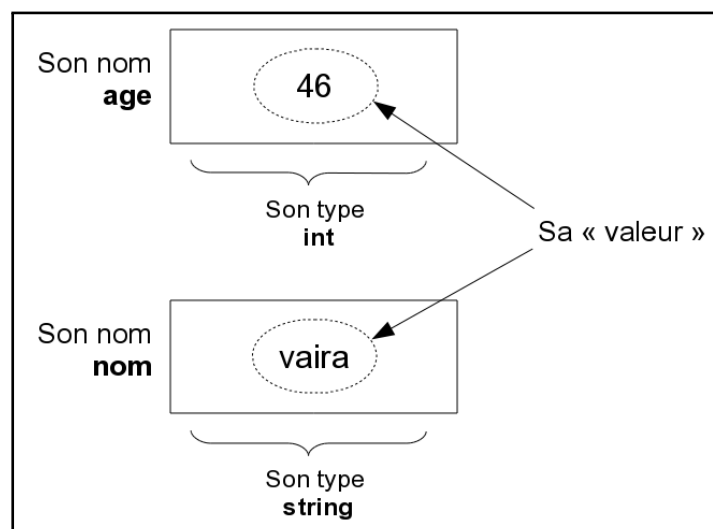
Pour pouvoir lire quelque chose, il faut dire où le placer ensuite. Autrement dit, il faut un “endroit” dans la mémoire de l'ordinateur où placer les données lues. On appelle cet “endroit” un **objet**.

Un objet est une région de la mémoire, dotée d'un **type** qui spécifie quelle sorte d'information on peut y placer. Un objet nommé s'appelle une **variable**.

🔍 *Exemples :*

- les **entiers** sont stockés dans des objets de type `int`
- les **réels** sont stockés dans des objets de type `float` ou `double`
- les **chaînes de caractères** sont stockées dans des objets de type `string` en C++
- etc ..

Vous pouvez vous représenter un objet comme “une boîte” (une case) dans laquelle vous pouvez mettre une **valeur** du **type** de l'objet :



Représentation d'objets

⚠ *Il est fondamentalement impossible de faire quoi que ce soit avec un ordinateur sans stocker des données en mémoire (on parle ici de la RAM).*

Une instruction qui définit une variable est ... une **définition** !

✎ Une déclaration est l'action de nommer quelque chose et une définition de la faire exister. Les déclarations (situées dans des fichiers `.h`) sont utilisées pendant la phase de compilation et les définitions (dans les fichiers `.cpp`) sont indispensables à l'édition des liens pour fabriquer un exécutable.

Une définition peut (et généralement doit) fournir une **valeur initiale**. Trop de programmes informatiques ont connu des *bugs* dûs à des oublis d'initialisation de variables. On vous obligera donc à le faire systématiquement. On appelle cela “respecter une règle de codage”. Il en existe beaucoup d'autres.

```
int nombreDeTours = 100; // Correct 100 est une valeur entière
string prenom = "Robert"; // Correct "Robert" est une chaîne de caractères

// Mais :
int nombreDeTours = "Robert"; // Erreur : "Robert" n'est pas une valeur entière
string prenom = 100; // Erreur : 100 n'est pas une chaîne de caractères (il manque les
    guillemets)
```

*Initialisation de variables*

✎ Le compilateur se souvient du type de chaque variable et s'assure que vous l'utilisez comme il est spécifié dans sa définition.

Le C++ dispose de nombreux types. Toutefois, vous pouvez écrire la plupart des programmes en n'en utilisant que cinq :

```
int nombreDeTours = 100; // int pour les entiers
double tempsDeVol = 3.5; // double pour les nombres en virgule flottante (double précision)
string prenom = "Robert"; // string pour les chaînes de caractères
char pointDecimal = '.'; // char pour les caractères individuels ou pour des variables
    entières sur 8 bits (un octet)
bool ouvert = true; // bool pour les variables logiques (booléennes)
```

*Les types usuels en C++*

✎ Le type `string` n'existe pas en langage C. Pour manipuler des chaînes de caractères en C, il faudra utiliser des tableaux que l'on verra plus tard.

```
#include <stdbool.h> /* pour le type bool en C */

int nombreDeTours = 100; // int pour les entiers
double tempsDeVol = 3.5; // double pour les nombres en virgule flottante (double précision)
char prenom[] = "Robert"; // un tableau de caractères pour stocker les chaînes de caractères
char pointDecimal = '.'; // char pour les caractères individuels ou pour des variables
    entières sur 8 bits (un octet)
bool ouvert = true; // bool pour les variables logiques (booléennes)
```

*Les types usuels en C*

## Nommer une variable

Un nom de variable est un nom principal (surtout pas un verbe) suffisamment éloquent, éventuellement complété par :

- une caractéristique d'organisation ou d'usage
- un qualificatif ou d'autres noms

On utilisera la convention suivante : **un nom de variable commence par une lettre minuscule puis les différents mots sont repérés en mettant en majuscule la première lettre d'un nouveau mot.**

Exemples : `distance`, `distanceMax`, `consigneCourante`, `etatBoutonGaucheSouris`, `nbreDEssais`, ...

Certaines abréviations sont admises quand elles sont d'usage courant : `nbre` (ou `nb`), `max`, `min`, ...

Les lettres `i`, `j`, `k` utilisées seules sont usuellement admises pour les indices de boucles.

Un nom de variable doit être uniquement composé de lettres, de chiffres et de "souligné" (`_`). Les noms débutant par le caractère "souligné" (`_`) sont réservés au système, et à la bibliothèque C.

Les noms débutants par un double "souligné" (`__`) sont réservés aux constantes symboliques privées (`#define ...`) dans les fichiers d'en-tête (`.h`).

Il est déconseillé de différencier deux identificateurs uniquement par le type de lettre (minuscule/majuscule). Les identificateurs doivent se distinguer par au moins deux caractères, parmi les 12 premiers, car pour la plupart des compilateurs seuls les 12 premiers symboles d'un nom sont discriminants.

**Les mots clés du langage sont interdits comme noms.**

✎ *l'objectif de respecter des règles de codage est d'augmenter la lisibilité des programmes en se rapprochant le plus possible d'expressions en langage naturel.*

## Instructions

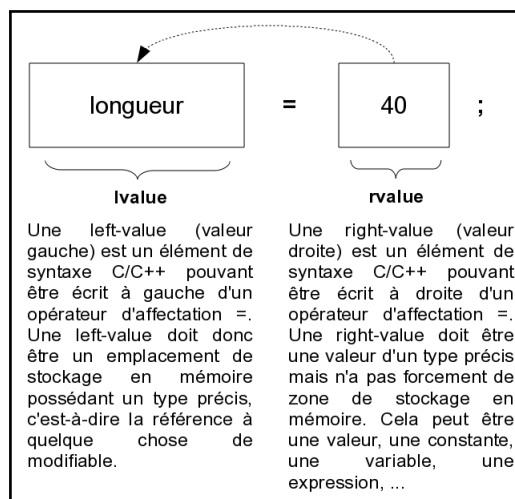
Les instructions principales sont :

- l'instruction déclarative
- l'instruction d'expression, l'assignation ou affectation
- l'instruction conditionnelle
- l'instruction itérative (la boucle)
- les branchements (sans condition) en effectuant un saut avec l'instruction `goto`

Les instructions (ou bloc d'instruction) sont exécutées séquentiellement : cela désigne le fait de faire exécuter par la machine une instruction, puis une autre, etc, en séquence. Cette construction se distingue du fait d'exécuter en parallèle des instructions (cf. programmation concurrente ou multi-tâches).

## Expressions

La brique de base la plus élémentaire d'un programme est une **expression**. Une expression calcule une **valeur** à partir d'un certain nombre d'opérandes. Cela peut être une **valeur littérale** comme `10`, `'a'`, `3.14`, `"rouge"` ou le **nom d'une variable**.



On utilise aussi des opérateurs dans une instruction :

```
// calcule une aire
int longueur = 40;
int largeur = 20;
int aire = longueur * largeur; // * est l'opérateur multiplication
```

Il existe de nombreux opérateurs : les opérateurs arithmétiques (+, -, \*, / et %), les opérateurs relationnels (<, <=, >, >=, == et !=), les opérateurs logiques && (et), || (ou) et ! (non), les opérateurs bits à bits & (et), | (ou) et ~ (non) ...

De manière générale, un programme informatique est constitué d'une **suite d'instructions**.

Une **instruction d'expression** est une expression suivie d'un point-virgule (;). Le point-virgule (;) est un élément syntaxique permettant au compilateur de "comprendre" ce que l'on veut faire dans le code (comme la ponctuation dans la langue française).

Dans les programmes comme dans la vie, il faut souvent choisir entre plusieurs possibilités. Le C/C++ propose plusieurs **instructions conditionnelles** : l'instruction **if** (choisir entre deux possibilités) ou l'instruction **switch** (choisir entre plusieurs possibilités).

Il est rare de faire quelque chose une seule fois. C'est pour cela que tous les langages de programmation fournissent des moyens pratiques de faire quelque chose plusieurs fois (on parle de traitement itératif). On appelle cela une **boucle** ou une **itération**.

Le C/C++ offrent plusieurs **instructions itératives** : la boucle **while** (et sa variante **do ... while**) et la boucle **for**.

## Conditionner une action

La célèbre attraction du train fou est interdite aux moins de 10 ans. On souhaite écrire un programme qui demande à l'utilisateur son âge et qui, si la personne a moins de 10 ans, affiche le texte « Accès interdit » ; ce qui peut se rédiger comme cela :

Variable **age** : Entier

```
age <- Lire un entier
Si age < 10
    Ecrire "Accès interdit"
```

Cela se traduit en C :

```
int age;

scanf("%d", &age);
if (age < 10)
{
    printf("Accès interdit\n");
}
```

On écrit donc le mot-clef **if**, la traduction en anglais de « **si** », puis on met entre parenthèses la condition à tester, à savoir **age < 10**. On n'oublie pas de mettre des accolades.

Ainsi, l'accès est interdit à un enfant de 8 ans :

```
$ ./tester-age
8
Accès interdit
```

À l’opposé, le programme n’affiche rien pour un âge de 13 ans :

```
$ ./tester-age
13
```

Pour exprimer la condition du « si » dans le programme, on a utilisé le symbole `<`, qui est l’opérateur de comparaison strictement inférieur. De manière symétrique, l’opérateur `>` permet de tester si un nombre est strictement supérieur à un autre. Lorsqu’on veut tester si un nombre est inférieur ou égal à un autre, on utilise le symbole `<=`. De manière symétrique, le symbole `>=` permet de tester si un nombre est supérieur ou égal à un autre.

Par exemple, le code suivant permet de tester si la température de l’eau a atteint 100 degrés :

```
int temperature;

scanf("%d", &temperature);
if (temperature >= 100)
{
    printf("L'eau bout !");
}
```

**Pour finir, le symbole `==` permet de tester l’égalité et la différence avec `!=` . Evidemment, il ne faut surtout pas confondre avec l’opérateur `=` qui permet d’effectuer une affectation.**

La notion de base est donc simple mais il est également facile d’utiliser `if` de façon trop simpliste.

Voici un exemple simple de programme de conversion *cm/inch* qui utilise une instruction `if` :

```
int main()
{
    const double conversion = 2.54; // nombre de cm pour un pouce (inch)
    int longueur = 1;                // longueur (en cm ou en in)
    char unite = 0; // 'c' pour cm ou 'i' pour inch
    cout << "Donnez une longueur suivi de l'unité (c ou i):\n";
    cin >> longueur >> unite;

    if (unite == 'i')
        cout << longueur << " in == " << conversion*longueur << " cm\n";
    else
        cout << longueur << " cm == " << longueur/conversion << " in\n";
}
```

*Conversion cm/inch (version 1)*

En fait cet exemple semble seulement fonctionner comme annoncé. Ce programme dit que si ce n’est pas une conversion en *inch* c’est forcément une conversion en cm. Il y a ici une dérive sur le comportement de ce programme si l’utilisateur tape ‘f’ car il convertira des cm en *inches* ce qui n’est probablement pas ce que l’utilisateur désirait. Un programme doit se comporter de manière sensée même si les utilisateurs ne le sont pas.

Voici une version améliorée en utilisant une instruction `if` imbriquée dans une instruction `if` :

```
if (unite == 'i')
    cout << longueur << " in == " << conversion*longueur << " cm\n";
else if (unite == 'c')
    cout << longueur << " cm == " << longueur/conversion << " in\n";
else
    cout << "Désolé, je ne connais pas cette unité " << unite << endl;
```

*Conversion cm/inch (version 2)*



De cette manière, vous serez tenté d'écrire des tests complexes en associant une instruction `if` à chaque condition. Mais, rappelez-vous, le but est d'écrire du code simple et non complexe.

En réalité, la comparaison d'unité à 'i' et à 'c' est un exemple de la forme de sélection la plus courante : une sélection basée sur la comparaison d'une valeur avec plusieurs constantes. Le C/C++ fournit pour cela l'instruction `switch`.

```
int main()
{
    const double conversion = 2.54; // nombre de cm pour un pouce (inch)
    int longueur = 1;               // longueur (en cm ou en in)
    char unite = 0; // 'c' pour cm ou 'i' pour inch
    cout << "Donnez une longueur suivi de l'unité (c ou i):\n";
    cin >> longueur >> unite;

    switch (unite)
    {
        case 'i':
            cout << longueur << " in == " << conversion*longueur << " cm\n";
            break;
        case 'c':
            cout << longueur << " cm == " << longueur/conversion << " in\n";
            break;
        default:
            cout << "Désolé, je ne connais pas cette unité " << unite << endl;
            break;
    }
}
```

*Conversion cm/inch (version 3)*

L'instruction `switch` utilisée ici sera toujours plus claire que des instructions `if` imbriquées, surtout si l'on doit comparer à de nombreuses constantes.

Vous devez garder en mémoire ces particularités quand vous utilisez un `switch` :

- la valeur utilisée pour le `switch()` doit être un entier, un `char` ou une énumération (on verra cela plus tard). Vous ne pourrez pas utiliser un `string` par exemple.
- les valeurs des étiquettes utilisées dans les `case` doivent être des expressions constantes (voir plus loin). Vous ne pouvez pas utiliser de variables.
- vous ne pouvez pas utiliser la même valeur dans deux `case`
- vous pouvez utiliser plusieurs `case` menant à la même instruction
- l'erreur la plus fréquente dans un `switch` est l'oubli d'un `break` pour terminer un `case`. Comme ce n'est pas une obligation, le compilateur ne détectera pas ce type d'erreur.

## Les variables booléennes

En C/C++ le programme suivant :

```
if (prix < 10)
{
    printf("Pas cher");
}
```

peut aussi s'écrire :

```
#include <stdbool.h> /* nécessaire en C */

bool estPasCher = (prix < 10);

if (estPasCher)
{
    printf("Pas cher");
}
```

La variable `estPasCher` est appelée une **variable booléenne** ou un **booléen** de type `bool` car elle ne peut être que vraie ou fausse, ce qui correspond en C/C++ aux valeurs `true` (pour vrai) et `false` (pour faux).

☞ N'oubliez pas que le type `bool` est natif en C++. Par contre en C, il faut inclure `stdbool.h` pour pouvoir l'utiliser.

Une maladresse classique avec les booléens est de faire quelque chose comme ceci :

```
int prix;
scanf("%d", &prix);
bool estCher = (prix > 100)

if (estCher == true)
{
    printf("C'est cher !");
}
```

Le code est correct mais on n'a pas besoin de tester si quelque chose est égal à `true` ou `false` , si ce quelque chose est lui même déjà `true` ou `false` !

On écrira donc :

```
if (estCher)
{
    printf("C'est cher !");
}

// ou :

if (!estCher)
{
    printf("Pas cher");
}
```

Il est bien sûr possible d'utiliser des opérateurs booléens (les opérateurs `&&` et `||`) pour combiner des conditions et les valeurs booléennes sont également utilisables.

Voici quelques extraits de code à titre d'exemple :

```
bool estSenior = (age >= 60);
bool estJeune = (age <= 25) && (age >= 12);
bool reductionPossible = (estSenior || estJeune);

if (reductionPossible)
{
    printf("Réduction!");
}
```

```
while (motDePasse != secret || agePersonne <= 3)
{
    printf("Accès refusé : mauvais mot de passe ou personne trop jeune\n");
    scanf("%d %d", &agePersonne, &motDePasse);
}

while (nbPersonnes <= nbMax && temperature <= 45)
{
    printf("Portes ouvertes\n");
    nbPersonnes = nbPersonnes + 1;
    scanf("%d", &temperature);
}
```

## Itérer une action

Le premier programme jamais exécuté sur un ordinateur à programme stocké en mémoire (l'EDSAC) est un exemple d'itération. Il a été écrit et exécuté par David Wheeler au laboratoire informatique de Cambridge le 6 mai 1949 pour calculer et afficher une simple liste de carrés comme ceci :

```
0 0
1 1
2 4
3 9
4 16
...
98 9604
99 9801
```

Ce premier programme n'a pas été écrit en C/C++ mais le code devait ressembler à ceci :

```
// Calcule et affiche une liste de carrés

#include <iostream>

using namespace std;

int main()
{
    int i = 0; // commencer à 0

    // tant que i est inférieur strict à 100 : on s'arrête quand i a atteint la valeur 100
    while (i < 100)
    {
        cout << i << '\t' << i * i << '\n'; // affiche i et son carré séparés par une
            tabulation
        ++i; // on incrémente le nombre et on recommence
    }

    return 0;
}
```

*Le premier programme jamais écrit (version while)*

Les accolades (`{}`) délimitent le **corps de la boucle** : c'est-à-dire le bloc d'instructions à répéter. La condition pour la répétition est exprimée directement dans le **while**. La boucle **while** s'exécutera **0 ou n fois**.

Il existe aussi une boucle `do ... while` à la différence près que cette boucle sera exécutée **au moins une fois**.

```
// faire
do
{
    cout << i << '\t' << i * i << '\n'; // affiche i et son carré séparés par une tabulation
    ++i; // on incrémente le nombre et on recommence
}
while (i < 100); // tant que i est inférieur strict à 100
```

*La boucle do .. while*

Donc écrire une boucle est simple. Mais cela peut s'avérer dangereux :

- Que se passerait-il si `i` n'était pas initialisé à 0 ? Voilà une première raison qui démontre que les variables non initialisées sont une source d'erreurs courante.
- Que se passerait-il si on oubliait l'instruction `++i` ? On obtient une boucle infinie (un programme qui ne "répond" plus). Il faut éviter au maximum d'écrire des boucles infinies. Il est conseillé d'éviter ce type de boucle : `while(1)` ou `while(true)` qui sont des boucles infinies.

Itérer sur une suite de nombres est si courant en C/C++ que l'on dispose d'une instruction spéciale pour le faire. C'est l'instruction **for** qui très semblable à **while** sauf que la gestion de la variable de contrôle de boucle est concentrée sur une seule ligne plus facile à lire et à comprendre. Il existe toujours une instruction **while** équivalente à une instruction **for**.

L'instruction **for** concentre : **une zone d'initialisation, une zone de condition et une zone d'opération d'incrément**. N'utilisez **while** que lorsque ce n'est pas le cas.

```
// Calcule et affiche une liste de carrés

#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    // exécute le bloc d'instructions de la boucle :
    // avec i commençant à 0 (initialisation)
    // tant que i est inférieur strict à 100 (condition)
    // en incrémentant i après chaque exécution du bloc d'instruction (opération d'
    //   incrément)
    for (int i = 0; i < 100; i++)
        cout << i << '\t' << pow(i, 2) << '\n'; // affiche i et son carré séparés par une
        //   tabulation

    return 0;
}
```

*Le premier programme jamais écrit (version for)*

☞ Ici, on utilise la fonction puissance (`pow`) de la bibliothèque mathématique. Pour cela, il faut inclure `math.h` en C ou `cmath` en C++ puis effectuer l'édition des liens avec l'option `-lm` (ce qui est fait par défaut maintenant).

## Questions de révision

L'idée de base des questions de révision est de vous donner une chance de voir si vous avez identifié et compris les points clés de cette partie.

**Question 11.** Qu'est-ce qu'une variable ?

**Question 12.** En C/C++, pourquoi doit-on lui donner un type ?

**Question 13.** Peut-on nommer une variable par un verbe ? Pourquoi ?

**Question 14.** Quelle est la différence entre une initialisation et une affectation ?

**Question 15.** Qu'est-ce qu'une *lvalue* ?

**Question 16.** Qu'est-ce qu'une variable booléenne ?

**Question 17.** Dans quel cas doit-on utiliser l'instruction `switch` ?

**Question 18.** Quelle est la différence entre une boucle `do ... while` et une boucle `while` ?

**Question 19.** Dans quel cas doit-on utiliser la boucle `for` ?

**Question 20.** Peut-on tester l'égalité entre deux nombres flottants ?

## Conclusion

Parce qu'il est clair que vous débutez à peine votre carrière de programmeur, n'oubliez pas qu'écrire de bons programmes, c'est écrire des programmes corrects, simples et efficaces.

**Rob Pike** (ancien chercheur des Laboratoires Bell et maintenant ingénieur chez Google) :

*« Règle n°4 : Les algorithmes élégants comportent plus d'erreurs que ceux qui sont plus simples, et ils sont plus difficiles à appliquer. Utilisez des algorithmes simples ainsi que des structures de données simples. »*

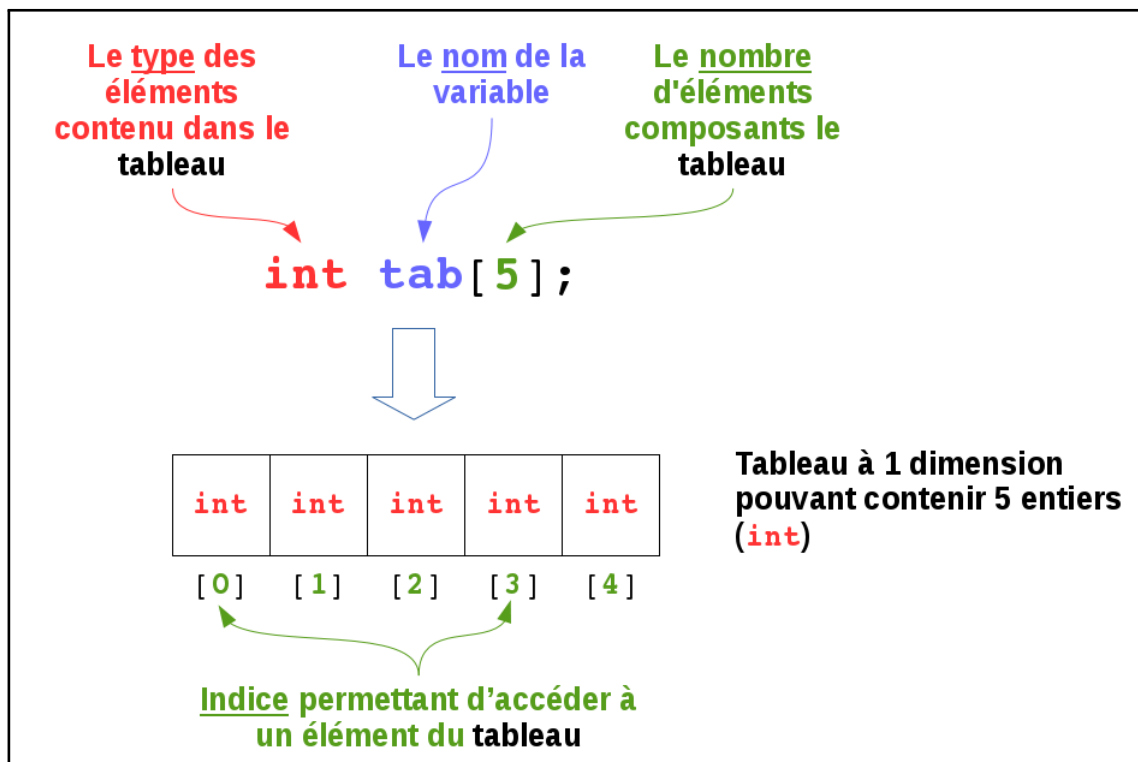
Cette règle n°4 est une des instances de la philosophie de conception KISS (*Keep it Simple, Stupid* dans le sens de « Ne complique pas les choses ») ou Principe KISS, dont la ligne directrice de conception préconise de rechercher la simplicité dans la conception et que toute complexité non nécessaire devrait être évitée.

## Les types dérivés

Dans de très nombreuses situations, les types de base s'avèrent insuffisants pour permettre de traiter un problème : il peut être nécessaire, par exemple, de stocker un certain nombre de valeurs en mémoire afin que des traitements similaires leurs soient appliqués.

Dans ce cas, il est impensable d'avoir recours à de simples variables car tout traitement itératif serait inapplicable. D'autre part, il s'avère intéressant de pouvoir regrouper ensemble plusieurs variables afin de les manipuler comme un tout.

Pour répondre à tous ces besoins, le langage C/C++ comporte la notion de **type agrégé** en utilisant : les **tableaux**. Il existe aussi les **structures**, les **unions** et les **énumérations**.



Notion de tableau

Les tableaux sont des **conteneurs** (*container*), c'est-à-dire est un **objet qui contient d'autres objets**.

☞ En C++, il existe de nombreux autres conteneurs (*vector*, *list*, *map*, *stack*, *queue*, ...). Un conteneur fournit généralement un moyen de gérer les objets contenus (au minimum ajout, suppression, parfois insertion, tri, recherche, ...) ainsi qu'un accès à ces objets.

## Les chaînes de caractères

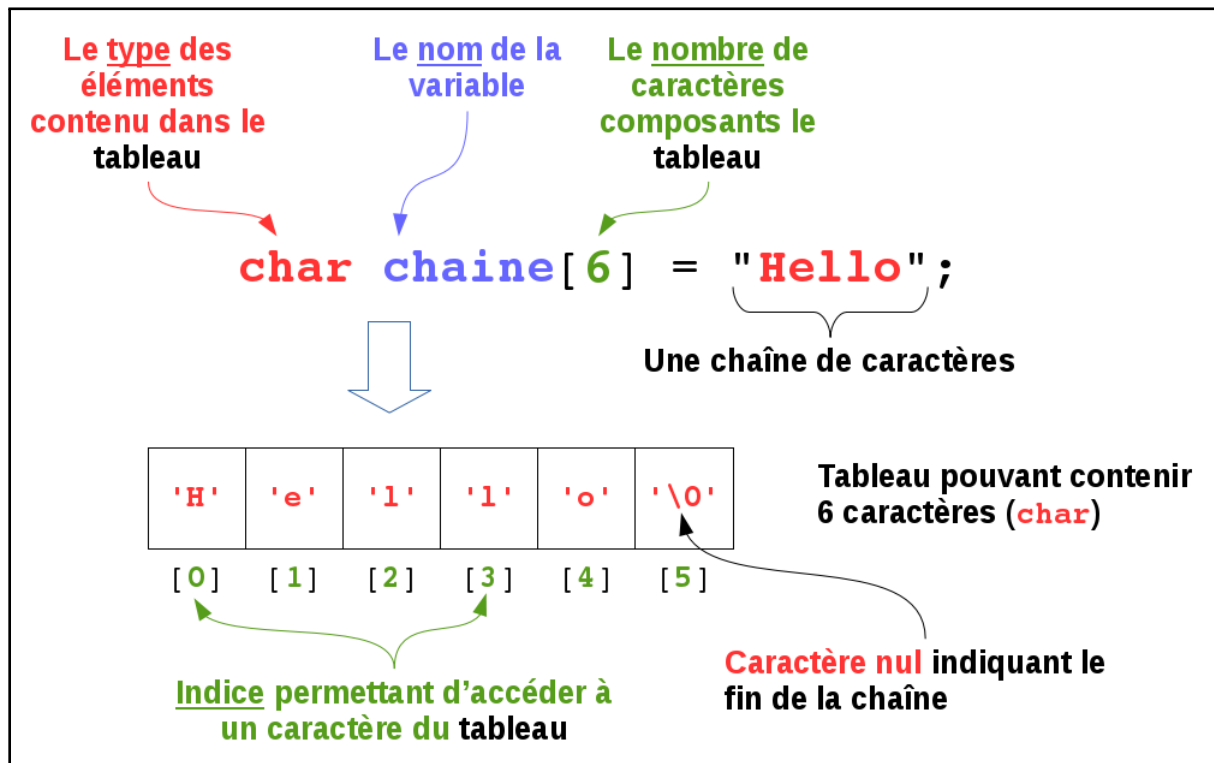
En C/C++, les **chaînes de caractères** sont délimitées par des guillemets anglais ("). "Hello world!\n" est donc une chaîne de caractères. Le code \n est un "caractère spécial" indiquant le passage à une nouvelle ligne (*newline*).

Les **chaînes de caractères** contiennent donc des caractères ! En C/C++, ils sont codés par défaut en ASCII (sur 8 bits soit un octet). Un **caractère** est délimité par des guillemets simples ('). En C/C++, un **caractères** est stocké dans une variable de type `char`.

☞ *man ascii*

Une **chaîne de caractères** est délimitée par un caractère spécial de **fin de chaîne** : le caractère nul qui a pour valeur 0 ou `'\0'`. Il est automatiquement ajouté lorsqu'on utilise la notation `("")`.

En C++, les **chaînes de caractères** sont stockées dans des variables de type `string`. En C, il n'y a pas de type dédié et on utilisera des tableaux de caractères.



## Déclarations de chaînes de caractères

En C++, les **chaînes de caractères** sont stockées dans des variables de type `string` :

```
string prenom = "Robert"; // string pour les chaînes de caractères
```

*Les chaînes de caractères en C++*

En C, il n'y a pas de type dédié et on utilisera des tableaux de caractères :

```
// une chaîne de caractères délimitée par un fin de chaîne (le fin de chaîne est ajouté automatiquement ici)
char nom[] = "Robert"; // la taille est calculée automatiquement (ici 6+1 = 7 caractères)

// un autre tableau de 6 caractères peut stocker une chaîne de 5 caractères (6-1 pour le fin de chaîne)
char autre[6];

// un tableau de caractères (ce n'est pas à proprement parler une chaîne de caractères car il n'y a pas de fin de chaîne)
char tab[4] = { 'a', 'b', 'c', 'd' };

// une chaîne de caractères (il y a le fin de chaîne)
char chaine[4] = { 'a', 'b', 'c', '\0' };

// Un simple caractère
```

```
char lettre = 'A'; // le caractère ASCII 'A' soit 0x41
```

*Les chaînes de caractères en C*

## Opérations sur les chaînes de caractères

En C++, le type `string` permet d'utiliser les opérateurs courants : `=` pour l'affectation, `+` pour la concaténation ou `==` pour la comparaison par exemple. En C, ce n'est pas possible et il faut passer par des fonctions : `strcpy()` pour copier une chaîne, `strcat()` pour la concaténation ou `strcmp()` pour la comparaison. Ces fonctions ont besoin impérativement que les chaînes de caractères se terminent par un fin de chaîne (caractère nul). Attention aussi aux dépassements de taille !

☞ *man string*

```
string nom = "Arizona";
string fils = nom + " Junior"; // Ajoute (concatène) des caractères à la fin

if (nom == fils) cout << "Les deux chaînes sont identiques.\n";
```

*Opérations sur les chaînes de caractères en C++*

☞ <http://www.cplusplus.com/reference/string/string/>

```
#include <string.h> /* pour les fonctions str... */


char nom[] = "Arizona"; // ici autorisé car c'est une initialisation et non une affectation
char fils[16];

strcpy(fils, nom); // copie nom dans fils
strcat(fils, " Junior"); // concatène deux chaînes

// comparaison de chaînes
if (strcmp(nom,fils) == 0) printf("Les deux chaînes sont identiques.\n");
```

*Opérations sur les chaînes de caractères en C*

## Lecture de chaînes de caractères

La lecture des chaînes de caractère (`scanf()` en C/C++ ou `cin` en C++) se termine sur ce qu'on appelle un espace blanc (*whitespace*), c'est-à-dire le caractère espace, une tabulation ou un caractère de retour à la ligne (généralement la touche ). Notez que les espaces sont ignorés par défaut.

```
// En C/C++ :
char msg[16];
scanf("%s", &msg[0]); // %s pour une chaîne mais attention au dépassement de taille (on peut
    utiliser fgets)
scanf("%15s", &msg[0]); // on peut limiter le nombre de caractères saisis (ici 15+1 = 16)

// si votre saisie comporte des espaces, vous pouvez utiliser :
// une expression rationnelle (entre crochets) acceptant tous les caractères sauf (^) le
// retour à la ligne (\n) :
scanf("%15[^\n]s", &msg[0]);

// ou la fonction fgets
fgets(&msg[0], 16, stdin); // attention fgets stocke le retour à ligne
```



```
// En C++ :  
string message; // chaîne de caractères seulement en C++  
cin >> message;  
  
// si votre saisie comporte des espaces, il vous faudra alors utilisé la fonction getline  
// cf. http://www.cplusplus.com/reference/string/string/getline/  
getline(std::cin, message);
```

*Lecture de chaînes de caractères en C/C++*

✚ Sous Linux, vous pouvez indiquer la fin d'un flux en combinant les touches **Ctrl** + **d** qui indiquera qu'il n'y a plus de saisie. Vous pouvez aussi stopper le programme avec **Ctrl** + **z** ou l'interrompre avec **Ctrl** + **c**.

Les caractères tapés ne sont pas directement transmis au programme, mais placés (par le système) dans un tampon (*buffer*). Il est possible qu'il reste des caractères d'une saisie précédente et donc vous aurez besoin de vider le *buffer* avant de (re)faire une saisie. La méthode pour vider le buffer clavier (*stdin*) consiste à consommer tous les caractères présents dans ce buffer jusqu'à ce qu'il soit vide :

```
// En C/C++ :  
int c = 0;  
while ((c = getchar()) != '\n' && c != EOF);  
  
// Ou :  
scanf("%*[^\\n]");  
getchar();  
  
// En C++ :  
#include <limits>  
  
cin.clear();  
cin.ignore(numeric_limits<streamsize>::max(), '\\n');
```

*Vider le tampon stdin*

✚ EOF signifie End Of File. Pour générer un EOF avec le clavier, il suffit de taper, en début de ligne, **Ctrl** + **z** sous DOS/Windows et **Ctrl** + **d** sous UNIX, puis de valider par Entrée.

## Affichage de chaînes de caractères

Pour l'affichage des chaînes de caractère, on utilise `printf()` en C/C++ ou `cout` en C++ :

```
char msg[] = "Bonjour"; // chaîne de caractères C/C++  
printf("msg : %s contient %d caractères\\n", msg, strlen(msg)); // %s pour une chaîne se  
    terminant par un fin de chaîne  
printf("premier caractère : %c\\n", msg[0]); // %c pour un simple caractère  
  
string message = "Bonjour"; // chaîne de caractères seulement en C++  
cout << "msg : " << msg << " contient " << msg.length() << " caractères" << endl; // endl  
    est équivalent à \\n  
cout << "premier caractère : " << msg[0] << "\\n";
```

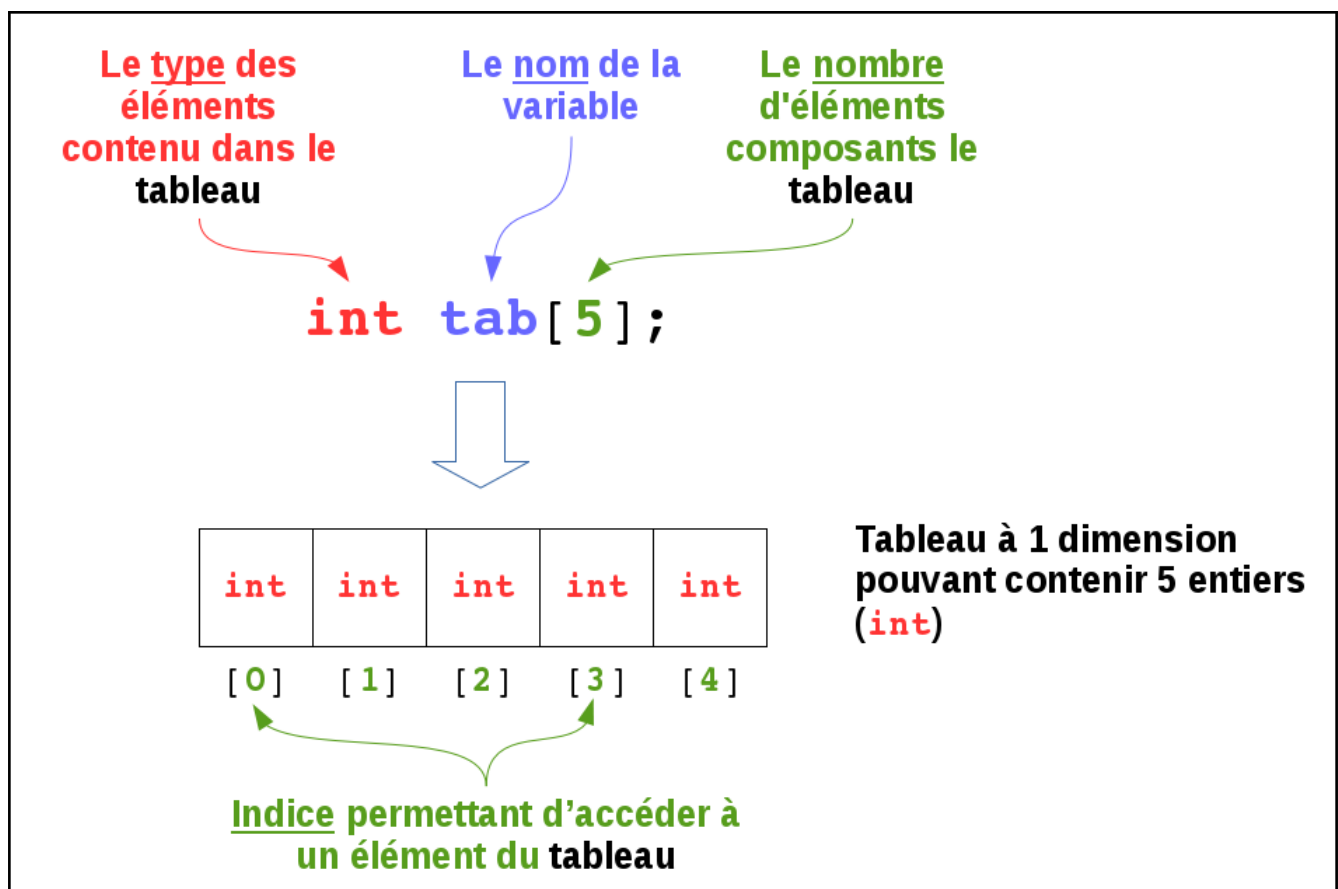
*Affichage de chaînes de caractères en C/C++*

☞ L'affichage est bufferisé lui aussi. Tant que le tampon n'est pas plein, les caractères transmis ne seront pas effectivement affichés mais tout simplement placés dans le tampon. Pour vider le contenu du tampon vers l'écran, il faut un retour à la ligne (`\n`) ou que le tampon soit plein. On peut aussi forcer le vidage de ce tampon à l'aide de la fonction `fflush(stdout)` ou carrément ne pas utiliser de buffer en appelant `setbuf(stdout, NULL)`.

## Les tableaux

Un **tableau** est un **ensemble d'éléments de même type** désignés par un identificateur unique (un nom). Chaque élément est repéré par une valeur entière appelée **indice** (ou index) indiquant sa position dans l'ensemble.

Les tableaux sont toujours à bornes statiques et leur indiciage démarre toujours à partir de 0.



## Déclarations de tableaux

La forme habituelle de déclaration d'un tableau est la suivante :

```
type identificateur[dimension1]... [dimensionn]
```

Exemple de déclarations de tableaux :

```
int notes[1000]; // un tableau de 1000 int non initialisé
float notes[1000]; // un tableau de 1000 float non initialisé
```

*Déclarations de tableaux en C/C++*

☞ Par “non initialisé”, on entend qu’il y a une valeur dans chaque case du tableau mais on ne la connaît pas. Il faut donc considérer qu’il y a “n’importe quoi” ! On rappelle que “rien” n’est pas une notion en informatique car les bits prennent soit la valeur 0 soit la valeur 1 : il n’y a pas de valeur “rien”.

Certains cas d’initialisation de tableaux sont admis :

```
int notes[1000] = {0}; // un tableau de 1000 int initialisé à 0

float f[1000] = {0.}; // un tableau de 1000 float initialisé à 0

int coefficients[4] = { '1', '2', '2', '4' }; // un tableau de 4 entiers

// La dimension d’un tableau peut être omise si le compilateur peut en définir la valeur
float t[] = {2., 7.5, 4.1}; // tableau de 3 éléments

// Et seulement avec le compilateur g++ :
// On peut utiliser une variable
int nbProduits = 1000;
int stock[nbProduits] = {0};
// et une saisie
int nbCases;
scanf("%d", &nbCases);
int cases[nbCases] = {0}; // certaines versions n’admettent pas l’initialisation
```

*Déclaration et initialisation de tableaux en C/C++*

Sinon il vous faudra le faire manuellement :

```
int nbEleves = 30;
int presences[nbEleves];
int i;

// Avec une boucle :
for(i=0; i<nbEleves; i++)
{
    presences[i] = 1; // ici 1 est la valeur initiale
}

// Avec la fonction memset :
memset(&presences[0], 0, 30*sizeof(int));
```

*Initialisation de tableaux en C/C++*

## Les tableaux à plusieurs dimensions

☞ Contrairement à beaucoup d’autres langages, il n’existe pas en C de véritable notion de tableaux multidimensionnels. De tels tableaux se définissent par composition de tableaux, c’est à dire que les éléments sont eux-mêmes des tableaux.

```
// tableau à 2 dimensions de 2 lignes et 5 colonnes :
int m[2][5] = { 2, 6, -4, 8, 11, // initialise avec des valeurs
               3, -1, 0, 9, 2 };

// tableau à 2 dimensions pour stocker plusieurs chaînes de caractères
char noms[][16] = { {"robert"},
                    {"roger"},
```

```
    {"raymond"},  
    {"alphonse"} };
```

```
int x[5][12][7]; // tableau a 3 dimensions, rarement au-delà de cette dimension
```

*Les tableaux à plusieurs dimensions en C*

## Parcourir un tableau

Pour parcourir un tableau, il faut utiliser une boucle :

```
int nbQuestions = 20;  
int points[nbQuestions];  
int i;  
  
// saisir des données dans un tableau :  
for(i=0;i<nbQuestions;i++)  
{  
    scanf("%d", &points[i]);  
}  
  
// afficher des données d'un tableau :  
for(i=0;i<nbQuestions;i++)  
{  
    printf("%d\n", points[i]);  
}
```

*Parcourir un tableau en C/C++*

⊕ Le plus grand danger dans la manipulation des tableaux est d'accéder en écriture en dehors du tableau. Cela provoque un accès mémoire interdit qui n'est pas contrôlé au moment de la compilation. Par contre, lors de l'exécution, cela provoquera une exception de violation mémoire (*segmentation fault*) qui se traduit généralement par une sortie prématurée du programme avec un message "Erreur de segmentation".

## Les tableaux et les pointeurs

⚠ L'identificateur du tableau (le nom de la variable) ne désigne pas le tableau dans son ensemble, mais plus précisément l'adresse en mémoire du début du tableau (l'adresse de la première case). Ceci implique qu'il est impossible d'affecter un tableau à un autre. L'identificateur d'un tableau sera donc "vu" comme un **pointeur constant**.

```
#define MAX 20 // définit l'étiquette MAX égale à 20
```

```
// Attention :  
int a[MAX], b[MAX];
```

```
a = b; // cette affectation est interdite ! Il faudra faire une boucle pour traiter chaque  
case
```

La dimension d'un tableau peut être omise dans 2 cas :

– le compilateur peut en définir la valeur

```
int t[] = {2, 7, 4}; // tableau de 3 éléments
```

- l’emplacement mémoire correspondant a été réservé

```
// la fonction fct admet en parametre
void fct(int t[]) // un tableau d'entiers qui existe déjà
```

✎ Lorsque le nom d’un tableau constitue l’argument d’une fonction, c’est l’adresse du premier élément qui est transmise. Ses éléments ne sont donc pas recopiés. Lorsque l’on passe un tableau en paramètre d’une fonction, il n’est pas possible de connaître sa taille et il faudra donc lui passer aussi sa taille.

Utilisation des pointeurs avec les tableaux :

```
int t[5] = {0, 2, 3, 6, 8}; // un tableau de 5 entiers
int *p1 = NULL; // le pointeur est initialisé à NULL (précaution obligatoire)
int *p2; // pointeur non initialisé : il pointe donc sur n'importe quoi (gros danger)

p1 = t; // p1 pointe sur t c'est-a-dire la première case du tableau
// identique à : p1 = &t[0];

p2 = &t[1]; // p2 pointe sur le 2eme élément du tableau

*p1 = 4; // la première case du tableau est modifiée
printf("%d ou %d\n", *p1, t[0]); // affiche 4 ou 4

printf("%d ou %d\n", *p2, t[1]); // affiche 2 ou 2
p2 += 2; // p2 pointe sur le 4eme élément du tableau (indice 3)
printf("%d ou %d\n", *p2, t[3]); // affiche 6 ou 6

// on peut utiliser les [] sur un pointeur :
p1[1] = 8; // identique à : *(p1+1) = 8; ou à : t[1] = 8;
printf("%d\n", t[1]); // affiche 8

// et inversement :
*(t+1) = 10; // identique à : t[1] = 10;
printf("%d\n", t[1]); // affiche 10
```

*Les tableaux et les pointeurs*

## Trier un tableau

La bibliothèque C standard offre une fonction de tri : `qsort()`. Elle sert à trier un tableau d’éléments à l’aide d’une fonction de comparaison à fournir. Pour l’utiliser, vous devez inclure le fichier `<stdlib.h>`.

`qsort()` trie les éléments dans l’ordre que vous lui demandez. Plus précisément, vous donnez à `qsort()` une fonction de comparaison. Cette fonction prend deux éléments A et B, et indique si A doit être avant ou après B dans le tableau trié. La fonction de comparaison est donc vitale, c’est elle qui indique dans quel ordre et selon quels critères il faut trier le tableau.

La fonction de comparaison a le prototype suivant : `int compareValeurs(const void* val1, const void* val2)`.

Elle reçoit en argument deux pointeurs `val1` et `val2`, et retourne un entier. `val1` et `val2` sont des pointeurs constants sur les éléments à comparer. Le `const` empêchera la fonction de comparaison de modifier le contenu du tableau pendant le tri. Le type `void *` est ici obligatoire car la fonction ne peut pas connaître avant le type des éléments à comparer. Elle utilise donc des pointeurs génériques (`void *`) qu’il faudra “caster” (transtyper) vers les types désirés.

✎ Si la fonction de comparaison est `<=`, alors à la fin du tri du tableau `a[0..n-1]` les éléments sont réordonnés de telle manière que `a[0] <= a[1] <= ... <= a[n-1]`. Si on utilise la fonction `>=`, cela donnera `a[0] >= a[1] >= ... >= a[n-1]`.

Seul le signe de la valeur retournée par la fonction de comparaison compte. Par exemple on peut retourner -1 pour A avant B, +1 pour A après B, et 0 (zéro) si l'ordre ne compte pas.

```
int compareEntiers(const void* val1, const void* val2)
{
    int i1 = *(const int*)val1; // on caste sur un type entier
    int i2 = *(const int*)val2; // on caste sur un type entier

    if (i1 == i2)
        return 0;
    if (i1 < i2)
        return -1;
    else
        return +1;
}
```

*Fonction de comparaison de deux entiers*

En pratique, pour trier un tableau `valeurs` contenant `nbValeurs` éléments, on donnera à la fonction `qsort()` les arguments suivants :

- Le nom du tableau, `valeurs`
- Le nombre d'éléments du tableau, `nbValeurs`
- La taille en octets de chaque élément, `sizeof(valeurs[0])`
- Le nom de la fonction de comparaison, `compareEntiers`

```
#include <stdlib.h>

int compareEntiers(const void* val1, const void* val2)
{
    int i1 = *(const int*)val1;
    int i2 = *(const int*)val2;
    if (i1 == i2)
        return 0;
    if (i1 < i2)
        return -1;
    else
        return +1;
}

qsort(valeurs, nbValeurs, sizeof(valeurs[0]), compareEntiers);
```

*Tri d'un tableau en C*

✎ Pour rechercher des valeurs dans un tableau, vous pouvez utiliser la fonction `bsearch()` de la bibliothèque C standard, déclarée dans le fichier `<stdlib.h>`. Cette fonction, qui ressemble beaucoup à `qsort()`, permet d'effectuer une recherche dichotomique. `bsearch()` recherche une valeur dans un tableau initialement trié et prend en argument l'objet cherché (la clé), le tableau, et la fonction de comparaison qui permet de savoir si un élément donné est situé avant ou après la clé, ou bien est égal à la clé.

☞ En C++, il existe aussi une fonction `sort()` pour le tri de conteneurs.

## Les tableaux dynamiques

Pour allouer dynamiquement des tableaux en C, il faut utiliser les fonctions `malloc()` et `free()` :

```
int *T; // pointeur sur un entier

// allocation dynamique d'un tableau de 10 int (un int est codé sur 4 octets)
T = (int *)malloc(sizeof(int) * 10); // alloue 4 * 10 octets en mémoire

// initialise le tableau avec des 0
for(int i=0;i<10;i++)
{
    *(T+i) = 0; // les 2 écritures sont possibles
    T[i] = 0; // identique à la ligne précédente
}
// ou avec la fonction memset
memset(T, 0, sizeof(int)*10); // il faudra alors inclure string.h

// libération de la mémoire
free(T);
```

*Allocation dynamique en C*

✎ La fonction `realloc()` modifie la taille d'un bloc de mémoire déjà alloué.

Pour allouer dynamiquement des tableaux en C++, il faut utiliser les opérateurs `new` et `delete` :

```
#include <iostream>
#include <new>

using namespace std;

int main ()
{
    int *t = new int[5]; // alloue un tableau de 5 entiers en mémoire

    // initialise le tableau avec des 0
    for(int i=0;i<5;i++)
    {
        *(t + i) = 0; // les 2 écritures sont possibles
        t[i] = 0; // identique à la ligne précédente
        cout << "t[" << i << "] = " << t[i] << endl;
    }

    delete [] t; // libère la mémoire allouée

    return 0;
}
```

*Allocation dynamique en C++*

☞ En C++, il existe aussi des tableaux dynamiques “prêts à l'emploi” avec le type `vector`.

## Questions de révision

L'idée de base des questions de révision est de vous donner une chance de voir si vous avez identifié et compris les points clés de cette partie.

**Question 21.** Sur combien de bits est codé un caractère ASCII ?

**Question 22.** Quelle est la taille en bits d'une variable de type `char` ?

**Question 23.** Quelle est la valeur en hexadécimale du caractère ASCII 'A' ?

**Question 24.** Quelle est la différence entre une chaîne de caractères et un tableau de caractères ?

**Question 25.** Quelle est la taille en caractères de la chaîne de caractères "hello" ?

**Question 26.** Quelle est la valeur de l'indice pour accéder à la première case d'un tableau ?

**Question 27.** Quelle est la taille en octets d'un tableau de 100 entiers (`int`) ?

**Question 28.** Dans la déclaration `int t[10]` ? Que représente réellement la variable `t` ?

**Question 29.** Que fait l'instruction `t[5] = 2;` ?

**Question 30.** Que provoque l'instruction `t[10] = 2;` ?

## Conclusion

Les types sont au centre de la plupart des notions de programmes corrects, et certaines des techniques de construction de programmes les plus efficaces reposent sur la conception et l'utilisation des types.

**Rob Pike** (ancien chercheur des Laboratoires Bell et maintenant ingénieur chez Google) :

*« Règle n°5 : Les données prévalent sur le code. Si vous avez conçu la structure des données appropriée et bien organisé le tout, les algorithmes viendront d'eux-mêmes. La structure des données est le coeur de la programmation, et non pas les algorithmes. »*

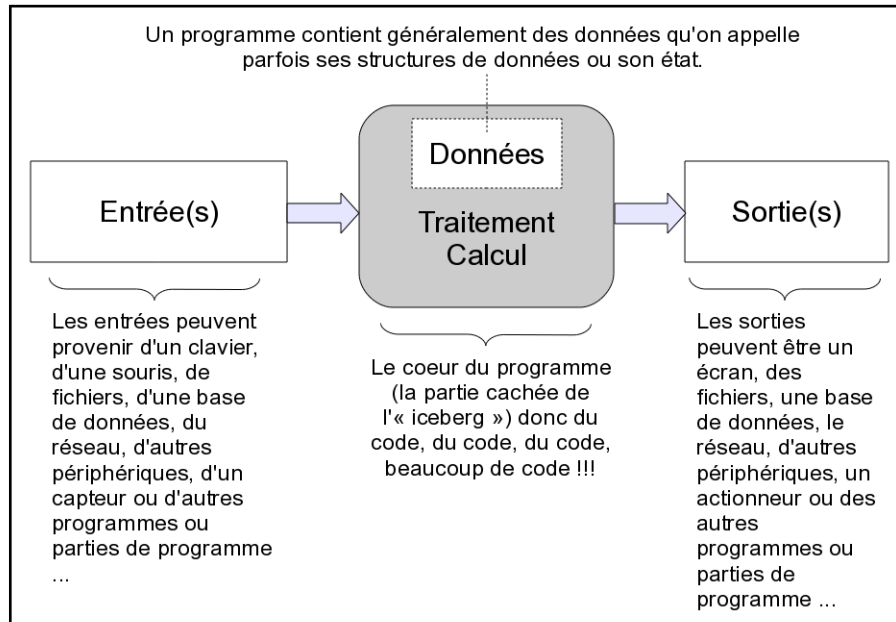
Cette règle n°5 est souvent résumée par « Écrivez du code stupide qui utilise des données futées ! ».



# Les fonctions

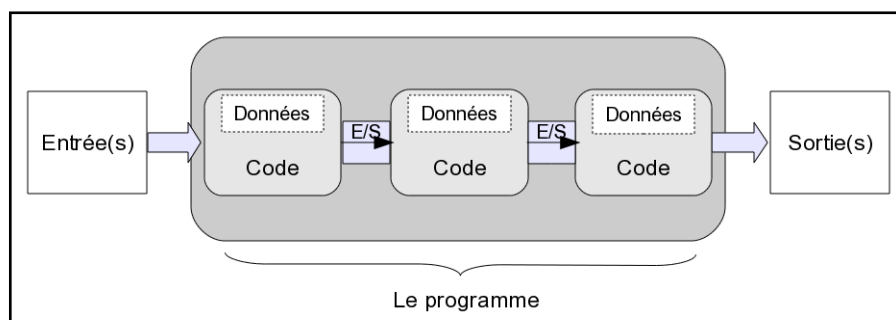
## Programmation procédurale

D'un certain point de vue, un programme informatique ne fait jamais rien d'autre que **traiter des données**. Comme on l'a déjà vu, un programme accepte des entrées et produit des sorties :



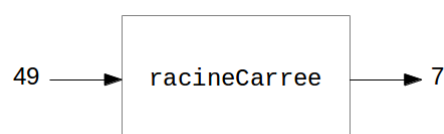
✎ *Un **traitement** est tout simplement l'action de produire des sorties à partir d'entrées. Les entrées dans une partie de programme sont souvent appelées des **arguments** (ou **paramètres**) et les sorties d'une partie de programme des **résultats**.*

La majeure partie du travail d'un programmeur est : comment exprimer un programme sous la forme d'un ensemble de parties (des sous-programmes) qui coopèrent et comment peuvent-elles partager et échanger des données ?



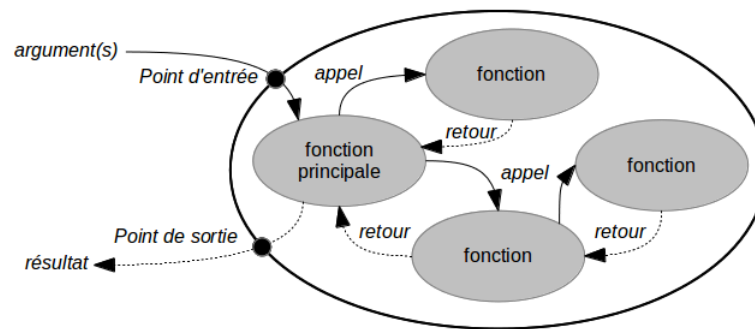
Par parties de programme (ou de code), on entend des entités (ou modules) comme une **fonction** produisant un résultat à partir d'un ensemble d'arguments en entrée.

☞ *Exemple : un traitement comme produire le résultat (sortie) 7 à partir de l'argument (entrée) 49 au moyen de la fonction **racineCarree**.*



En langage C, on pratique la **programmation procédurale**. La programmation procédurale se fonde sur le concept d'appel de **procédure**. Une procédure, aussi appelée **routine**, **sous-routine** ou **fonction**, contient simplement une série d'étapes à réaliser. Un **appel** de procédure (ou de fonction) déclenche l'exécution de la série d'instructions qui la compose.

En programmation procédurale, un programme n'est plus une simple séquence d'instructions mais un ensemble de sous-programmes (en C, des fonctions) s'appelant entre eux :



Le déroulement d'un programme est le suivant :

- L'exécution du programme commence toujours par l'exécution de la fonction principale (la fonction `main()` en C/C++)
- L'appel à une fonction permet de déclencher son exécution, en interrompant le déroulement séquentiel des instructions de la fonction principale
- Le déroulement des instructions de la fonction principale reprend, dès que la fonction appelée est terminée, à l'instruction qui suit l'appel

✎ En C++, on aura la possibilité de développer un programme en **Programmation Orientée Objet (POO)**.

➡ Décomposer un programme en **fonctions** conduit souvent à diminuer la complexité d'un problème et permet de le résoudre plus facilement.

L'utilisation des fonctions permettra :

- d'éviter de répéter plusieurs fois les mêmes lignes de code : ceci simplifie le code, améliore la lisibilité et facilite la maintenance.
- de généraliser certaines parties de code : la décomposition en fonction permettra la réutilisation du code dans d'autres contextes (cf. bibliothèque logicielle).

On décomposera un programme en fonctions parce que procéder ainsi :

- rend le traitement distinct du point de vue logique
- rend le programme plus clair et plus lisible
- permet d'utiliser la fonction à plusieurs endroits dans un programme (à chaque fois qu'on en a besoin)
- facilitera les tests (on simulera des entrées et on comparera le résultat obtenu à celui attendu)

Bonnes pratiques :

- les programmes sont généralement plus facile à écrire, à comprendre et à maintenir lorsque chaque fonction réalise une SEULE ACTION logique et bien évidemment celle qui correspond à son nom ;
- on limitera la taille des fonctions à une valeur comprise entre **10 à 15 lignes maximum** ;
- les fonctions se concentrent sur le traitement qu'elles doivent réaliser et n'ont pas (la plupart du temps) à réaliser la saisie de leurs entrées et l'affichage de leur sortie. On évitera le plus possible d'utiliser des saisies et des affichages dans les fonctions pour permettre notamment leur ré-utilisation.

## Fonction vs Procédure

⇒ Il existe deux types de sous-programmes :

- Les **fonctions**
  - Sous-programme qui retourne **une et une seule valeur** : permet de ne récupérer qu'**un résultat**.
  - Par convention, ce type de sous-programme ne devrait pas interagir avec l'environnement (écran, utilisateur).
- Les **procédures**
  - Sous-programme qui ne retourne **aucune valeur** : permet de produire **0 à n résultats**
  - Par convention, ce type de sous-programme peut interagir avec l'environnement (écran, utilisateur).

⚠ Cette distinction ne se retrouve pas dans tous les langages de programmation ! Le C/C++ n'admet que le concept de fonction qui servira à la fois pour écrire des fonctions et des procédures.

Pour les fonctions en C/C++, il faut distinguer :

- la **déclaration** qui est une instruction fournissant au compilateur un certain nombre d'informations concernant une fonction (qui déclare son existence). Il existe une forme recommandée dite **prototype** :  
`int plus(int, int);` ← fichier en-tête (.h)
- la **définition** qui revient à écrire le **corps** de la fonction (qui définit donc les traitements effectués dans le bloc {} de la fonction)  
`int plus(int a, int b) { return a + b; }` ← fichier source (.c ou .cpp)
- l'**appel** qui est son utilisation. Il doit correspondre à la déclaration faite au compilateur qui le vérifie.  
`int res = plus(2, 2);` ← fichier source (.c ou .cpp)

⚠ La définition d'une fonction tient lieu de déclaration. Mais elle doit être "connue" avant son utilisation (un appel). Sinon, le compilateur **gcc** génèrera un message d'avertissement (warning) qui indiquera qu'il a lui-même fait une déclaration implicite de cette fonction (ce n'est pas une bonne chose). Par contre, le compilateur **g++** génèrera une erreur : `'...' was not declared in this scope`.

🔍 C/C++ supporte la **récursivité** : c'est une technique qui permet à une fonction de s'auto-appeler. La récursivité est une manière simple et élégante de résoudre certains problèmes algorithmiques, notamment en mathématique, mais cela ne s'improvise pas !

## Déclaration de fonction

Une fonction se caractérise par :

- son **nom** (un identificateur)
- sa **liste de paramètre(s)** : le nombre et le type de paramètre(s) (la liste peut être vide)
- son **type de retour** (un seul résultat)

⚠ Comme une procédure ne retourne aucune valeur, son type de retour sera **void**.

Ces informations sont communément appelées le **prototype** de la fonction :

```
// Le prototype de la fonction calculeNombreDeSecondes :  
int calculeNombreDeSecondes(int heures, int minutes, int secondes)  
  
// Soit :  
// - son nom : calculeNombreDeSecondes  
// - sa liste de paramètre(s) : elle reçoit 3 int  
// - son type de retour : int
```

C'est ce qu'il est nécessaire de connaître pour appeler une fonction.

Quand une fonction n'est pas encore définie, il est possible de **déclarer** son existence afin de pouvoir l'appeler. Il faut pour cela indiquer son prototype, suivi d'un point-virgule :

```
// La déclaration de la fonction calculeNombreDeSecondes :  
int calculeNombreDeSecondes(int heures, int minutes, int secondes);
```

☞ Les déclarations de fonction sont placées dans des fichiers d'en-tête (header) d'extension **.h**. Si vous voulez utiliser (i.e. appeler) une fonction, il faudra donc inclure le fichier d'en-tête correspondant (**#include**). Un fichier d'en-tête regroupe un ensemble de déclarations.

```
#ifndef TEMPS_H /* si l'étiquette TEMPS_H n'est pas défini */  
#define TEMPS_H /* alors on définit l'étiquette TEMPS_H */  
  
int calculeNombreDeSecondes(int heures, int minutes, int secondes);  
  
#endif /* fin si TEMPS_H */
```

*Le fichier d'en-tête temps.h*

⚠ Le langage C n'interdit pas l'inclusion multiple de fichiers d'en-tête mais il n'accepte pas toujours de déclarer plusieurs fois la même chose ! Par précaution, il faut donc s'assurer par des directives de pré-compilation (**#ifndef**, **#define** et **#endif**) que l'inclusion du fichier sera unique. Les directives de pré-compilation (ou préprocesseur) commencent toujours par un dièse (**#**). Ce ne sont donc pas des instructions du langage C.

Si vous voulez utiliser (i.e. appeler) une fonction, il faudra donc inclure le fichier d'en-tête correspondant :

```
#include "temps.h"  
  
// Vous pouvez maintenant utiliser (appeler) la fonction calculeNombreDeSecondes :  
int s = calculeNombreDeSecondes(1, 0, 0);
```

☞ Si vous créer vos propres fichiers d'en-tête, il est conseillé d'indiquer le nom du fichier entre guillemets ("**temps.h**") dans la directive de pré-compilation **#include**. Vous pourrez indiquer le chemin où se trouve vos propres fichiers d'en-tête avec l'option **-I** du compilateur **gcc/g++**. On n'utilisera pas les délimiteurs **<>** (**#include <stdio.h>**) qui sont réservés pour les fichiers d'en-tête systèmes. Dans ce cas, le compilateur connaît les chemins d'installation de ces fichiers.

```
$ ls  
main.c temps.c include/
```

```
$ ls include/  
temps.h
```

```
$ gcc -I./include -c temps.c  
$ gcc -I./include -c main.c
```

## Définition de fonction

La **définition** d'une fonction revient à écrire le **corps** de la fonction dans le bloc `{}`. Cela définira la suite d'instructions qui sera exécutée à chaque appel de la fonction.

```
#include "temps.h"

// La définition de la fonction calculeNombreDeSecondes :
int calculeNombreDeSecondes(int heures, int minutes, int secondes)
{
    return ((heures*3600) + (minutes*60) + secondes);
}
```

⚠ Attention à ne pas mettre de point-virgule à la fin du prototype ! Si vous déclarez votre propre fonction, il vous faudra absolument la définir si vous voulez passer l'étape de l'édition de lien (ld). Sinon, vous obtiendrez une erreur : **undefined reference**.

Pour que la fonction puisse effectivement retourner une valeur, il faut qu'elle contienne une instruction composée du mot-clé **return** et de la valeur que l'on veut retourner. L'instruction **return** quitte la fonction et transmet la valeur qui suit au programme appelant.

🔗 Dans le cas des fonctions dont le type de retour est **void**, il est également possible d'utiliser l'instruction **return**. Elle n'est alors suivie d'aucune valeur et a simplement pour effet de quitter la fonction immédiatement.

## Appel de fonction

Un appel à une fonction signifie qu'on lui demande d'exécuter son traitement :

```
// Appel de la fonction calculeNombreDeSecondes :
int s = calculeNombreDeSecondes(1, 0, 0);
```

Soit la fonction **carre()** ci-dessous :

```
// Déclaration :
// Calcule et retourne le carré d'un nombre
int carre(int); // le nom du paramètre n'est pas obligatoire

// Définition :
// Calcule et retourne le carré d'un nombre
int carre(int x) // le nom du paramètre est obligatoire
{
    // le paramètre x est une variable locale à la fonction
    return x*x;
}
```

L'appel doit correspondre à la déclaration faite au compilateur qui le vérifie :

```
// Principe : on n'est pas obligé d'utiliser le résultat de retour mais on doit donner à la
// fonction exactement les arguments qu'elle exige

int c5 = carre(2); // Correct

int valeur = 2;
int c6 = carre(valeur); // Correct (valeur est copié dans x)
```

```
// Mais on peut faire des erreurs :

int c1 = carre(); // Erreur : pas assez d'argument

int c2 = carre; // Erreur : parenthèses manquantes

int c3 = carre(1, 2); // Erreur : trop d'arguments

int c4 = carre("deux"); // Erreur : mauvais type d'argument car un int attendu

// Attention :
carre(2); // probablement une erreur car la valeur retour n'est pas utilisée

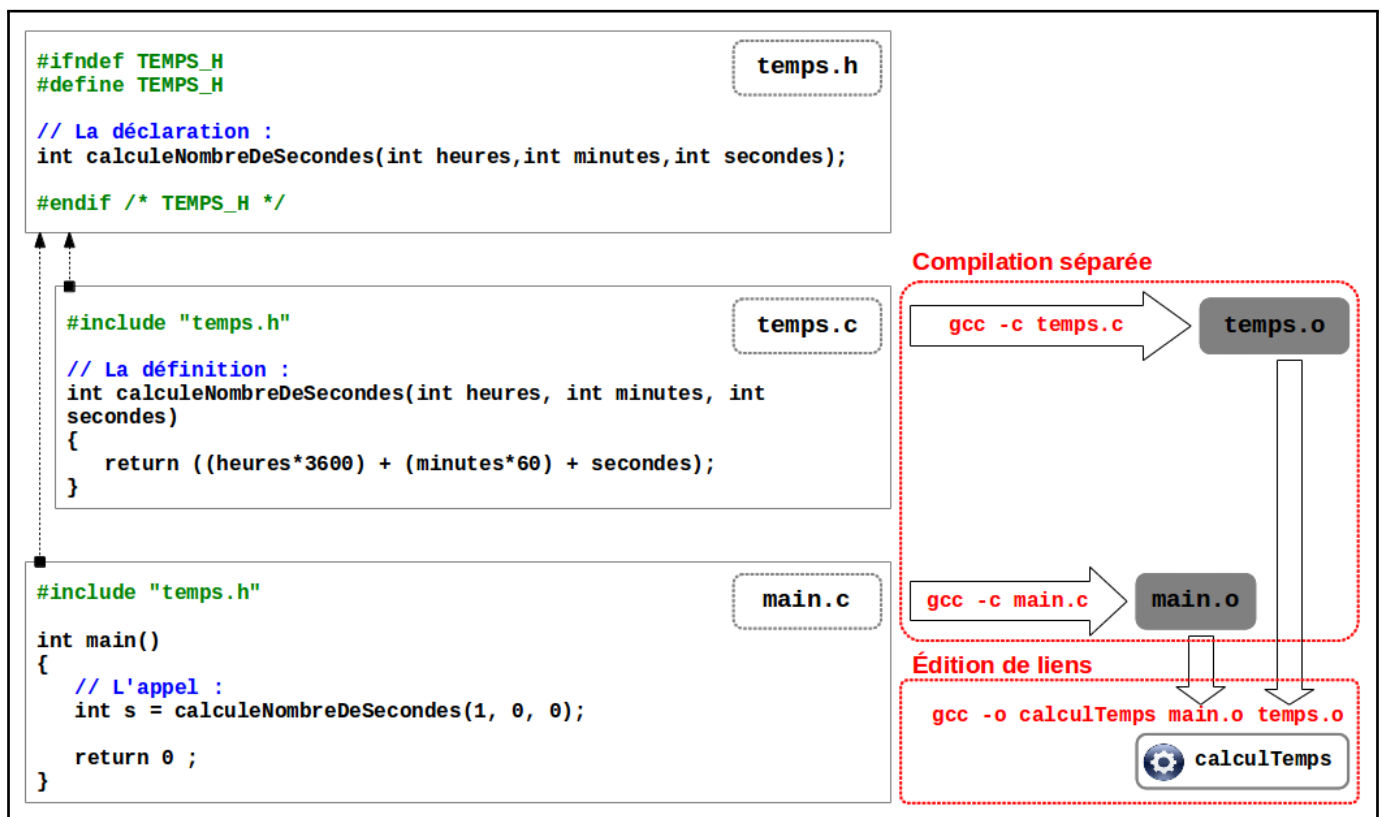
// Soyez vigilant :
int x = 2;
int c7 = carre(x); // Correct (x est copié dans x)
// après l'appel : x contient toujours 2 et c7 contient 4
```

Une fonction peut être utilisée :

- comme un **opérande** dans une expression, à partir du moment où il y a concordance de types;
- comme une **instruction**.

## Programmation modulaire

Le découpage en fonctions d'un programme permet la programmation modulaire :



Par la suite, un utilitaire comme **make** permettra d'automatiser la fabrication de programme. Il utilise un fichier de configuration appelé *makefile* qui porte souvent le nom de **Makefile**. Ce fichier décrit des

cibles (qui sont souvent des fichiers, mais pas toujours), de quelles autres cibles elles dépendent, et par quelles actions (des commandes) y parvenir. Au final, le fichier **Makefile** contient l'ensemble des règles de fabrication du programme :

```
CC=gcc
RM=rm

TARGET := calculTemps

all: $(TARGET)

main.o: main.c temps.h
    $(CC) -o $@ -c $<

temps.o: temps.c temps.h
    $(CC) -o $@ -c $<

$(TARGET): main.o temps.o
    $(CC) -o $(TARGET) $^

clean:
    $(RM) -f $(TARGET) *.o *~
```

*Exemple de fichier Makefile*

La fabrication de l'exécutable est assurée ensuite par l'utilitaire **make** :

```
$ make
gcc -o main.o -c main.c
gcc -o temps.o -c temps.c
gcc -o calculTemps main.o temps.o

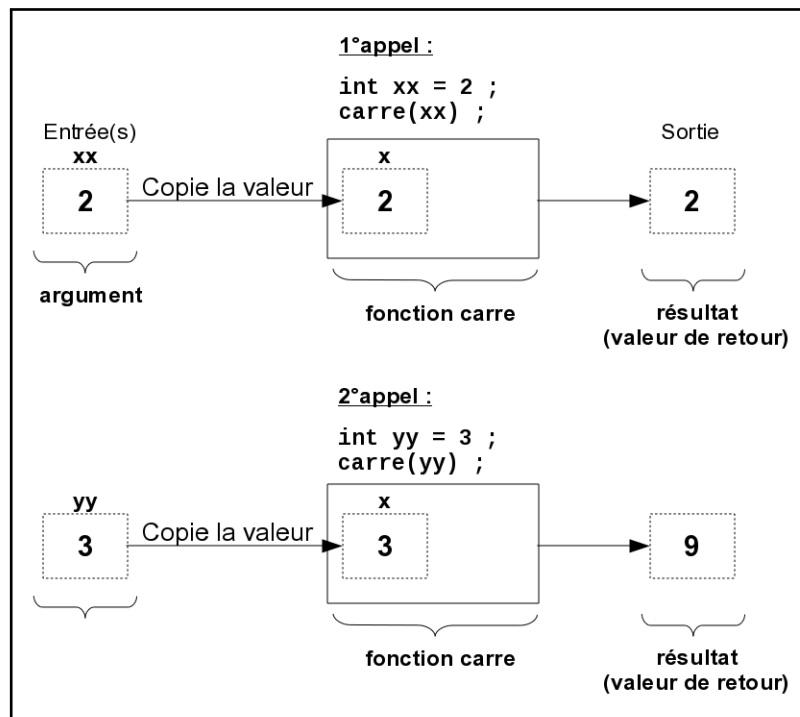
$ ./calculTemps
```

☞ Avec son système de dépendance, **make** ne compile que ce qui est nécessaire. Lire : <https://fr.wikipedia.org/wiki/Make>.

## Passage de paramètre(s)

### Passage par valeur

Lorsque l'on passe une valeur en paramètre à une fonction, cette valeur est copiée dans une variable locale de la fonction correspondant à ce paramètre. Cela s'appelle un **passage par valeur**.



Passage par valeur

🕒 Dans un passage par valeur, il est impossible pour une fonction de modifier les paramètres qu'elle reçoit. Essayons de permuter deux variables :

```
// Tente de permuter deux entiers
void permute(int a, int b)
{
    printf("avant [permute] : a=%d et b=%d\n", a, b); // affiche : a=5 et b=3
    int temp = a;
    a = b;
    b = temp;
    printf("après [permute] : a=%d et b=%d\n", a, b); // affiche : a=3 et b=5
}

int main()
{
    int a = 5, b = 3;
    printf("avant [main] : a=%d et b=%d\n", a, b); // affiche : a=5 et b=3
    permute(a, b);
    printf("après [main] : a=%d et b=%d\n", a, b); // affiche : a=5 et b=3
    return 0;
}
```

Passage par valeur

🔗 Les variables **a** et **b** sont locales au bloc (`{}`) où elles sont définies. Des variables définies dans des blocs différents peuvent porter le même nom.

## Passage par adresse

Pour permettre à une fonction de modifier les paramètres qu'elle reçoit, il faudra passer les adresses des variables comme paramètres. On utilise dans ce cas des pointeurs. Cela s'appelle un **passage par adresse** :



```
// Permute deux entiers
void permute(int *pa, int *pb)
{
    printf("avant [permute] : a=%d et b=%d\n", *pa, *pb); // affiche : a=5 et b=3
    int temp = *pa;
    *pa = *pb;
    *pb = temp;
    printf("après [permute] : a=%d et b=%d\n", *pa, *pb); // affiche : a=3 et b=5
}

int main()
{
    int a = 5, b = 3;

    printf("avant [main] : a=%d et b=%d\n", a, b); // affiche : a=5 et b=3
    permute(&a, &b); // passe l'adresse des variables a et b
    printf("après [main] : a=%d et b=%d\n", a, b); // affiche : a=3 et b=5

    return 0;
}
```

### *Passage par adresse*

Le passage d'un **tableau** en paramètre d'une fonction est évidemment possible :

```
#include <stdio.h>

void raz(int t[], int n) /* équivalent à : void raz(int *t, int n) */
{
    int i;

    for(i=0; i<n; i++)
        t[i] = 0;
}

int main()
{
    int t[2] = { 2, 3 }; // 2 éléments

    printf("avant [main] : t[0] = %d et t[1] = %d\n", t[0], t[1]); // affiche t[0] = 2 et t
    [1] = 3
    raz(t, 2); // équivalent à : raz(&t[0], 2);
    printf("après [main] : t[0] = %d et t[1] = %d\n", t[0], t[1]); // affiche t[0] = 0 et t
    [1] = 0

    return 0;
}
```

### *Passage d'un tableau en paramètre d'une fonction*

Le tableau passé en paramètre a bien été modifié car la fonction `raz()` à travailler avec l'adresse du tableau. Les cases du tableau n'ont pas été recopiées et la fonction accède au tableau original. On rappelle qu'il n'existe pas de variable désignant un tableau comme un tout. Quand on déclare `int t[2]`, `t` ne désigne pas l'ensemble du tableau mais l'adresse de la première case. `t` est une **constante de type pointeur** vers un `int` dont la valeur est `&t[0]`, l'adresse du premier élément du tableau.

☞ C'est une très bonne chose en fait car dans le cas d'un "gros tableau", on évite ainsi de recopier toutes les cases. Le passage par adresse sera beaucoup plus efficace et rapide.

## Passage par référence

En C++, il existe aussi un **passage par référence** :

```
// Permute deux entiers
void permute(int &ra, int &rb)
{
    printf("avant [permute] : a=%d et b=%d\n", ra, rb); // affiche : a=5 et b=3
    int temp = ra;
    ra = rb;
    rb = temp;
    printf("après [permute] : a=%d et b=%d\n", ra, rb); // affiche : a=3 et b=5
}

int main()
{
    int a = 5, b = 3;

    printf("avant [main] : a=%d et b=%d\n", a, b); // affiche : a=5 et b=3
    permute(a, b);
    printf("après [main] : a=%d et b=%d\n", a, b); // affiche : a=3 et b=5
    return 0;
}
```

*Passage par référence (C++)*

☞ Intérêt : lorsqu'on passe des variables en paramètre de fonctions et que le coût d'une recopie par valeur est trop important, on choisira un passage par référence. Si le paramètre ne doit pas être modifié, on utilisera alors un passage par référence sur une variable constante :

```
void foo(const long double &a) // la référence est déclarée const
{
    printf("J'ai un accès en lecture : a = %.1Lf\n", a);
    printf("mais pas en écriture !\n");
    //a = 3.5; // interdit car déclarée const ! erreur: assignment of read-only reference 'a'
}

int main()
{
    long double a = 2.0;

    printf("Je suis une grosse variable de taille %d octets\n", sizeof(a)); // affiche Je
        suis une grosse variable de taille 12 octets
    printf("Je suis a et j'ai pour valeur : a = %.1Lf\n", a); // affiche Je suis a et j'ai
        pour valeur : a = 2.0
    foo(a);

    return 0;
}
```

*Passage par référence constante (C++)*

## Valeur de retour

Par définition, une fonction fournit un **résultat**. Pour cela, on lui déclare un **type de retour** et on renvoie une valeur (du type déclaré) avec l'instruction **return**. Cette instruction provoque évidemment la sortie de la fonction appelée. La valeur de retour peut servir à **renvoyer un résultat** ou **un état sur l'exécution** de la fonction appelée.

Certains programmeurs utilisent donc la valeur de retour pour indiquer si la fonction a réalisé son traitement avec succès ou si elle a rencontré une erreur. C'est le cas de beaucoup de fonctions systèmes. On distingue différentes pratiques pour la convention du type de retour :

- valeur de retour de type booléen (`bool`) : `true` (succès) ou `false` (erreur)
- valeur de retour de type entière (`int`) : `0` (succès) ou `!=0` (un code d'erreur)
- valeur de retour de type entière (`int`) : `1` (succès) ou `<0` (un code d'erreur)

⚠ Il ne faut jamais négliger ce type de code de retour lorsqu'on utilise une fonction. Prenons un exemple : vous appelez une fonction pour créer un répertoire (`man 2 mkdir`) puis une fonction pour copier un fichier dans ce répertoire. Si la création du répertoire échoue, cela ne sert à rien d'essayer de copier ensuite le fichier. La réussite (ou l'échec) de la première action conditionne l'exécution de la deuxième action. Si vous ne testez jamais les codes de retour de fonction, vous allez provoquer des cascades d'erreurs.

Dans certaines situations extrêmes, vous risquer d'avoir besoin de sortir immédiatement du programme quelque soit l'endroit où vous êtes. Vous pouvez utiliser la fonction `exit()` qui termine normalement un programme en indiquant une valeur de retour. Évidemment, la fonction `exit()` ne revient jamais.

```
#include <stdio.h>
#include <stdlib.h> /* pour exit() */
#include <sys/stat.h> /* pour mkdir() */
#include <sys/types.h>

int main()
{
    mode_t mode = 0750;
    int retour;

    retour = mkdir("./dossier", mode); // Création d'un répertoire
    if(retour == -1) // Une erreur s'est-elle produite ?
    {
        // La fonction perror() affiche un message d'erreur décrivant la dernière erreur
        // rencontrée durant un appel système ou une fonction de bibliothèque
        perror("mkdir");
        // Cela ne sert à rien de continuer, on quitte le programme en indiquant qu'on a
        // rencontré une erreur
        exit(EXIT_FAILURE);
    }

    return EXIT_SUCCESS; // sortir de la fonction main() revient à quitter le programme
}
```

*Quitter un programme*

Cela donne :

```
$ gcc mkrep.c
```

```
$ ./a.out
```

```
$ ls -l
-rwxrwxr-x 1 tv tv 8,3K août 9 10:47 a.out*
drwxr-x--- 2 tv tv 4,0K août 9 10:47 dossier/
-rw-rw-r-- 1 tv tv 329 août 9 10:47 mkrep.c

$ ./a.out
mkdir: File exists
```

## Nommer une fonction

Un nom de fonction est construit à l'aide d'un **verbe** (surtout pas un nom), et éventuellement d'éléments supplémentaires comme :

- une quantité
- un complément d'objet
- un adjectif représentatif d'un état

☞ *Une fonction est composée d'une série d'instructions qui effectue un traitement. Il faut donc utiliser un verbe pour caractériser l'action réalisée par la fonction.*

On utilisera la convention suivante : **un nom de fonction commence par une lettre minuscule puis les différents mots sont repérés en mettant en majuscule la première lettre d'un nouveau mot.**

Le verbe peut être au présent de l'indicatif ou à l'infinitif. L'adjectif représentatif d'un état concerne surtout les fonctions booléennes. La quantité peut, le cas échéant, enrichir le sens du complément.

Exemples : `void ajouter()`, `void sauverValeur()`, `estPresent()`, `estVide()`, `videAMoitieLeReservoir()`, ...

⚠ *Les mots clés du langage sont interdits comme noms.*

Les noms des paramètres d'une fonction sont construits comme les noms de variables : ils commencent, notamment par une minuscule. L'ordre de définition des paramètres doit respecter la règle suivante :

`nomFonction(parametrePrincipal, listeParametres)`

où `parametrePrincipal` est la donnée principale sur laquelle porte la fonction, la `listeParametres` ne comportant que des données secondaires, nécessaires à la réalisation du traitement réalisé par la fonction.

Exemple : `ajouter(float mesures[], float uneMesure)`

Le rôle de cette fonction est d'ajouter une **mesure** à un ensemble de **mesures** (qui est la donnée principale) et non, d'ajouter un ensemble de **mesures** à une **mesure**.

✍ *Bonne pratique : L'objectif de la programmation procédurale est de décomposer un traitement de grande taille en plusieurs parties plus petites jusqu'à obtenir quelque chose de suffisamment simple à comprendre et à résoudre (cf. Descartes et son discours de la méthode). Pour cela, on va **s'obliger à limiter la taille de nos fonctions à une valeur comprise entre 10 à 15 lignes maximum** (accolades exclues).*

## La fonction main

Tout programme C/C++ doit posséder une **fonction** nommée **main** (dite fonction principale) pour indiquer où commencer l'exécution. La fonction `main()` représente le point d'entrée (et de sortie) d'un programme exécuté par le "système" (c'est-à-dire le système d'exploitation).

```
// Forme simplifiée :
int main()
{

    return 0; // on doit retourner une valeur entière
}

// Forme normalisée :
int main(int argc, char **argv)
{

    return 0; // on doit retourner une valeur entière
}

int main(int argc, char *argv[])
{

    return 0; // on doit retourner une valeur entière
}
```

*La fonction main()*

➡ **Paramètres d'entrée** : la fonction `main()` reçoit deux paramètres (ils peuvent être ignorés) : un entier (`int`) qui précisera le nombre d'arguments passés au programme et un tableau de chaînes de caractères (`char **` ou `char *[]`) qui contiendra la liste de ses arguments.

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int i;

    printf("nb d'arguments = %d\n", argc);

    for(i=0; i<argc; i++)
        printf("argv[%d] = %s\n", i, argv[i]);

    //...
    return 0;
}
```

*Les arguments d'un programme*

Ce qui donne :

```
$ ./a.out les parametres du programme
nb d'arguments = 5
argv[0] = ./a.out
argv[1] = les
argv[2] = parametres
argv[3] = du
argv[4] = programme

$ ./a.out
nb d'arguments = 1
argv[0] = ./a.out
```

☞ Il existe une fonction `getopt()` qui permet d'analyser plus facilement les options d'une ligne de commande (*man 3 getopt*).

➡ **Valeur de retour** : la fonction `main()` doit retourner une valeur entière (`int`). Sur certains systèmes (Unix/Linux), elle peut servir à vérifier si le programme s'est exécuté correctement. Un zéro (0) indique alors que le programme s'est terminé avec succès (c'est une convention). Évidemment, une valeur différente de 0 indiquera que le programme a rencontré une erreur. Et sa valeur précisera alors le type de l'erreur. Sur un système Unix/Linux, la dernière valeur de retour reçue est stockée dans une variable `$?` du *shell*.

```
#include <stdio.h>

int main(int argc, char **argv)
{
    // Exemple : on peut vérifier que l'on reçu le nombre suffisant d'arguments
    if (argc < 2)
        return 1;
    //...
    return 0;
}
```

*La valeur de retour d'un programme*

Ce qui donne :

```
$ ./a.out les parametres du programme
```

```
// Affichage de la valeur de retour
```

```
$ echo $?
```

```
0
```

```
// 0=succès
```

```
$ ./a.out
```

```
// Affichage de la valeur de retour
```

```
$ echo $?
```

```
1
```

```
// 1=erreur
```

## Intérêt des fonctions par l'exemple

Pour présenter l'intérêt d'utiliser des fonctions, voici un exemple de programme en pseudo-code :

```
Variable prix : RÉEL
Variable quantité : ENTIER
Variable total_ttc : RÉEL = 0.00
Constante tva : RÉEL = 19.60

Début
  -- début des achats
  Ecrire "*****"

  -- achat de 3 articles à 10,50 euros HT chacun
  prix <- 10.50
  quantité <- 3
  total_ttc <- total_ttc + tva * ( prix * quantité )
  Ecrire "Sous-total TTC : " & total_ttc

  -- achat de 2 articles à 21,30 euros HT chacun
  prix <- 21.30
  quantité <- 2
  total_ttc <- total_ttc + tva * ( prix * quantité )
  Ecrire "Sous-total TTC : " & total_ttc

  -- achat d'1 article à 2,40 euros HT
  prix <- 2.40
  quantité <- 1
  total_ttc <- total_ttc + tva * ( prix * quantité )
  Ecrire "Sous-total TTC : " & total_ttc

  -- fin des achats
  Ecrire "*****"

  -- total des achats TTC
  Ecrire "#####"
  Ecrire "Total TTC : " & total_ttc
  Ecrire "#####"
Fin
```

➡ Pour les 3 achats de l'exemple, le calcul est répété à chaque fois. Ce qui n'est pas pratique pour la maintenance du code : si la formule n'est pas la bonne (une erreur est toujours possible) ou si la formule doit être modifiée (autre mode de calcul), il faudra changer le code en plusieurs endroits pour un même type de calcul.

On va donc créer une fonction qui calcule le prix TTC pour l'achat d'une certaine quantité d'articles, et le retourne au programme appelant :

```
const double tva = 19.6;

double calculePrixTTC(double prix, int quantite)
{
    return tva * (prix * quantite);
}
```

Celui-ci utilise le prix TTC retourné pour l'ajouter au total :

```
int main()
{
    double total_ttc = 0.;

    // ...
    total_ttc += calculePrixTTC(10.50, 3); // achat de 3 articles à 10,50 euros HT chacun
    // ...

    // ...
    total_ttc += calculePrixTTC(21.30, 2); // achat de 2 articles à 21,30 euros HT chacun
    // ...

    // ...
    total_ttc += calculePrixTTC(2.40, 1); // achat d'1 article à 2,40 euros HT
    // ...
}
```

⇒ L'affichage des '\*' et des dièses '#' pourrait lui aussi être factorisé en fonction. De manière naïve, on pourrait faire :

```
void afficher40Etoiles()
{
    for (i = 0; i < 40; i++)
    {
        printf("*");
    }
    printf("\n");
}

void afficher30Dieses()
{
    for (i = 0; i < 30; i++)
    {
        printf("#");
    }
    printf("\n");
}
```

On pourrait surtout améliorer le pseudo-code comme ceci :

```
...
Début
    ...
    Afficher 40 "*"
    Ecrire "Sous-total TTC : " & total_ttc
    ...
    Afficher 40 "*"
    ...
    Afficher 30 "#"
    Ecrire "Total TTC : " & total_ttc
    Afficher 30 "#"
Fin
```

On va donc créer une fonction paramétrable avec 2 arguments : le **caractere** à afficher et **nb** fois qu'il faut l'afficher :



```
void afficherCaracteres(char caractere, int nb)
{
    for (i = 0; i < nb; i++)
    {
        printf("%c", caractere);
    }
    printf("\n");
}

// Pour afficher les 40 étoiles :
afficherCaracteres('*', 40);

// Pour afficher les 30 dièses :
afficherCaracteres('#', 30);
```

Au final, on obtient le code source suivant :

```
#include <stdio.h>

const double tva = 19.6;

double calculePrixTTC(double prix, int quantite)
{
    return (1. + tva/100.) * (prix * (double)quantite);
}

void afficherCaracteres(char caractere, int nb)
{
    int i;

    for (i = 0; i < nb; i++)
    {
        printf("%c", caractere);
    }
    printf("\n");
}

void afficherTotalTTC(double total_ttc)
{
    afficherCaracteres('#', 30);
    printf("Total TTC : %.3f euros\n", total_ttc);
    afficherCaracteres('#', 30);
}

int main()
{
    double total_ttc = 0.;

    // début des achats
    afficherCaracteres('*', 40);

    // achat de 3 articles à 10,50 euros HT chacun
    total_ttc += calculePrixTTC(10.50, 3);
    printf("Sous-total TTC : %.2f euros\n", total_ttc);
```

```
// achat de 2 articles à 21,30 euros HT chacun
total_ttc += calculePrixTTC(21.30, 2);
printf("Sous-total TTC : %.2f euros\n", total_ttc);

// achat d'1 article à 2,40 euros HT
total_ttc += calculePrixTTC(2.40, 1);
printf("Sous-total TTC : %.2f euros\n", total_ttc);

// fin des achats
afficherCaracteres('*', 40);

// total des achats TTC
afficherTotalTTC(total_ttc);
}
```

➡ Le programme est maintenant plus simple à lire et à maintenir !

## Surcharge de fonctions

Il est interdit en C de définir plusieurs fonctions qui portent le même nom. Mais c'est autorisé en C++ si le compilateur peut différencier deux fonctions en analysant les paramètres qu'elle reçoit.

La **surcharge** (*overloading*) est une technique permettant de définir plusieurs fonctions qui portent le même nom si celles-ci ont des **signatures différentes**. La signature d'une fonction correspond au prototype **sans le type de retour**. Une surcharge de fonction est donc possible si les paramètres (leurs nombre et/ou leur type) sont différents :

```
// Retourne le minimum entre 2 entiers
int min(int val1, int val2)
{
    if(val1 <= val2)
        return val1;
    return val2;
}

// Surcharge : les types des paramètres sont différents (2 flottants)
float min(float val1, float val2)
{
    if(val1 <= val2)
        return val1;
    return val2;
}

// Surcharge : le nombre de paramètres est différent (3 entiers)
int min(int val1, int val2, int val3)
{
    int val = min(val1, val2);
    val = min(val, val3);
    return val;
}
```

*Surcharge de fonctions (C++)*

⚠ La surcharge n'est pas possible en langage C. Elle est utilisable seulement en C++.

## Paramètre par défaut

Le C++, mais pas le C, offre la possibilité d'affecter des **valeurs par défaut** aux paramètres d'une fonction. La syntaxe de la liste de paramètres sera alors la suivante :

type variable [= valeur] [, type variable [= valeur] [...]]

où **type** est le type du paramètre variable qui le suit et **valeur** sa valeur par défaut. La valeur par défaut d'un paramètre est la valeur que ce paramètre prend si aucune valeur ne lui est attribuée lors de l'appel de la fonction.

```
// Retourne la somme entre 2 entiers
int somme(int val1=0, int val2=0)
{
    return val1 + val2;
}

int main()
{
    int a = 5, b = 10;

    printf("%d\n", somme(a, b)); // affiche 15 car 10 + 5
    printf("%d\n", somme(b, a)); // affiche 15 car 5 + 10

    printf("%d\n", somme(a)); // affiche 5 car 5 + 0
    printf("%d\n", somme(b)); // affiche 10 car 10 + 0

    printf("%d\n", somme()); // affiche 0 car 0 + 0

    return 0;
}
```

*Paramètre par défaut (C++)*

⚠ Les paramètres par défaut doivent être déclarés successivement de la droite vers la gauche. La fonction « `int somme(int val1=0, int val2) {...}` » est invalide, car si on ne passe pas deux paramètres, `val2` ne sera pas initialisé. Les paramètres par défaut n'existent pas en langage C, seulement en C++.

## Fonctions inline

Le C++ dispose du mot clé **inline**, qui permet de modifier la méthode d'implémentation des fonctions. Placé devant la déclaration d'une fonction, il propose au compilateur de ne pas instancier cette fonction (notion de **macro**). Cela signifie que l'on désire que le compilateur remplace l'appel de la fonction par le code correspondant.

☞ Si la fonction est “grosse” ou si elle est appelée souvent, le programme devient **plus volumineux**, puisque la fonction est réécrite à chaque fois qu'elle est appelée. En revanche, il devient nettement **plus rapide**, puisque les mécanismes d'appel de fonctions, de passage des paramètres et de la valeur de retour sont ainsi évités.

🔗 *En pratique, on réservera cette technique pour les “petites” fonctions appelées dans du code devant être rapide. Cela peut s'apparenter à de l'optimisation.*

## Fonctions statiques

Par défaut, lorsqu'une fonction est définie dans un fichier C/C++, elle peut être utilisée dans tout autre fichier pourvu qu'elle soit déclarée avant son utilisation. Dans ce cas, la fonction est dite **externe** (mot-clé `extern`).

Il peut cependant être intéressant de définir des fonctions locales à un fichier, soit afin de résoudre des conflits de noms (entre deux fonctions de même nom et de même signature mais dans deux fichiers différents), soit parce que la fonction est uniquement d'intérêt local. Le C/C++ fournit donc le mot clé `static` qui, une fois placé devant la définition et les éventuelles déclarations d'une fonction, la rend unique et utilisable uniquement dans ce fichier. À part ce détail, les fonctions statiques s'utilisent exactement comme des fonctions classiques.

➡ Cas des variables :

Le mot clé `static`, placé devant le nom d'une **variable globale** (une variable déclarée en dehors de tout bloc `{}`), a le même effet que pour les fonctions statiques. On parle alors de variable globale cachée.

🕒 Le mot clé `static`, placé devant le nom d'une **variable locale** (une variable déclarée dans un bloc `{}`), rend la variable persistante (la variable conserve sa valeur) entre chaque entrée dans le bloc.

```
#include <stdio.h>

void compte()
{
    static int compteur = 0; // la variable compteur conservera sa valeur entre chaque appel

    ++compteur;
    printf("La fonction a été appelée : %d fois\n", compteur);
}

int main()
{
    compte();
    compte();
    compte();

    return 0;
}
```

*Variable locale statique*

On obtient :

```
$ ./a.out
La fonction a été appelée : 1 fois
La fonction a été appelée : 2 fois
La fonction a été appelée : 3 fois
```

## Nombre variable de paramètres

En général, les fonctions ont un nombre constant de paramètres. Pour les fonctions qui ont des paramètres par défaut en C++, le nombre de paramètres peut apparaître variable à l'appel de la fonction, mais en réalité, la fonction utilise toujours le même nombre de paramètres.

Le C et le C++ disposent toutefois d'un mécanisme qui permet au programmeur de réaliser des fonctions dont le nombre et le type des paramètres sont variables. C'est le cas par exemple des fonctions `printf()` et `scanf()`.

Pour indiquer au compilateur qu'une fonction peut accepter une liste de paramètres variable, il faut simplement utiliser des points de suspensions dans la liste des paramètres dans les déclarations et la définition de la fonction :

`type identificateur(paramètres, ...)`

Dans tous les cas, il est nécessaire que la fonction ait au moins un paramètre classique. Les paramètres classiques doivent impérativement être avant les points de suspensions.

On utilisera le type `va_list` et les expressions `va_start`, `va_arg` et `va_end` pour récupérer les arguments de la liste de paramètres variable, un à un. Pour cela, il faudra inclure le fichier d'en-tête `stdarg.h`.

```
#include <stdarg.h>

/* effectue la somme de compte flottants (float ou double) et la renvoie dans un double */
double somme(int compte, ...)
{
    double resultat = 0; /* Variable stockant la somme */
    va_list varg; /* Variable identifiant le prochain paramètre */

    va_start(varg, compte); /* Initialisation de la liste */

    /* Parcours de la liste */
    while (compte != 0)
    {
        resultat = resultat + va_arg(varg, double); /* récupère le prochain paramètre dans
            la liste */
        compte = compte-1;
    }

    va_end(varg);

    return resultat;
}

int main()
{
    double total = 0;

    total = somme(3, 1.5, 2.5, 5.); // 4 paramètres
    printf("%.1f\n", total); // affiche 9.0

    total = somme(2, 1.5, 2.5); // et maintenant 3 paramètres
    printf("%.1f\n", total); // affiche 4.0

    return 0;
}
```

*Nombre variable de paramètres*

## Pointeur vers une fonction

Le **nom** d'une fonction est **une constante de type pointeur** :

```
// une fonction
int f(int x, int y)
{
    return x+y;
}

// un pointeur sur une fonction :
int (*pf)(int, int); // pf est un pointeur vers une fonction admettant 2 entiers en
    paramètres et retournant un entier

pf = f;           // pf pointe vers la fonction f

// appel :
printf("%d\n", (*pf)(3, 5)); // affiche 8
```

*Pointeur sur une fonction*

## Questions de révision

L'idée de base des questions de révision est de vous donner une chance de voir si vous avez identifié et compris les points clés de cette partie.

**Question 31.** Quelles sont les quatre parties d'une fonction ?

**Question 32.** Comment s'appelle une fonction qui ne retourne aucun résultat ?

**Question 33.** Citer les trois types de passage de paramètres ?

**Question 34.** Donner la déclaration d'une fonction `foo` qui reçoit en paramètres un tableau d'entiers et le nombre d'éléments qu'il contient et qui retourne la somme de ces éléments.

**Question 35.** Citer quatre améliorations apportées aux fonctions en C++ ?

## Conclusion

Il est indispensable de décomposer un traitement de grande taille en plusieurs parties plus petites jusqu'à obtenir quelque chose de suffisamment simple à comprendre et à résoudre. Cette approche consistant à décomposer une tâche complexe en une suite d'étapes plus petites (et donc plus facile à gérer) ne s'applique pas uniquement à la programmation et aux ordinateurs. Elle est courante et utile dans la plupart des domaines de l'existence.

**Descartes** (mathématicien, physicien et philosophe français) dans le *Discours de la méthode* :

« diviser chacune des difficultés que j'examinerais, en autant de parcelles qu'il se pourrait, et qu'il serait requis pour les mieux résoudre. »

## Types composés

Les types composés permettent de regrouper des variables au sein d'une même entité :

- Il est possible de regrouper des variables de types différents dans des **structures de données** ;
- Il est possible de regrouper des variables de types identiques dans des **tableaux**.

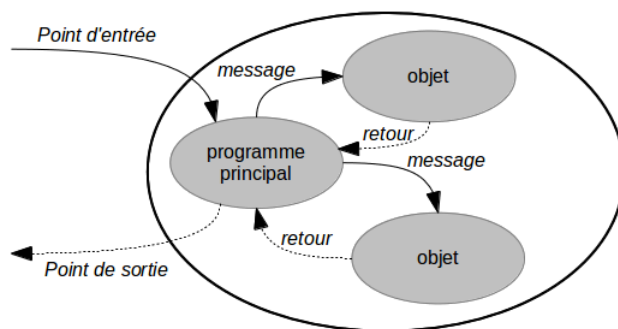
✎ Une chaîne de caractères peut être considérée comme un type composé.

En C, on dispose de trois types de structures de données :

- les structures (**struct**)
- les unions (**union**)
- les champs de bits

Le C++ ajoute la notion de type **classe** qui permet de réaliser des programmes orientés objet (POO). La classe est le modèle (le « moule ») pour créer des objets logiciels.

✎ En POO, un programme est vu comme un ensemble d'objets interagissant entre eux en s'échangeant des messages.



## Structures

⇒ *Besoin* : Il est habituel en programmation d'avoir besoin d'un mécanisme permettant de grouper un certain nombre de variables de types différents au sein d'une même entité.

Exemple : On travaille par exemple sur un fichier de personnes et on voudrait regrouper une variable de type chaîne de caractères pour le nom, une variable de type entier pour le numéro d'employé, etc.

La réponse à ce besoin est le concept d'**enregistrement** : un enregistrement est un ensemble d'éléments de types différents repérés par un nom. Les éléments d'un enregistrement sont appelés des **champs**.

En langage C, on utilise le vocabulaire suivant :

- enregistrement → **structure**
- champ d'un enregistrement → **membre** d'une structure

Une **structure** est donc un **objet agrégé comprenant un ou plusieurs membres** d'éventuellement différents types que l'on regroupe sous un seul nom afin d'en faciliter la manipulation et le traitement.

Chacun des membres peut avoir n'importe quel type, y compris une structure, à l'exception de celle à laquelle il appartient.

## Déclaration de structure

Il y a plusieurs méthodes possibles pour déclarer des structures.

```
struct [etiquette]
{
    type champ_1;
    ...
    type champ_n;
} [identificateur];
```

Pour déclarer une structure, on utilise le mot clé **struct** suivi d'une liste de déclarations entre accolades. Il est possible de faire suivre le mot **struct** d'un nom baptisé etiquette de la structure. Cette etiquette désigne alors cette sorte de structure et, par la suite, peut servir pour éviter d'écrire entièrement toute la déclaration. Il est aussi possible d'instancier directement une variable de nom **identificateur**.

```
// Déclaration d'une type struct Date :
struct Date
{
    int    jour;
    int    mois;
    int    annee;
};

// Instanciation d'une variable de type struct Date :
struct Date dateNaissance;

// Ou directement :
struct
{
    int    jour;
    int    mois;
    int    annee;
} dateDeces; // dateDeces est une variable de type structuré
```

*Exemple de déclaration d'un type structuré*

En C, le type s'appelle en fait **struct Date**. On préfère souvent créer un **synonyme** avec **typedef** :

```
// Déclaration d'une type struct Date :
struct Date
{
    int    jour,
           mois,
           annee;
};

// Création d'un type synonyme :
typedef struct Date Date;

// Ou directement :
typedef struct
{
    int    jour,
           mois,
           annee;
```



```
} Date;  
  
// Une variable de type Date :  
Date dateNaissance;
```

*Un synonyme de type structuré*

✍ En C++, ce synonyme est créé naturellement avec la structure. Le **typedef** est donc inutile dans ce langage.

## Initialisation d'une structure

Comme pour les tableaux, les accolades peuvent être utilisées pour indiquer les valeurs initiales des membres d'une variable de type structuré. Cela ne fonctionne qu'à l'initialisation.

☹ Ce n'est pas utilisable pour une affectation.

```
// Une variable de type Date :  
Date naissanceMarie = {7, 11, 1867}; // initialisation de la structure naissanceMarie  
  
printf("La structure struct Date occupe une taille de %d octets\n", sizeof(struct Date));
```

*Initialisation d'une structure*

✍ La taille d'une structure est la somme des tailles de tous les objets qui la compose (cf. `sizeof()`). Dans notre exemple, la structure aura une taille de  $3 \times 4$  (`int`) soit 12 octets.

## Accès aux membres

Pour accéder aux membres d'une structure, on utilise :

– l'opérateur d'accès `.` (point) pour une variable de type structuré :

```
Date naissanceMarie = {7, 11, 1867}; // initialisation de la structure naissanceMarie  
  
printf("Marie Curie est née le %02d/%02d/%4d\n", naissanceMarie.jour, naissanceMarie.mois,  
naissanceMarie.annee);
```

– l'opérateur d'indirection `->` (flèche) pour un pointeur sur un type structuré :

```
struct Date naissanceMarie = {7, 11, 1867}; // initialisation de la structure  
naissanceMarie  
struct Date *p_naissanceMarie = &naissanceMarie;  
  
printf("Marie Curie est née le %02d/%02d/%4d\n", p_naissanceMarie->jour, p_naissanceMarie  
->mois, p_naissanceMarie->annee);  
  
// Ou :  
printf("Marie Curie est née le %02d/%02d/%4d\n", (*p_naissanceMarie).jour->jour, (*  
p_naissanceMarie).mois, (*p_naissanceMarie).annee);
```

## Affectation de structure

Il suffit d'accéder aux membres d'une structure pour leurs affecter une valeur.

```
struct Date naissanceMarie = {7, 11, 1867}; // initialisation de la structure naissanceMarie
struct Date uneDate;

// Quelques affectations :
uneDate.jour = 1;
scanf("%d", &uneDate.mois);
uneDate.jour = naissanceMarie.annee;
```

Il est aussi possible d'affecter des structures de même type entre elles :

```
struct Date naissanceMarie = {7, 11, 1867};
struct Date copie;

copie = uneDate;
```

Dans ce cas, les valeurs sont copiées.

Ainsi, lorsque vous prenez une structure en paramètre d'une fonction, la valeur donnée à l'appel de la fonction est copiée dans l'emplacement mémoire du paramètre. Le paramètre a ainsi son propre emplacement mémoire. Si la structure est très grosse, appeler la fonction va donc nécessiter un espace supplémentaire, et une copie qui peut prendre du temps. Pour éviter cela, on pourra utiliser un **passage par adresse** en C/C++ ou **par référence** en C++.

⊕ Aucune comparaison n'est possible sur les structures, même pas les opérateurs == et !=.

## Tableaux de structures

Une déclaration de tableau de structures se fait selon le même modèle que la déclaration d'un tableau dont les éléments sont de type simple :

```
struct Date t[100]; // 100 éléments de type struct Date

// Ou :
Date t[100]; // 100 éléments de type Date

// Chaque élément du tableau est de type Date :
for(i=0<100;i++)
    printf("%02d/%02d/%4d\n", t[i].jour, t[i].mois, t[i].annee);
```

## Liste de structures

Une des utilisations fréquentes des structures, est de créer des listes de structures chaînées. Pour cela, il faut que chaque structure contienne un membre qui soit de type **pointeur** vers une structure du même type. Cela se fait de la façon suivante :

```
struct personne
{
    ... /* les différents membres */

    struct personne *suivant;
};
```

⚡ La dernière structure de la liste devra avoir un membre suivant dont la valeur sera le pointeur **NULL** pour indiquer la fin.

Quand on crée une liste chaînée, on ne sait généralement pas à la compilation combien elle comportera d'éléments à l'exécution. Pour pouvoir créer des listes, il est donc nécessaire de pouvoir allouer de l'espace dynamiquement :

```
#include <stdlib.h>

struct personne *p;

// Allocation :
p = malloc(sizeof(struct personne));

// Libération :
free(p);
```

*Allocation dynamique d'une structure*

## Union

➡ *Besoin : Il est parfois nécessaire de manipuler des variables auxquelles on désire affecter des valeurs de type différents.*

Une **union** est conceptuellement identique à une structure mais peut, à tout moment, contenir n'importe lequel des différents champs.

Une union définit en fait **plusieurs manières de regarder le même emplacement mémoire**. A l'exception de ceci, la façon dont sont déclarés et référencés les structures et les unions est identique.

```
union [etiquette] {
    type champ_1;
    ...
    type champ_n;
} [identificateur];
```

Exemple : On va déclarer une **union** pour conserver la valeur d'une mesure issue d'un capteur générique, qui peut par exemple fournir une mesure sous forme d'un **char** (-128 à +127), d'un **int** (-2147483648 à 2147483647) ou d'un **float**. **valeurCapteur** pourra, à tout moment, contenir SOIT un entier, SOIT un réel, SOIT un caractère.

```
union mesureCapteur
{
    int    iVal;
    float  fVal;
    char   cVal;
} valeurCapteur;
```

⚡ *La taille mémoire de la variable **valeurCapteur** est égale à la taille mémoire du plus grand type qu'elle contient (ici c'est **float**).*

```
#include <stdio.h>
typedef union mesureCapteur
{
    int    iVal;
    float  fVal;
    char   cVal;
} Capteur;
```

```
int main()
{
    Capteur vitesseVent, temperatureMoteur, pressionAtmospherique;
    pressionAtmospherique.iVal = 1013; /* un int */
    temperatureMoteur.fVal = 50.5; /* un float */
    vitesseVent.cVal = 2; /* un char */

    printf("La pression atmosphérique est de %d hPa\n", pressionAtmospherique.iVal);
    printf("La température du moteur est de %.1f °C\n", temperatureMoteur.fVal);
    printf("La vitesse du vent est de %d km/h\n", vitesseVent.cVal);

    printf("Le type Capteur occupe une taille de %d octets\n", sizeof(Capteur));

    return 0;
}
```

*Utilisation d'une **union***

L'exécution du programme d'essai permet de vérifier cela :

```
La pression atmosphérique est de 1013 hPa
La température du moteur est de 50.5 °C
La vitesse du vent est de 2 km/h
Le type Capteur occupe une taille de 4 octets
```

## Champs de bits

➡ *Besoin : Il est parfois nécessaire pour un programmeur de décrire en termes de bits la structure d'une information.*

Les **champs** de bits ("Drapeaux" ou "*Flags*"), qui ont leur principale application en informatique industrielle, sont des **structures** qui ont la possibilité de regrouper (au plus juste) plusieurs valeurs. La taille d'un champ de bits **ne doit pas excéder celle d'un entier**. Pour aller au-delà, on créera un deuxième champ de bits.

On utilisera le mot clé **struct** et on donnera le type des groupes de bits, leurs noms, et enfin leurs tailles :

```
struct [etiquette]
{
    type champ_1 : nombre_de_bits;
    type champ_2 : nombre_de_bits;
    [...]
    type champ_n : nombre_de_bits;
} [identificateur];
```

Si on reprend le type structuré `Date`, on peut maintenant décomposer ce type en trois groupes de bits (jour, mois et année) avec le nombre de bits suffisants pour coder chaque champ. Les différents groupes de bits seront tous accessibles comme des variables classiques d'une structure ou d'une union.

```
struct Date
{
    unsigned short jour : 5; // 2^5 = 0-32 soit de 1 à 31
    unsigned short mois : 4; // 2^4 = 0-16 soit de 1 à 12
    unsigned short annee : 7; // 2^7 = 0-128 soit de 0 à 99 (sans les siècles)
};
```

```
int main (int argc, char **argv)
{
    struct Date naissanceRitchie = {9, 9, 41};
    struct Date naissanceThompson = {4, 2, 43};
    struct Date mortRitchie = {12, 10, 11};

    printf("Dennis Ritchie est né le %02d/%02d/%2d\n", naissanceRitchie.jour,
        naissanceRitchie.mois, naissanceRitchie.annee);
    printf("Dennis Ritchie est mort le %02d/%02d/%2d\n\n", mortRitchie.jour, mortRitchie.mois
        , mortRitchie.annee);
    printf("Ken Thompson est né le %02d/%02d/%2d\n", naissanceThompson.jour,
        naissanceThompson.mois, naissanceThompson.annee);

    printf("La structure champs de bits date occupe une taille de %d octets\n", sizeof(struct
        date));
    return 0;
}
```

*Le champ de bits Date*

✎ Il est autorisé de ne pas donner de nom aux champs de bits qui ne sont pas utilisés.

L'exécution du programme d'essai permet de vérifier cela :

```
Dennis Ritchie est né le 09/09/41
Dennis Ritchie est né le 09/09/41
Dennis Ritchie est mort le 12/10/11
Ken Thompson est né le 04/02/43

La structure champs de bits date occupe une taille de 2 octets
```

✎ La taille mémoire d'une variable de ce type sera égale à 2 octets ( $5 + 4 + 7 = 16$  bits).

## Questions de révision

L'idée de base des questions de révision est de vous donner une chance de voir si vous avez identifié et compris les points clés de cette partie.

**Question 36.** Donner la déclaration d'une structure **Voiture** qui possède les champs suivants : la capacité du réservoir et le nombre de chevaux fiscaux ?

**Question 37.** Quelle est la taille en octets d'une structure qui contient 2 champs : un entier (**int**) et un flottant (**float**) ?

**Question 38.** Quelle est la taille en octets d'une union qui contient 2 champs : un entier (**int**) et un flottant (**float**) ?

**Question 39.** Donner la déclaration d'un type énuméré **CouleurCarte** pour les quatres couleurs de cartes à jouer ?

**Question 40.** Peut-on réaliser des tableaux de structures ? Peut-on réaliser des structures qui contiennent des tableaux ?

## Conclusion

Les types sont au centre de la plupart des notions de programmes corrects, et certaines des techniques de construction de programmes les plus efficaces reposent sur la conception et l'utilisation des types.

**Rob Pike** (ancien chercheur des *Laboratoires Bell* et maintenant ingénieur chez *Google*) :

« Règle n°5 : Les données prévalent sur le code. Si vous avez conçu la structure des données appropriée et bien organisé le tout, les algorithmes viendront d'eux-mêmes. La structure des données est le coeur de la programmation, et non pas les algorithmes. »

Cette règle n°5 est souvent résumée par « Écrivez du code stupide qui utilise des données futées ! »

## Annexe 1 : environnement de développement

Si vous souhaitez développer sur un autre système d'exploitation que GNU/Linux, il vous faudra installer sur votre système une **chaîne de développement** comprenant au minimum un éditeur de texte et un compilateur.

### Quelques solutions :

⇒ Éditeur de texte GUI<sup>1</sup> (libre et gratuit) :

- Komodo Edit : <https://www.activestate.com/komodo-ide/downloads/edit>
- Notepad++ : <https://notepad-plus-plus.org/download/>

☞ voir aussi *Framapack* qui est un outil qui vous permet d'installer une collection de logiciels libres pour ©Windows de votre choix en une seule fois.

⇒ Compilateur (libre et gratuit) pour ©Windows :

- MinGW : <http://www.mingw.org/>
- Cygwin : <http://www.cygwin.com/>

⇒ EDI<sup>2</sup> (IDE<sup>3</sup>) :

- Code::Blocks : <http://www.codeblocks.org/>
- Netbeans : <https://netbeans.org/downloads/>
- Eclipse : <https://www.eclipse.org/downloads/>
- Dev-C++ ©Windows : <http://orwelldevcpp.blogspot.fr/>
- ...

☞ L'ensemble de ces logiciels existent aussi sous GNU/Linux à l'exception de Dev-C++ et évidemment de MinGW/Cygwin.

Si vous voulez disposer d'un environnement de développement GNU/Linux, il vous faut :

- utiliser une distribution “live” (DVD ou clé USB)
- installer GNU/Linux sur votre disque dur (en double *boot* éventuellement)
- installer GNU/Linux dans une machine virtuelle (avec VirtualBox par exemple)
- installer Cygwin sur votre machine ©Windows

Si vous voulez programmer sans installer d'environnement de développement sur votre machine, vous pouvez utiliser des “compilateurs” en ligne (cf. CodingGround) :

- C : [https://www.tutorialspoint.com/compile\\_c\\_online.php](https://www.tutorialspoint.com/compile_c_online.php)

---

1. GUI : *Graphical User Interface*

2. EDI : Environnement de Développement Intégré

3. IDE : *Integrated Development Environment*

- C99 : [https://www.tutorialspoint.com/compile\\_c99\\_online.php](https://www.tutorialspoint.com/compile_c99_online.php)
- C++ : [https://www.tutorialspoint.com/compile\\_cpp\\_online.php](https://www.tutorialspoint.com/compile_cpp_online.php)
- C++ 11 : [https://www.tutorialspoint.com/compile\\_cpp11\\_online.php](https://www.tutorialspoint.com/compile_cpp11_online.php)



## Annexe 2 : éditer un fichier texte avec vim

**vi** est l'éditeur de texte standard d'Unix et il a été l'éditeur favori de nombreux *hackers* jusqu'à l'arrivée d'Emacs en 1984. Tout système se conformant aux spécifications Unix intègre **vi** et il est donc encore largement utilisé par les utilisateurs (surtout les administrateurs et programmeurs) des différentes variantes d'Unix.

La version incluse actuellement dans les **Linux** est le plus souvent **vim** (*vi improved*), un clone de **vi** qui comporte quelques différences avec celui-ci. **vi/vim** comprend trois modes de fonctionnement : le mode normal, le mode **commande** et le mode **insertion**. Après le lancement de **vi/vim**, c'est le mode normal qui est actif. Pour passer en mode insertion (de texte évidemment) il faut appuyer sur la touche ou . On sait que l'on est en mode insertion par l'affichage de **INSERT** en bas de la fenêtre. Pour sortir de ce mode, il faut appuyer sur la touche et l'affichage de **INSERT** en bas de la fenêtre disparaît. Pour passer en mode commande, il faut taper **':'**.

Quelques commandes intéressantes :

```
:q! : sortie sans sauvegarde
:wq : sortie avec sauvegarde
:x : sortie avec sauvegarde
ZZ : sortie avec sauvegarde
:w : sauvegarde sans sortie
$ : se déplacer sur le dernier caractère de la ligne
Ctrl f : afficher la page suivante
Ctrl b : afficher la page précédente
Ctrl d : afficher la demi-page suivante
Ctrl u : afficher la demi-page précédente
e : se déplacer à la fin du mot
b : se déplacer au début du mot
w : se déplacer au début du mot suivant
H : se déplacer en haut de l'écran
L : se déplacer en bas de l'écran
M : se déplacer au milieu de l'écran
z. : décaler l'affichage avec la ligne courante au centre
z (return) : décaler l'affichage avec la ligne courante en haut
z- : décaler l'affichage pour que la ligne courante
:num_ligne : se déplacer à la ligne num_ligne
G (ou :$) : aller à la fin du fichier
```

u : annulation de la dernière modification  
dd : suppression de la ligne courante  
2dd : suppression des deux lignes suivantes  
D : suppression de la fin de la ligne à partir du curseur  
:3,7 d : suppression des lignes 3 à 7  
:3,7 t 10 : copie des lignes 3 à 7 après la ligne 10  
:3,7 m 10 : transfert des lignes 3 à 7 après la ligne 10  
yy : mémorisation de la ligne courante (copier)  
3yy : mémorisation des 3 lignes suivantes (copier)  
p : copie ce qui a été mémorisé après le curseur  
P : copie ce qui a été mémorisé avant le curseur  
:set nu : affichage des numéros de ligne  
/mot : recherche le mot mot (on se déplace avec n ou N ou \*)

▣ Il existe en réalité une quantité astronomiques de commandes dans *vi*, et en particulier dans *vim*, et chaque personne utilise, en général, qu'une petite partie d'entre elles en fonction de ses habitudes (et souvent, pas les mêmes que vous...).

## Annexe 3 : Une liste succincte de commandes de base

Voici quelques commandes usuels :

dpkg : un gestionnaire de paquet pour Debian  
apt-get : utilitaire APT pour la gestion des paquets (voir aussi aptitude)  
alias : crée ou supprime des alias de commandes  
pwd : affiche le chemin d'accès au répertoire courant  
man : permet de consulter les manuels de référence  
clear : efface l'écran  
echo : affiche une ligne de texte (et aussi des variables)  
cd : permet de se déplacer dans une arborescence  
ls : liste le contenu d'un répertoire  
rm : supprime un fichier (voir aussi rmdir)  
cp : permet la copie de fichier (voir aussi cp -a)  
mv : déplace ou renomme une partie d'une arborescence  
mkdir : crée un répertoire dans une arborescence  
touch : modifie l'horodatage d'un fichier (permet aussi de créer un fichier vide)  
file : affiche le type des fichiers  
type : indique le type pour une commande  
locate : localise un fichier  
find : recherche des fichiers sur le système  
cat : affiche et/ou concatène le(s) fichier(s) sur la sortie standard  
more : affiche à l'écran l'entrée standard (page par page)  
less : idem avec possibilité de retour en arrière  
cut : permet d'isoler des colonnes dans un fichier  
head : affiche les n premières lignes  
join : joint les lignes de deux fichiers en fonction d'un champ commun  
sort : trie les lignes de texte en entrée  
paste : concatène les lignes des fichiers  
tail : affiche les n dernières lignes d'un fichier  
tac : concatène les fichiers en inversant l'ordre des lignes  
uniq : élimine les doublons d'un fichier trié  
rev : inverse l'ordre des lignes d'un fichier  
diff : compare des fichiers texte



`cmp` : compare deux fichiers octet par octet  
`tr` : remplace ou efface des caractères  
`grep` : recherche des chaînes de caractères dans des fichiers  
`sed` : éditeur de flux pour le filtrage et la transformation de texte  
`awk` : manipulation avancée de fichiers texte  
`md5sum` : génère et vérifie un hachage MD5  
`whereis` : permet de trouver l'emplacement d'une commande  
`whatis` : donne une description d'une commande  
`which` : donne le chemin complet d'une commande  
`du` : affiche une arborescence et sa taille (`du -h`)  
`df` : fournit la quantité d'espace occupé par les systèmes de fichiers (`df -Th`)  
`od` : affiche le dump d'un fichier (voir aussi `hexdump`)  
`wc` : compte les caractères, les mots et les lignes en entrée  
`date` : affiche et modifie la date et l'heure  
`cal` : affiche le calendrier  
`bc` : calculatrice  
`ln` : crée des liens physiques et symboliques  
`lsattr` : liste les attributs des fichiers  
`chattr` : change les attributs des fichiers  
`stat` : affiche des informations sur un fichier ou un système de fichier  
`lsuf` : affiche des informations sur les fichiers ouverts  
`fuser` : identifie les processus utilisant des fichiers  
`fdisk` : gère les tables de partitions pour Linux  
`cfdisk` : manipule les tables de partitions pour Linux (voir aussi `sfdisk`)  
`dd` : convertit et copie un fichier physiquement  
`sync` : vider les tampons du système de fichiers (finalise les opérations d'écriture)  
`id` : affiche les identifiants d'utilisateur et de groupe effectifs et réels  
`whoami` : affiche l'identifiant d'utilisateur  
`who` : montre qui est connecté (voir aussi `w` et `users`)  
`last` : affiche une liste des utilisateurs dernièrement connectés  
`su` : change l'identifiant d'utilisateur ou permet de devenir un superutilisateur (`root`)  
`sudo` : exécute une commande sous un autre compte (voir `/etc/sudoers`)  
`uname` : affiche des informations sur le système  
`ps` : affiche les processus en cours  
`top` : affiche les tâches  
`kill` : envoie un signal à un processus  
`time` : exécute un programme et affiche un résumé des ressources utilisées  
`uptime` : indique depuis quand le système a été mis en route  
`free` : affiche les quantités de mémoire libre et utilisée du système  
`printenv` : affiche l'ensemble ou une partie des variables d'environnement

## Annexe 4 : `printf()` et `scanf()`

Il existe trois flux de base pour chaque programme :

- `stdin` (0) : Entrée standard (par défaut le clavier)
- `stdout` (1) : Sortie standard (par défaut l'écran)
- `stderr` (2) : Sortie erreur (par défaut l'écran également)

✎ En C, un flux est un canal destiné à transmettre ou à recevoir de l'information. Il peut s'agir de fichier ou de périphériques.

Pour afficher des résultats à l'écran, il faudra écrire sur le flux `stdout`. Pour cela, on utilise généralement la fonction `printf()` (déclarée dans `stdio.h`) :

```
#include <stdio.h>

printf("Hello World!\n");

// ce qui correspond à :
fprintf(stdout, "Hello World!\n");

// ou :
write(1, "Hello World!\n"); // 1 étant le descripteur de stdout
```

Pour afficher une erreur, il est recommandé de la faire dans le flux `stderr` :

```
// En C/C++
fprintf(stderr, "Ceci est une erreur.\n");

// En C++
cerr << "Ceci est une erreur.\n";
```

✎ Si on ne désire pas l'affichage des messages d'erreur d'un programme, il suffit de rediriger son flux `stderr` vers le périphérique `null`. Par exemple pour le programme `a.out` :

```
$ ./a.out 2> /dev/null
```

La fonction `printf()` (ainsi que `sprintf()` et `fprintf()`) prend en argument une chaîne de format qui spécifie les types de ses autres arguments. La chaîne de format de `printf()` est riche en possibilités, car elle indique non seulement le type des arguments mais aussi la manière de les écrire. Par exemple, vous pouvez choisir le nombre de chiffres après la virgule d'un nombre flottant. Chaque code spécial de la chaîne de format est précédé d'un `%`. Si vous voulez écrire le symbole `%`, vous devez l'écrire deux fois : `%%`.

```
printf("%d\n", 42); // affichage d'un entier
printf("Taux de réussite : %.1f %%\n", 92.9); // Taux de réussite : 92.9 %
```

Chaque code spécial est découpé en plusieurs sections (les sections entre crochets sont facultatives) :

`%[largeur][.precision]type`

1. Largeur : nombre minimum de caractères à écrire, `printf()` complètera si nécessaire par des espaces sur la gauche, la valeur est alignée à droite.
2. Précision : nombre de chiffres après la virgule à écrire pour les nombres flottants, précédé d'un point. La valeur est automatiquement arrondie : le dernier chiffre est incrémenté s'il y a besoin.
3. Type : le type de la valeur à écrire

Voici les différents types utilisables :

<code>%d</code>	<code>int</code>	entier 32 bits en décimal
<code>%u</code>	<code>unsigned int</code>	entier 32 bits non signé en décimal
<code>%x</code>	<code>int</code>	entier 32 bits en hexadécimal (en minuscules et <code>%X</code> en majuscules)
<code>%lld</code>	<code>long long int</code>	entier 64 bits
<code>%f</code>	<code>float</code> ou <code>double</code>	avec par défaut 6 chiffres après la virgule
<code>%g</code>	<code>float</code> ou <code>double</code>	en utilisant la notation <code>1.234e5</code> quand cela est approprié
<code>%c</code>	<code>char</code>	caractère unique
<code>%s</code>	<code>char*</code>	chaîne de caractères
<code>%p</code>	<code>void*</code>	l'adresse mémoire en hexadécimal

☞ *man 3 printf*








La fonction `scanf()` permet de lire des données sur l'entrée standard (`stdin`). Cette fonction prend comme premier argument une chaîne de format, qui indique combien de variables vous souhaitez lire, et de quel type. Ensuite vous donnez passer en arguments des pointeurs sur les variables qui recevront le résultat, dans l'ordre de la chaîne de format.

Prenons le cas le plus simple :

```
#include <stdio.h>

// lire un entier et le placer dans la variable n :
int n;
scanf("%d", &n); // ne pas oubliez le & qui donne l'adresse de la variable n, c'est
                 nécessaire ici pour que la fonction puisse modifier la variable passée en argument

// lire deux entiers :
int a, b;
scanf("%d %d", &a, &b);
```

✎ La lecture des chaînes de caractère (avec `scanf()` en C/C++ ou `cin` en C++) se termine sur ce qu'on appelle un espace blanc (*whitespace*), c'est-à-dire le caractère espace, une tabulation ou un caractère de retour à la ligne (généralement la touche ). Notez que les espaces sont ignorés par défaut. Sous Linux, vous pouvez indiquer la fin d'un flux (EOF) en combinant les touches  +  qui indiquera qu'il n'y a plus de saisie. Vous pouvez aussi stopper le programme avec  +  ou l'interrompre avec  + .

La chaîne de format se compose de codes spéciaux indiquant le type des valeurs à lire. Chaque code est précédé du symbole `%`. Il en existe une multitude, nous nous contenterons ici de ceux dont vous aurez besoin :

<code>%d</code>	<code>int</code>	entier 32 bits en décimal
<code>%lld</code>	<code>long long int</code>	entier 64 bits
<code>%f</code>	<code>float</code>	nombre réel simple précision
<code>%lf</code>	<code>double</code>	nombre réel double précision
<code>%c</code>	<code>char</code>	caractère unique
<code>%s</code>	<code>char*</code>	chaîne de caractères

☞ *man 3 scanf*

## Annexe 5 : erreurs du débutant

⌘ *Le bug du débutant n°1* : il ne faut pas mettre un ; après un **if** car cela exécuterait une instruction nulle si expression est vraie. Cela donne un comportement défectueux :

```
int a = 0;
if(a != 0); /* c'est sûrement un bug involontaire */
    printf("bug : a n'est pas nul et pourtant a = %d !\n", a);
```

⌘ *Le bug du débutant n°2* : il ne faut pas confondre l'opérateur '=' (d'affectation) avec l'opérateur '==' (comparaison d'égalité). Cela risque de donner un comportement défectueux :

```
int a = 0;
if(a = 0) /* c'est sûrement un bug involontaire */
    printf("a = %d : a est égal à 0\n", a);
else printf("bug : a n'est pas égal à 0 et pourtant a = %d !\n", a);
```

⌘ *Le bug du débutant n°3* : il faut penser à déterminer si un **break** est nécessaire dans un **case**. Le code suivant le prouve :

```
a = 1;
switch (a)
{
    case 1: printf("a = %d : a est égal à 1\n", a); /* ne faudrait-il pas un break ? */
    case 2: printf("bug : a = %d : a est égal à 2 !\n", a);
}
```

Vous obtiendrez :

```
a = 1 : a est égal à 1
bug : a = 1 : a est égal à 2 !
```

⌘ *Le bug du débutant n°4* : il ne faut pas mettre un ; après le **while** car cela exécuterait une instruction nulle si expression est vraie. Le code suivant sera dans une **boucle infinie** sans afficher aucun message "Salut" :

```
long compteur = 15;
while (compteur > 0); // c'est sûrement un bug involontaire
{
    printf("Salut\n");
    compteur = compteur - 1;
} // ici non plus ne pas mettre de ;
```

⌘ *Le bug du débutant n°5* : il ne faut pas mettre un ; à la fin d'un **for** car cela exécuterait une instruction nulle si expression est vraie. Le code suivant n'affichera qu'un seul message "Salut" car le **for** boucle dans le vide :

```
long compteur;
for (compteur = 0; compteur < 15; compteur++); // c'est sûrement un bug involontaire
{
    printf("Salut\n");
}
```

⌘ *Le bug du débutant n°6* : il ne faut pas oublier les accolades {} à la suite d'une instruction conditionnelle **if/else** ou itérative **for/while** sinon vous n'exécuterez qu'une seule instruction au lieu de plusieurs. Par exemple :



**On ne peut donc pas tester l'égalité entre deux nombres réels.**

Pour déterminer si deux nombres réels sont égaux, il faut commencer par calculer la différence des deux nombres. Si la différence est très proche de zéro, c'est-à-dire si la valeur absolue de la différence est très petite, alors on considère que les nombres sont égaux.

Concrètement, pour comparer  $x$  et  $y$ , on commence par définir une constante très petite que l'on nomme traditionnellement *epsilon*, et on teste si  $|x - y| < \text{epsilon}$ . La fonction `fabs()` calcule la valeur absolue.

```
#include <math.h>

const double epsilon = 1e-10;

double x = 2.;
x = sqrt(x); // Racine carrée de x
x = x * x;    // x^2

if (fabs(x - 2.) < epsilon)
    printf("OK\n");

// pour tester si x est égal à zéro, il faut faire :
if (fabs(x) < epsilon)
    printf("x est égal à 0 !\n");
```