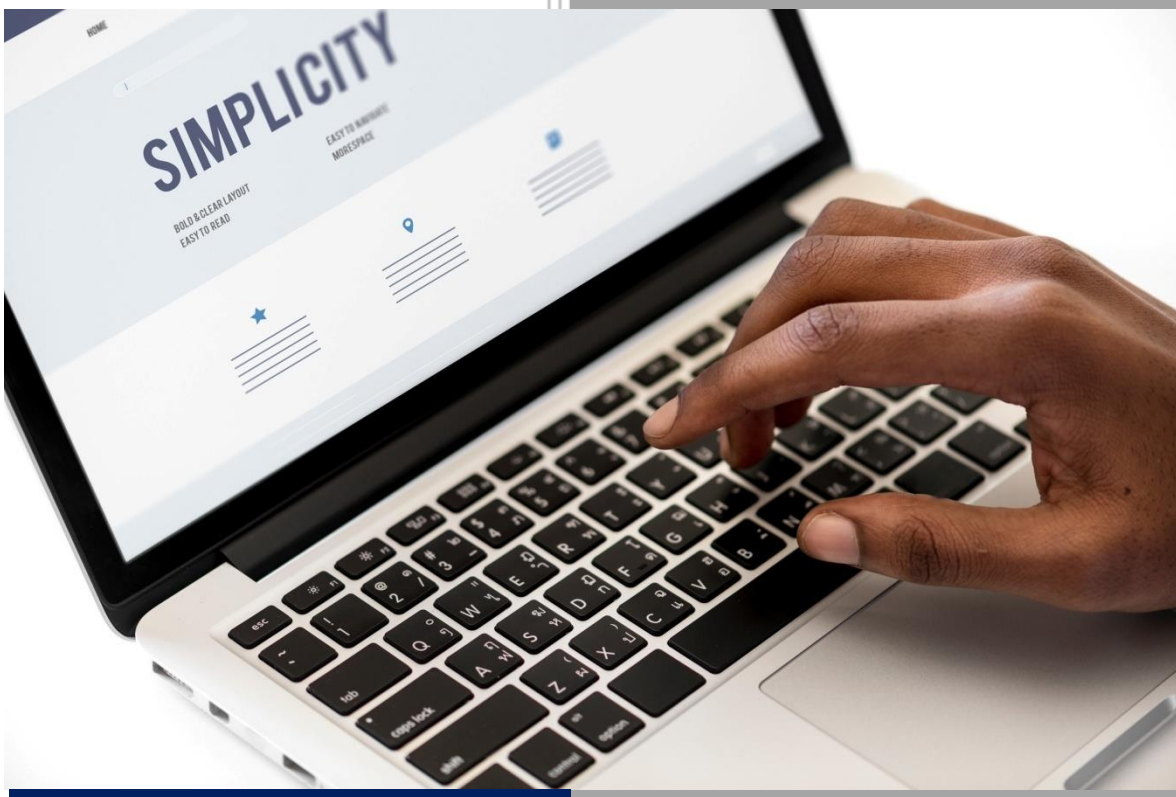
2023 / 2024

Programmation Structurée



CODE UE : IGL123
MODULE : Programmation I
CREDITS : 04
CIBLES : GL 1, GSI 1

"N'importe quel idiot peut écrire du code qu'un ordinateur peut comprendre. Les bons programmeurs écrivent du code que les humains peuvent comprendre." - **Martin Fowler**

M. TEJIOGNI

Manager des Systèmes d'Information et d'Infrastructure

Email : mtejiogni@yahoo.fr
Tél : 693 90 91 21 / 682 66 37 31

SOMMAIRE

OBJECTIF GENERAL

Le **C** est un langage incontournable qui en a inspiré beaucoup d'autres. Inventé dans les années 70, il est toujours d'actualité dans la programmation système et la robotique. Il est plutôt complexe, mais si vous le maîtrisez-vous aurez des bases de programmation très solides !

Dans ce cours, vous commencerez par découvrir le fonctionnement de la mémoire, des variables, des conditions et des boucles dans le langage C. Puis vous apprendrez à manipuler les procédures, les fonctions, les pointeurs, les directives au préprocesseur, les fichiers et quelques structures de données les plus courantes pour organiser les informations en mémoire : tableaux, listes, piles, files, … Enfin vous apprendrez quelques notions sur la Programmation Orienté Objet en C++.

OBJECTIFS SPECIFIQUES

- **Les bases de la programmation en C**
- **Les types composés**
- **Les pointeurs**
- **Les fonctions**
- **Les directives au préprocesseur**
- **Les structures de données**
- **La programmation modulaire ou structurée**

PRE-REQUIS

Afin de suivre ce cours sans difficulté, il serait intéressant:

- D'être familier avec l'utilisation de base d'un système d'exploitation : Windows ou Linux ;
- De posséder quelques notions d'algorithmique de base ;
- Posséder des connaissances de base sur l'architecture des ordinateurs.

PARTIE I : LES BASES DE LA PROGRAMMATION EN C

OBJECTIF DE LA PARTIE

Acquérir des connaissances de base solides de l'algorithmique et de la programmation en C. Plus spécifiquement, faire l'apprentissage de méthodologies permettant d'aborder la programmation avec aisance, la conception d'algorithmes pour résoudre des problèmes de nature scientifique et la traduction de ces algorithmes en langage C.

OBJECTIFS SPECIFIQUES

- Présentation du langage C
- Notions de variables et constantes
- Notions sur les opérateurs
- Notions sur les sorties formatées et la lecture de données
- Les structures conditionnelles
- Les structures itératives
- Notions sur les fonctions

CHAPITRE I : PRESENTATION DU LANGAGE C

Objectif du chapitre

Présenter le langage C à travers son histoire, son mode de fonctionnement, ces outils et sa structure.

Objectifs spécifiques

- Définition des concepts de compilation, de langage de bas niveau et haut niveau, de langage binaire, d'IDE et d'Exécutable ;
- Présentation de quelques langages de bas niveau et haut niveau ;
- Présentation de la différence entre le langage C et C++ ;
- Présentation des outils nécessaires à la programmation en C / C++.

I. Définition et historique du langage C

Votre ordinateur est une machine bizarre, c'est le moins que l'on puisse dire. On ne peut s'adresser à lui qu'en lui envoyant des **0** et des **1**. Ainsi, si je traduis « **Fais le calcul 3 + 5** » en langage informatique, ça pourrait donner quelque chose comme « **00101101100100110100111110** » (ceci n'est qu'un exemple).
Ce que vous voyez là, c'est le langage informatique de votre ordinateur, appelé **langage binaire**. Votre ordinateur ne connaît que ce langage-là et, comme vous pouvez le constater, c'est absolument incompréhensible, immonde et imbuvable.

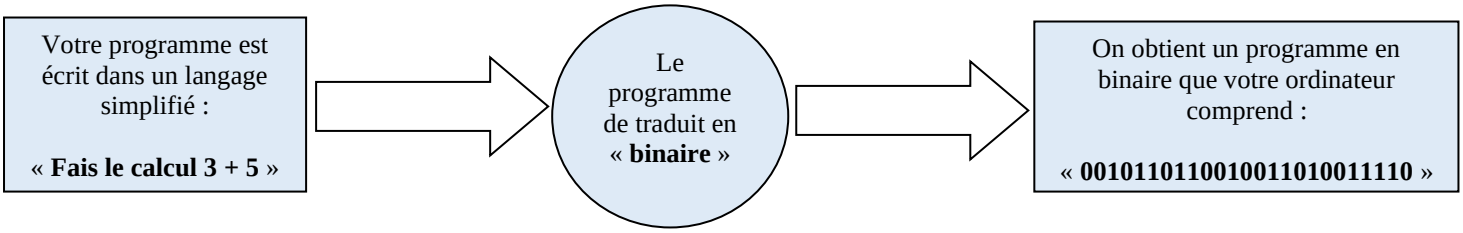
Donc voilà notre premier vrai problème :

Comment parler à l'ordinateur plus simplement qu'en binaire avec des 0 et des 1 ?

Votre ordinateur ne parle pas l'anglais et encore moins le français. Pourtant, il est inconcevable d'écrire un programme en langage binaire. Même les informaticiens les plus fous ne le font pas.

Eh bien, l'idée que les informaticiens ont eue, c'est d'inventer de **nouveaux langages** qui seraient ensuite traduits en binaire pour l'ordinateur. Le plus dur à faire, c'est de réaliser le programme qui fait la « **traduction** ». Heureusement, ce programme a déjà été écrit par des informaticiens et nous n'aurons pas à le refaire.
On va au contraire s'en servir pour écrire des phrases comme :
« **Fais le calcul 3 + 5** »
Qui seront traduites par le programme de "traduction" en quelque chose comme :
« **00101101100100110100111110** »

Si on fait un schéma de ce que je viens de dire, ça donne quelque chose comme ça :



Jusqu'ici j'ai parlé avec des mots simples, mais il faut savoir qu'en informatique il existe un mot pour chacune de ces choses-là. Tout au long de ce cours, vous allez d'ailleurs apprendre pas mal de vocabulaire. Non seulement vous aurez l'air de savoir de quoi vous parlez, mais si un jour (et ça arrivera) vous devez parler à un autre programmeur, vous saurez vous faire comprendre. Certes, les gens autour de vous vous regarderont comme des extra-terrestres, mais ça il ne faudra pas y faire attention.

Reprenons le schéma qu'on vient de voir.
La première case est « **Votre programme est écrit dans un langage simplifié** ». Ce fameux « **langage simplifié** » est appelé soit :

- « **langage de bas niveau** » permettant de créer un programme qui tient compte des caractéristiques particulières (registres, etc) de l'ordinateur censé exécuter le programme. Il permet de manipuler explicitement des registres, des adresses mémoires, des instructions machines.
- « **langage de haut niveau** » qui est orienté autour du problème à résoudre. Il permet d'écrire des programmes en utilisant des mots usuels des langues naturelles (très souvent de l'anglais) et des symboles mathématiques familiers. Un langage de haut niveau fait abstraction des caractéristiques techniques du matériel utilisé pour exécuter le programme, tels que les registres du processeur. Plus un langage est haut niveau, plus il est proche de votre vraie langue (comme le français). Un langage de haut niveau est donc facile à utiliser.

Il existe de nombreux langages de plus ou moins haut niveau en informatique dans lesquels vous pouvez écrire vos programmes. En voici quelques-uns par exemple :

- Le C
- Le C++
- Assembleur
- Java
- Visual Basic
- Delphi
- etc...

Notez que je ne les ai pas classés par niveau de langage, donc n'allez pas vous imaginer que le premier de la liste est plus facile que le dernier ou l'inverse. Ce sont juste quelques exemples de langages de programmation.

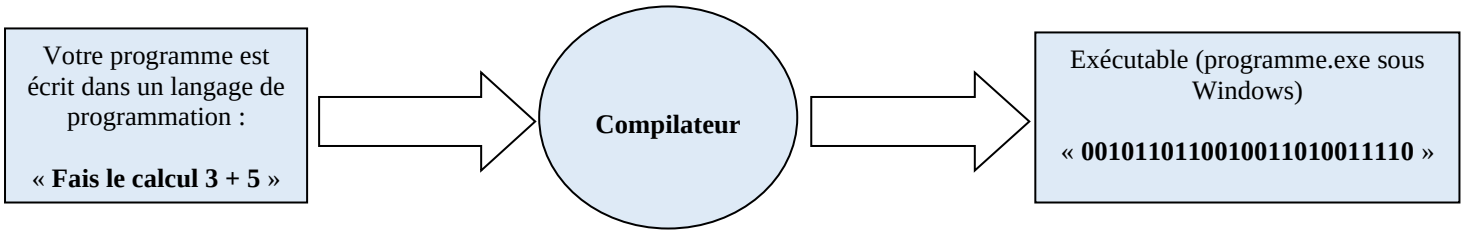
Certains de ces langages sont de haut niveau (Java, C++, Visual Basic, …) et d'autres de bas niveau (Assembleur, C, …). Dans le cadre de ce cours nous verrons en détail le langage C et présenterons quelques notions de la programmation Orienté Objet en C++.

Un autre mot de vocabulaire à retenir est : **code source**. Ce qu'on appelle le **code source**, c'est tout simplement le code de votre programme écrit dans un langage de programmation. C'est donc vous qui écrivez le code source, qui sera ensuite traduit en binaire.

Venons-en justement au « **programme de traduction** » qui traduit notre langage de programmation (comme le C ou le C++) en binaire. Ce programme a un nom : on l'appelle le **compilateur**. il existe un compilateur différent pour chaque langage de programmation. C'est d'ailleurs tout à fait logique : les langages étant différents, on ne traduit pas le **C** de la même manière qu'on traduit le **Delphi**. Vous verrez par la suite que, pour les langages C / C++ par exemple, il existe même plusieurs compilateurs différents.

Enfin, le programme binaire créé par le compilateur est appelé : l'**exécutable**. C'est d'ailleurs pour cette raison que les programmes (tout du moins sous Windows) ont l'extension « **.exe** » qui signifie **EXEcutable**.

Reprenons notre schéma de tout à l'heure, et utilisons cette fois des vrais mots d'informaticien. Ça donne :



Pourquoi choisir d'apprendre le C / C++ ?

Comme je vous l'ai dit plus haut, il existe de très nombreux langages de programmation. Doit-on commencer par l'un d'entre eux en particulier ? Grande question.

Pourtant, il faut bien faire un choix, commencer la programmation à un moment ou à un autre. Et là, vous avez en fait le choix entre :

- **Un langage de haut niveau** : c'est facile à utiliser, plutôt "grand public". Parmi eux, on compte Python, Ruby, Visual Basic, C++ (plus ou moins) et bien d'autres. Ces langages permettent d'écrire des programmes plus rapidement en règle générale. Ils nécessitent toutefois d'être accompagnés de fichiers pour qu'ils puissent s'exécuter (comme un interpréteur).

- **Un langage de bas niveau** (mais pas trop quand même !) : ils sont peut-être un peu plus difficiles certes, mais avec un langage comme le C vous allez en apprendre beaucoup plus sur la programmation et sur le fonctionnement de votre ordinateur. Vous serez ensuite largement plus capables d'apprendre un autre langage de programmation si vous le désirez. Vous serez donc plus autonomes. Par ailleurs, le C est un langage très populaire. Il est utilisé pour programmer une grande partie des logiciels que vous connaissez.

Voilà en gros les raisons qui m'incitent à vous apprendre le langage C plutôt qu'un autre. Je ne dis pas qu'il faut commencer par ça, mais je vous dis plutôt que c'est un bon choix qui va vous donner de solides connaissances.

Quelle est la différence en le C et le C++ ?

Voici ce qu'il faut savoir sur la différence entre les deux (02) avant de continuer :

Au tout début, à l'époque où les ordinateurs pesaient des tonnes et faisaient la taille de votre maison, on a commencé à inventer un langage de programmation appelé l'**Algol**.

Ensuite, les choses évoluant, on a créé un nouveau langage appelé le **CPL**, qui évolua lui-même en **BCPL**, puis qui pris le nom de **langage B** (euh si vous retenez pas tout ça c'est pas grave, j'écris juste pour faire semblant d'avoir de la culture).

Puis, un beau jour, on en est arrivés à créer encore un autre langage qu'on a appelé le **langage C**. Ce langage, même s'il a subi quelques modifications, il reste encore un des langages les plus utilisés aujourd'hui.

Un peu plus tard, on a proposé d'ajouter des choses au langage C. Une sorte d'amélioration si vous voulez. Ce nouveau langage, que l'on a appelé « **C++** », est entièrement basé sur le C. Le langage C++ n'est en fait rien d'autre que le langage C avec des ajouts.

Il y a plusieurs façons d'apprendre la programmation. Certaines personnes pensent qu'il est bien d'enseigner directement le C++. Elles n'ont peut-être pas tort. Après tout, si le C++ c'est du langage C « **avec des trucs en +** », ça revient un peu au même.

Pourtant, moi (et cet avis n'engage que moi), je pense que ce serait mélanger les choses. Aussi j'ai décidé que j'allais séparer mon cours en 2 grosses parties :

- **Le langage C**
- **Le langage C++**

Vu que vous aurez déjà appris le langage C dans un premier temps, quand on en viendra au langage C++ ça ira bien plus vite. Je n'aurai pas à vous réapprendre toutes les bases du C, j'aurai juste besoin de vous indiquer quels ajouts ont été faits dans le C++.

NB : Qu'il n'y ait pas de malentendus. Le langage C++ n'est pas meilleur que le langage C, il permet juste de programmer différemment. Il permet disons aussi au final de programmer un peu plus vite et de mieux organiser le code de son programme.
Le langage C n'est pas un « **vieux langage oublié** », au contraire il est encore très utilisé aujourd'hui. Il est à la base des plus grands systèmes d'exploitation tels **Unix (et donc Linux et Mac OS), ou encore Windows**.

Retenez donc : **le C et le C++** ne sont pas des langages concurrents, on peut faire autant de choses avec l'un qu'avec l'autre. Ce sont juste 2 manières de programmer assez différentes.

Après une première section plutôt introductive, nous commençons à entrer dans le vif du sujet. Nous allons répondre à la question suivante : « **De quels logiciels a-t-on besoin pour programmer ?** ».

Il n'y aura rien de difficile à faire, on va prendre le temps de se familiariser avec de nouveaux logiciels. Profitez-en ! Nous commencerons à vraiment programmer et il ne sera plus l'heure de faire la sieste !

II. Les outils nécessaires au programmeur

Alors à votre avis, de quels outils un programmeur a-t-il besoin ? Si vous avez attentivement suivi ce qui précédent, vous devez en connaître au moins un !
Bon ! Je ne vais pas vous laisser deviner plus longtemps. Voici le strict minimum pour un programmeur :

- Un **éditeur de texte** pour écrire le code source du programme. En théorie un logiciel comme le **Bloc-notes** sous Windows, ou « **vi** » sous Linux fait l'affaire. L'idéal, c'est d'avoir un éditeur de texte intelligent qui colore tout seul le code, ce qui vous permet de vous y repérer bien plus facilement ;
- Un **compilateur** pour transformer (« compiler ») votre source en binaire ;
- Un **débogueur** pour vous aider à traquer les erreurs dans votre programme. On n'a malheureusement pas encore inventé le « correcteur » qui corrigerait tout seul nos erreurs. Ceci dit, quand on sait bien se servir du débogueur, on peut facilement retrouver ses erreurs.

À partir de maintenant on a deux possibilités :

- Soit on récupère chacun de ces trois programmes **séparément**. C'est la méthode la plus compliquée, mais elle fonctionne. Sous Linux en particulier, bon nombre de programmeurs préfèrent utiliser ces trois programmes séparément. Je ne détaillerai pas cette méthode ici, je vais plutôt vous parler de la méthode simple ;
- Soit on utilise un programme « **trois-en-un** » qui combine éditeur de texte, compilateur et débogueur. Ces programmes « trois-en-un » sont appelés **IDE** pour « **Integrated Development Environment** » en anglais, ou encore **EDI** pour « **Environnement de Développement Intégré** ».

Il existe plusieurs environnements de développement. Au début, vous aurez peut-être un peu de mal à choisir celui qui vous plaît. Une chose est sûre en tout cas : vous pouvez réaliser n'importe quel type de programme, quel que soit l'IDE que vous choisissez.

Il me semble intéressant de vous montrer quelques IDE parmi les plus connus. Tous sont disponibles gratuitement :

- **Code::Blocks** : Il est gratuit et fonctionne sur la plupart des systèmes d'exploitation (Windows, Mac et Linux)
- **Visual C++** : Qui fonctionne sous Windows uniquement. C'est le plus célèbre IDE sous Windows. Il existe à la base en version payante (chère !), mais il existe heureusement une version gratuite intitulée **Visual Studio Express** qui est vraiment très bien (il y a peu de différences avec la version payante). Il est très complet et possède un puissant module de correction des erreurs (débogage).
- **Xcode** : Qui fonctionne sous Mac OS X uniquement. Il généralement fourni sur le CD d'installation de Mac OS X. C'est un IDE très apprécié par tous ceux qui font de la programmation sur Mac.

Tous ces IDE vous permettront de programmer et de suivre le reste de ce cours sans problème. Certains sont plus complets au niveau des options, d'autres un peu plus intuitifs à utiliser, mais dans tous les cas les programmes que vous créerez seront les mêmes quel que soit l'IDE que vous utilisez. Ce choix n'est donc pas si crucial qu'on pourrait le croire.

Vous l'aurez peut-être deviner, tout au long de tout ce cours, nous utiliserons l'IDE **Code::Blocks** parce qu'il est gratuit et qu'on le retrouve sur la plupart des systèmes d'exploitation.

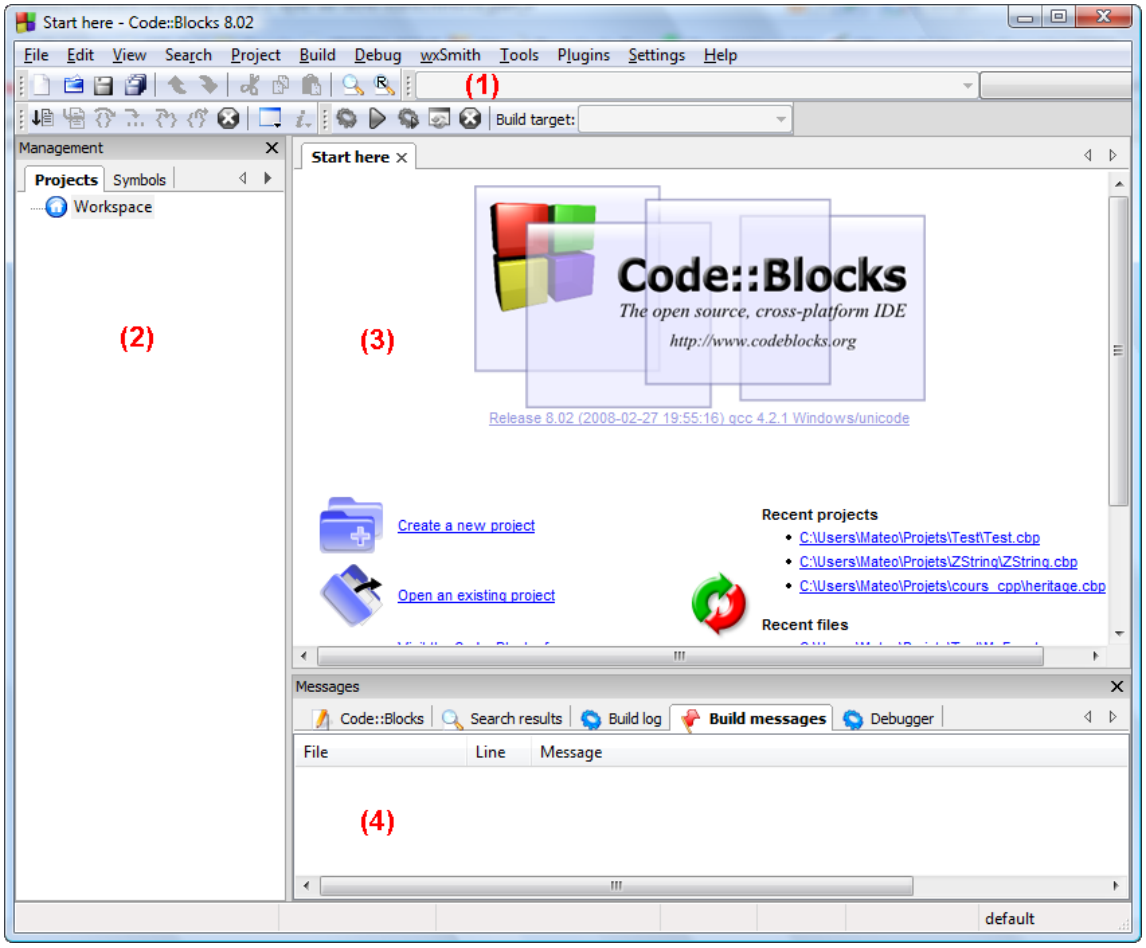
II.1. Comment télécharger Code::Blocks ?

Rendez-vous sur la page « http://www.codeblocks.org/downloads/binaries » et télécharger une version en fonction de votre système d'exploitation :

- Si vous êtes **sous Windows**, repérez la section « **Windows** » sur cette page. Téléchargez le logiciel en prenant le programme qui contient « **mingw** » dans le nom (**exemple : codeblocks-17.12mingw-setup.exe**). L'autre version étant sans compilateur, vous auriez eu du mal à compiler vos programmes !
- Si vous êtes **sous Linux**, choisissez le package qui correspond à votre distribution. **Exemple : CodeBlocks-13.12-mac.zip**
- Enfin, **sous Mac**, choisissez le fichier le plus récent de la liste. **Exemple : CodeBlocks-13.12-mac.zip**.

II.2. Comment utiliser Code::Blocks ?

Une fois télécharger et installer, on distingue 4 grandes sections dans la fenêtre, numérotées sur l'image ci-dessous :



1. **La barre d'outils** : elle comprend de nombreux boutons, mais seuls quelques-uns nous seront régulièrement utiles. Nous y reviendrons plus loin ;
2. **La liste des fichiers du projet** : c'est à gauche que s'affiche la liste de tous les fichiers source de votre programme. Notez que sur cette capture aucun projet n'a été créé, on ne voit donc pas encore de fichiers à l'intérieur de la liste ;
3. **La zone principale** : c'est là que vous pourrez écrire votre code en langage C ;
4. **La zone de notification** : aussi appelée la « **zone de la mort** », c'est ici que vous verrez les erreurs de compilation s'afficher si votre code comporte des erreurs. Cela arrive très régulièrement !

Intéressons-nous maintenant à une section particulière de la barre d'outils. Vous trouverez les boutons suivants (dans l'ordre) : « **Compiler** », « **Exécuter** », « **Compiler & Exécuter** » et « **Tout recompiler** ». Retenez-les, nous les utiliserons régulièrement.

Voici la signification de chacune des quatre icônes que vous voyez sur la fig. suivante, dans l'ordre :

- « **Compiler** » : tous les fichiers source de votre projet sont envoyés au compilateur qui va se charger de créer un exécutable. S'il y a des erreurs - ce qui a de fortes chances d'arriver tôt ou tard, l'exécutable ne sera pas créé et on vous indiquera les erreurs en bas de Code::Blocks ;
- « **Exécuter** » : cette icône lance juste le dernier exécutable que vous avez compilé. Cela vous permettra donc de tester votre programme et de voir ainsi ce qu'il donne. Dans l'ordre, si vous avez bien suivi, on doit d'abord compiler, puis exécuter pour tester ce que ça donne. On peut aussi utiliser le troisième bouton…
- « **Compiler & Exécuter** » : pas besoin d'être un génie pour comprendre que c'est la combinaison des deux boutons précédents. C'est d'ailleurs ce bouton que vous utiliserez le plus souvent. Notez que s'il y a des erreurs pendant la compilation (pendant la génération de l'exécutable), le programme ne sera pas exécuté. À la place, vous aurez droit à une « beeeeeeelle » liste d'erreurs à corriger !
- « **Tout reconstruire** » : quand vous faites « **Compiler** », Code::Blocks ne recompile en fait que les fichiers que vous avez modifiés et non les autres. Parfois - je dis bien parfois - vous aurez besoin de demander à Code::Blocks de vous recompiler tous les fichiers.

NB : Je vous conseille d'utiliser les raccourcis plutôt que de cliquer sur les boutons, parce que c'est quelque chose qu'on fait vraiment très souvent. Retenez en particulier qu'il faut taper sur « **F9** » pour faire « **Compiler & Exécuter** ».

II.3. Comment créer un projet sur Code::Blocks ?

Pour **créer un nouveau projet**, c'est très simple :

1. Allez dans le menu « **File / New / Project** ».
2. Dans la fenêtre qui s'ouvre, choisissez « **Console application** ».
3. Cliquez sur le bouton « **Go** » pour créer le projet. Un assistant s'ouvre.
4. Dans la liste qui s'affiche, sélectionnez « **C** » pour le langage C et faites « **Next** ».
5. On vous demande le nom de votre projet et dans quel dossier les fichiers source seront enregistrés.
6. Enfin, la dernière fenêtre vous permet de choisir de quelle façon le programme doit être compilé. Vous pouvez laisser les options par défaut, ça n'aura pas d'incidence sur ce que nous allons faire dans l'immédiat (veillez à ce que la case « **Debug** » ou « **Release** » au moins soit cochée).
7. Cliquez sur « **Finish** ». Code::Blocks vous créera un premier projet avec déjà un tout petit peu de code source dedans.

Dans le cadre de gauche « **Projects** », développez l'arborescence en cliquant sur le petit « **+** » pour afficher la liste des fichiers du projet. Vous devriez avoir au moins un « **main.c** » que vous pourrez ouvrir en double-cliquant dessus. Vous voilà parés !

III. Structure d'un programme C : Code minimal

Nous allons écrire et décrire un programme qui affiche « **Hello, world !** » en langage C.

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

L'analyse de ce code nous donne :

Ligne 1 et 2

Ce sont des lignes spéciales que l'on ne voit qu'en haut des fichiers source. Ces lignes sont facilement reconnaissables car elles commencent par un dièse #. Ces lignes spéciales, on les appelle **directives de préprocesseur**. Ce sont des lignes qui seront lues par un programme appelé **préprocesseur** : un programme qui se lance au début de la compilation.

Le mot « **include** » en anglais signifie « **inclure** » en français. Ces lignes demandent d'inclure des fichiers au projet, c'est-à-dire d'ajouter des fichiers pour la compilation.

Il y a 2 lignes, donc 2 fichiers inclus. Ces fichiers s'appellent « **stdio.h** » et « **stdlib.h** ». Ce sont des fichiers qui existent déjà, des fichiers source tout prêts. On les appelle des **bibliothèques** (certains parlent aussi de **librairies**). En gros, ces fichiers contiennent du code tout prêt qui permet d'afficher du texte à l'écran.

Sans ces fichiers, écrire du texte à l'écran aurait été mission impossible. **L'ordinateur à la base ne sait rien faire**, il faut tout lui dire.

De la ligne 4 à la ligne 7

C'est ce qu'on appelle une **fonction**. Un programme en langage C est constitué de fonctions, il ne contient quasiment que ça. Pour le moment, notre programme ne contient donc qu'une seule fonction.

Une fonction permet « grosso modo » de rassembler plusieurs commandes à l'ordinateur. Regroupées dans une fonction, les commandes permettent de faire quelque chose de précis. Par

exemple, on peut créer une fonction « **ouvrir_fichier** » qui contiendra une suite d'instructions pour l'ordinateur lui expliquant comment ouvrir un fichier.

L'avantage, c'est qu'une fois la fonction écrite, vous n'aurez plus qu'à dire « **ouvrir_fichier** », et votre ordinateur saura comment faire sans que vous ayez à tout répéter !

La **ligne 4** contient le nom de la fonction, « c'est le deuxième mot ».
Oui notre fonction s'appelle donc « **main** ». C'est un nom de fonction particulier qui signifie « **principal** ». « **main** » est la fonction principale de votre programme, c'est toujours par la fonction **main** que le programme commence.

Une fonction a un début et une fin, délimités par des accolades **{** et **}**. Toute la fonction **main** se trouve donc entre ces accolades.

Ces lignes à l'intérieur d'une fonction ont un nom. On les appelle **instructions**.

Chaque instruction est une commande à l'ordinateur. Chacune de ces lignes demande à l'ordinateur de faire quelque chose de précis.

L'instruction « **printf("Hello world! \n");** » demande à afficher le message « **Hello world!** » à l'écran. Quand votre programme arrivera à cette ligne, il va donc afficher un message à l'écran, puis passer à l'instruction suivante.

L'instruction « **return 0** » indique qu'on arrive à la fin de notre fonction main et demande de renvoyer la valeur 0.

En fait, chaque programme une fois terminé renvoie une valeur, par exemple pour dire que tout s'est bien passé (**0 = tout s'est bien passé, n'importe quelle autre valeur = erreur**). La plupart du temps, cette valeur n'est pas vraiment utilisée, mais il faut quand même en renvoyer une.

Votre programme aurait marché sans le return 0, mais on va dire que c'est plus propre et plus sérieux de le mettre, donc on le met.

Ligne 8

Notez la présence d'une ligne vide à la fin de ce code. Chaque fichier en C devrait normalement se terminer par une ligne vide comme celle-là. Si vous ne le faites pas, ce n'est pas grave, mais le compilateur risque de vous afficher un avertissement (warning).

Arguments du « main »

Il est possible de déclarer la fonction **main** avec arguments de la façon suivante :

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

Ou

```
int main(int argc, char **argv) { … }
```

Ainsi, un programme C peut prendre des arguments en ligne de commande lors de son exécution. Par exemple, si un fichier **monprog.c** a permis de générer un exécutable **monprog.exe** à la compilation, alors on peut invoquer le programme **monprog.exe** avec des arguments :

```
monprog.exe  argument1  argment2  argument3
```

Pour récupérer les arguments dans le programme C, on utilise les paramètres **argc** et **argv** du **main**. L'entier **argc** donne le nombre d'arguments rentrés dans la ligne de commande **plus 1**, et le paramètre **argv** est un tableau de chaînes de caractères qui contient comme éléments :

- Le premier élément **argv[0]** est une chaîne qui contient le nom du fichier exécutable du programme ;
- Les éléments suivants **argv[1], argv[2], etc...** sont des chaînes de caractères qui contiennent les arguments passés en ligne de commande.

Tests de connaissances

1. Définir : Langage de programmation, Compilation, IDE.
2. Pourquoi dit-t-on que le langage C est un langage bas niveau ?
3. Quelle différence faites-vous entre langage compilé et langage interprété ? Donner quelques exemples.
4. Comment s'appelle le programme chargé de traduire un code source en binaire ?
5. Quelle est la seule mémoire qui n'est pas vidée lorsque l'ordinateur est éteint ?
6. Citer quelques outils nécessaires pour développer en langage C.
7. Quelle est la structure minimale d'un programme C ?

CHAPITRE II : LE MONDE DES VARIABLES

Objectif du chapitre

Présenter de quelques éléments de base associés au langage C.

Objectifs spécifiques

- Définition des concepts d'identificateur, d'objet, de type de données, de variable et constante ;
- Manipulation des variables et constantes en langage C ;
- Définition des opérations possibles sur les variables et/ou les constantes ;
- Gestion des sorties formatées et lecture des données.

I. Notion d'identificateur

Un **identificateur** est un mot composé de lettres et des chiffres ainsi que certains caractères spéciaux comme le souligné « _ ». Un identificateur doit commencer par une lettre et ne doit pas contenir des caractères accentués. Il doit être choisit de façon à refléter le problème posé et à facilement se rappeler.

Exemple : un_homme, unhomme

Les caractères proscrits dans les identificateurs sont :

‘’ ‘’	+	-	/	*	.	;	,	%

NB : Il est important de noter qu'identificateur ne doit pas également contenir d'espaces.

II. Notion d'objet

Un **objet** est un outil ou un être du monde réel que l'on manipule dans l'algorithme.

L'univers des objets du traitement d'une information est un ensemble fini d'éléments. Il est donc possible de le décrire entièrement et sans ambigüité en décrivant chaque objet. Le traitement d'un objet concerne la valeur de cet objet. Si cette valeur ne peut être modifiée nous parlons de **constante** sinon nous parlons de **variable**.

Un objet est parfaitement définit si nous connaissons :

- **Son nom**
- **Son type**
- **Sa valeur**

III. Intérêt des types de données

On y voit surtout deux avantages :

- **Pour décrire les objets de même nature et admettant les mêmes opérations** : il suffit d'avoir décrit le type une fois pour tout et d'indiquer pour un type donné, la liste d'objets associés. On évite ainsi les répétitions pénibles.
- **On dispose d'un moyen pour détecter un certain nombre d'erreur sans exécuter le programme** : simplement par l'examen des opérations sur les objets et en contrôlant qu'elles sont licites.

Exemple :

Définissons les types suivants :

- **SOLIDE** : matière ayant une forme propre. Comme exemple d'opérateur sur un SOLIDE, on peut avoir « Fondre ».
- **LIQUIDE** : matière tendant à écouler. Comme opérateur on a « Bouillir », « Boire ».

Ainsi, l'on peut définir la suite suivante :

Objets	Actions ou opérations possibles
caoutchouc, beurre : SOLIDE	Fondre le caoutchouc, Fondre le beurre
eau, vin : LIQUIDE	Boire le vin, Bouillir l'eau

Une opération telle que « **Fondre le vin** » serait erronée car le **vin** est un objet de type **LIQUIDE** et l'opération « Fondre » n'est pas associée à ce type.

IV. Les variables

Alors une variable, c'est quoi ? Eh bien c'est une petite information temporaire qu'on stocke dans la RAM (dans des cases mémoire identifiées par des adresses mémoire unique). Tout simplement, on dit qu'elle est « variable » car c'est une valeur qui peut changer pendant le déroulement du programme.

Nos programmes, vous allez le voir, sont remplis de variables. Vous allez en voir partout, à toutes les sauces.

En langage C et comme dans la majorité dans langage de programmation, une variable est définie par :

1. Un **nom (identificateur)** : c'est ce qui permet de repérer la variable et de l'utiliser.
2. Un **type** : Qui permet de définir quelles données la variable peut contenir.
3. Eventuellement une **valeur** : Qui correspond à l'information stockée dans la variable.

Les **variables** sont stockées en mémoire (RAM) dans des sortes d'espaces appelés **cases mémoire** (correspondant à la valeur de la variable qui dépend du type de donné), elles mêmes identifiées dans la RAM par des **adresses mémoire unique** (correspondant au nom de la variable).

Ainsi on pourrait croire que pour manipuler les variables, il faudrait retenir ces adresses mémoires. Mais en réalité, en programmant en C et comme dans la majorité dans langage de programmation, on n'aura pas à retenir l'adresse mémoire : à la place, on va juste indiquer des noms de variables. C'est le compilateur qui fera la conversion entre le nom et l'adresse.

IV.1. Comment donner un nom à une variable ?

En langage C, chaque variable doit avoir un nom. Si nous voulions par exemple écrire un programme C qui fait l'addition de deux nombres on aimerait bien déclarer une variable du genre « **somme de deux nombres** » ou quelque chose dans ce sens. Hélas, il y a quelques contraintes. Vous ne pouvez pas appeler une variable n'importe comment :

- Il ne peut y avoir que des minuscules, majuscules et des chiffres (abcABC012) ;
- Votre nom de variable doit commencé par une lettre ;
- Les espaces sont interdits. À la place, on peut utiliser le caractère « **underscore (_)** » (qui ressemble à un trait de soulignement). C'est le seul caractère différent des lettres et chiffres autorisé ;
- Vous n'avez pas le droit d'utiliser des accents (é, à, ê, etc.).

Enfin, et c'est très important à savoir, le langage C fait la différence entre les majuscules et les minuscules. Pour votre culture, sachez qu'on dit que c'est un langage qui « **respecte la casse** ». Donc, du coup, les variables **largeur**, **LARGEUR** ou encore **LArgEuR** sont trois variables différentes en langage C, même si pour nous ça a l'air de signifier la même chose !

Voici quelques exemples de noms de variables corrects : **somme_nombres, sommeNombres, prenom, nom, numero_de_telephone, numeroDeTelephone**.

Chaque programmeur a sa propre façon de nommer des variables. Pendant ce cours, je vais vous montrer ma manière de faire :

- Je commence tous mes noms de variables par une lettre minuscule ;
- S'il y a plusieurs mots dans mon nom de variable, je mets une lettre majuscule au début de chaque nouveau mot.

Je vais vous demander de faire de la même manière que moi, ça nous permettra d'être sur la même longueur d'ondes.

Attention : Quoi que vous fassiez, faites en sorte de donner des noms clairs à vos variables. On aurait pu abréger « **sommeNombres** », en l'écrivant par exemple « **sN** ». C'est peut-être plus court, mais c'est beaucoup moins clair pour vous quand vous relisez votre code. N'ayez donc pas peur de donner des noms un peu plus longs pour que ça reste compréhensible.

IV.2. Les types de variables

Le **type** identifie la nature d'une donnée. Le type d'une variable détermine sa taille, les données qu'elle peut contenir et les opérations permises sur cette variable.

Lorsque vous créez une variable, vous allez donc devoir indiquer son type. Voici les principaux types de variables existant en langage C :

Type entier (int)

Le **type int** (abréviation de l'anglais **integer**) est une représentation des nombres entiers. Comme toute variable informatique, un « **int** » ne peut prendre qu'un nombre fini de valeur. Les opérations arithmétiques binaires **+, −, ∗ , /** sont définies sur les « **int** » et donnent toujours pour résultat un « **int** ».

Types réels (float et double)

Les **types float et double** permettent de représenter des nombres réels avec une certaine précision (suivant une représentation des nombres appelée **virgule flottante ou nombre flottant**).

Un nombre réel tel qu'il est ainsi représenté possède une **mantisse** (des chiffres) et un **exposant** qui correspond à la multiplication par une certaine puissance de 10.

Exemple : **$3.546E^{-3}$** est égal à **3.546×10^{-3}**, ce qui fait **0.003546**.

Le **type double (codé sur 8 octets)** est plus précis que le type **float (codé sur 4 octets)**.
Les quatre opérations binaires **+, −, ∗, /** sont définies sur les réels.

Type char

Le **type char** (abréviation de l'anglais **character**) est un type caractère codé sur **1 octet**. C'est la plus petite donnée qui puisse être stockée dans une variable. Les valeurs (**de −126 à 125**) peuvent représenter des caractères conventionnels. Par exemple, les caractères alphabétiques majuscules A,B,...,Z ont pour valeur 65,66,...,90 et les minuscules correspondantes a,b,...,z ont pour valeurs 97,98,...,122. On appelle ce codage des caractères : le **code ASCII**. Une variable de type char peut être considérée soit comme **un nombre**, soit comme **un caractère** que l'on peut afficher.

Type unsigned

Aux types **int** et **char** correspondent des **types unsigned int (aussi sur 4 octets) et unsigned char (aussi sur 1 octet)** qui représentent uniquement des valeurs positives.

Un **unsigned int** sur 4 octets va de **0 à $2^{32} - 1$** (c'est-à-dire de **0 à 4294967295**).
Un **unsigned char** sur 1 octet va de **0 à $2^{8} - 1$** (c'est-à-dire de **0 à 255**).

IV.3. Déclarer une variable

On y arrive. Maintenant, créez un nouveau projet console ou utiliser celui créer précédemment dans ce cours. On va voir comment déclarer une variable, c'est-à-dire **demandé à l'ordinateur la permission d'utiliser un peu de mémoire**.

Une déclaration de variable, c'est très simple maintenant que vous savez tout ce qu'il faut. Il suffit dans l'ordre :

1. D'indiquer le type de la variable que l'on veut créer ;
2. D'insérer un espace ;
3. D'indiquer le nom que vous voulez donner à la variable ;
4. Et enfin, de ne pas oublier le point-virgule (**;**).

Par exemple, si je veux créer ma variable « **sommeNombres** » de type « **double** », je dois taper la ligne suivante :

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

int main() {
    double sommeNombres;

    return 0;
}
```

Ce qu'on fait là s'appelle une **déclaration de variable** (un vocabulaire à retenir). Vous devez faire les déclarations de variables au début des fonctions. Comme pour le moment on n'a qu'une seule fonction (la fonction main).

Si vous avez plusieurs variables du même type à déclarer, inutile de faire une ligne pour chaque variable. Il vous suffit de séparer les différents noms de variables par des virgules sur la même ligne : **double sommeNombres, nombreA, nombreB;**. Cela créera trois variables « **double** » appelées **sommeNombres, nombreA et nombreB**.

IV.4. Affecter une valeur à une variable

C'est tout ce qu'il y a de plus bête. Si vous voulez donner une valeur à la variable **sommeNombres**, il suffit de procéder comme ceci :

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

int main() {
    double sommeNombres;
    sommeNombres= 1.5;

    return 0;
}
```

Ou bien

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

int main() {
    double sommeNombres= 1.5;

    return 0;
}
```

V. Les constantes

Il arrive parfois que l'on ait besoin d'utiliser une variable dont on voudrait qu'elle garde la même valeur pendant toute la durée du programme. C'est-à-dire qu'une fois déclarée, vous voudriez que votre variable conserve sa valeur et que personne n'ait le droit de changer ce qu'elle contient.

Ces variables particulières sont appelées **constantes**, justement parce que leur valeur reste constante.
Une **constante** est un identificateur, (par exemple **PI**), utilisé pour conserver une valeur (par exemple **6.67428×10^{-11}**). L'assignation d'une valeur à une constante n'est permise que lors de sa déclaration.

On distingue deux (02) types de constantes à savoir :

- **Les constantes littérales**
- **Les constantes symboliques**

V.1. Les constantes littérales

Les **constantes littérales** représentent des valeurs numériques ou caractères entrés dans le code source du programme et réservés et initialisés lors de la phase de compilation de code source.

Il existe quatre types de constantes littérales :

- Les constantes entières (éventuellement long)
- Les constantes flottantes (réelles)
- Les constantes caractères
- Les chaînes de caractères littérales

Les constantes entières

Il existe trois notations pour les constantes entières :

- Décimal
- Octal
- Hexadécimal

Exemple :

- Constante dans le système décimal : 90
- Constante octale : 0132 (qui correspond à 90 en décimal)
  NB** : Les constantes octales commencent par un **zéro (0)
- Constante hexadécimale : 0x5A (qui correspond à 90 en décimal)
  NB** : Les constantes hexadécimales commencent par **0x

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

int main() {
    printf("Déc = %d, Oct = %d, Hex = %d", 90, 0132, 0x5A);
    return 0;
}
```

NB : **printf** permet d'afficher un texte à l'écran. Cette fonction sera expliquée plus en détail plus bas dans ce cours.

Il existe une convention permettant d'indiquer au compilateur qu'une constante entière est de type **long int**. Cette convention consiste à faire suivre la constante de la **lettre L** (en majuscule ou minuscule).

Exemple :

- O**L** (décimal long int)
- 0457**L** (octal long int)
- 0x7AFF**L** (Hexadecimal long int)

Les constantes flottantes

La notation utilisée est la notation classique par **la mantisse et l'exposant**. Pour introduire l'exposant, on peut utiliser la **lettre e** en minuscule ou majuscule.

Exemple :

NOTATION MATHEMATIQUE	NOTATION C
2	2.
0.3	.3
2.3	2.3
2×10^4	2e4
2×10^4	2.e4
0.3×10^4	.3e4
2.3×10^4	2.3e4
2.3×10^{-4}	2.3e – 4

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

int main() {
    printf("%f", 2e4); //affichera 20000.0000
    return 0;
}
```

Toute constante flottante est considérée comme étant de type flottant double précision.

Les constantes caractères

La **valeur d'une constante caractère** est la valeur numérique du caractère dans le code de la machine.

Si le caractère dispose d'une représentation imprimable, une constante caractère s'écrit entouré du signe « ' ».

Exemple : 'g'

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

int main() {
    printf("%c", 'g'); //affichera g
    return 0;
}
```

Pour les caractères ne disposant pas de représentation imprimable, le langage C autorise une notation à l'aide d'un caractère d'escape qui est « \ ».

Les plus utilisés sont :

- **‘ \n ’** : new line (saut de ligne)
- **‘ \t ’** : horizontal tabulation
- **‘ \b ’** : back space
- **‘ \r ’** : carriage return (retour chariot : entrée)
- **‘ \f ’** : form feed (page suivante)
- **‘ \\ ’** : back slash
- **‘ \’ ’** : single quote (simple quote)
- **‘ \a ’** : bell (bip sonore)

Les chaînes de caractères littérales

Une **chaîne de caractères littérale** est une suite de caractères entourés du signe « " ».

Exemple : " Entrez une valeur "

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

int main() {
    printf("%s", "Entrez une valeur"); // ou bien « printf("Entrez une valeur"); »
    return 0;
}
```

Les caractères non imprimables sont utilisables dans les chaînes :

Exemple : " Si on imprime cette ligne, on aura : \n deux lignes à cause de new line ".

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

int main() {
    printf("Si on imprime cette ligne, on aura : \n deux lignes à cause de new line ");
    /* affichera deux lignes
       1. Si on imprime cette ligne, on aura :
       2. deux lignes à cause de new ligne
    */
    return 0;
}
```

V.2. Les constantes symboliques

Les **constantes symboliques** sont définies par le programmeur ou pour le programmeur dans des fichiers inclus dans le programme C comme par exemple **stdio.h**.

Les constantes « const »

Pour déclarer une constante, il est possible d'utiliser le mot « **const** » juste devant le type quand vous déclarez votre variable. Par ailleurs, il faut obligatoirement lui donner une valeur au moment de sa déclaration.

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

int main() {
    const int MAX_NOMBRES = 10;
    return 0;
}
```

NB : Ce n'est pas une obligation, mais par convention on écrit les noms des constantes entièrement en majuscules comme je viens de le faire là. Cela nous permet ainsi de distinguer facilement les constantes des variables. Notez qu'on utilise **l'underscore « _ »** à la place de l'espace.

Les constantes de préprocesseur

Le **préprocesseur** est un programme qui s'exécute juste avant la compilation. Il assure une phase préliminaire la compilation des programmes informatiques écrits dans les langages de programmation C et C++.
Dans tout programme C, on trouve dans les codes source des lignes un peu particulières appelées **directives de préprocesseur**. Ces directives de préprocesseur ont la caractéristique suivante : elles commencent toujours par le symbole « # ». Elles sont donc faciles à reconnaître.

La première (et seule) directive que nous ayons vue pour l'instant est « **#include** ». Cette directive permet d'inclure le contenu d'un fichier dans un autre.

On s'en sert en particulier pour inclure des fichiers d'extension « **.h** » tels que **stdlib.h, stdio.h, …**

Eh bien, il existe une autre directive de préprocesseur appelé « **#define** » qui permet de définir des constantes de préprocesseur. Cela permet ainsi d'associer une valeur à un mot.

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

#define MAX_NOMBRES 10

int main() {
    printf("%d", MAX_NOMBRES);
    return 0;
}
```

Différence entre constantes « const » et constantes de pour de préprocesseur

Les **constantes de préprocesseur** permettent au programmeur de créer un nom pour une constante et d'utiliser ce nom tout au long du programme. Si la constante doit être modifié au sein du programme, il suffit de la modifier une seule fois dans la directive de précompilation **#define**;

lorsque le programme est recompilé, toutes les occurrences de la constante dans le programme sont automatiquement modifiés. **Remarque** : Tout ce qui se trouve à droite du nom de la constante de préprocesseur remplace la constante de préprocesseur. Ainsi, **#define PI = 3.14159** entraîne le remplacement de chaque occurrence de **PI** par **= 3.14159**.

Il est donc important de noter que les constantes de préprocesseur ne nécessitent par d'emplacements mémoire contrairement aux **constantes « const »** qui qualifient une variable comme étant « **en lecture seule** » (c'est-à-dire que sa valeur n'est pas sensée être modifiée), mais exigent un emplacement mémoire de la taille de leur type de donnée (c'est-à-dire qu'il est possible de modifier sa valeur à partir de son adresse mémoire à l'aide des pointeurs que nous verrons plus bas dans ce cours).

Les énumérations

L'énumération est une suite de constantes renfermées dans une seule.

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

int main() {
    enum DIRECTION {NORD, SUD, EST, OUEST}; // Exemple de déclaration d'une énumération

    printf("Nord : %d\n", NORD);    // affichera 0
    printf("Sud : %d\n", SUD);      // affichera 1
    printf("Est : %d\n", EST);      // affichera 2
    printf("Ouest : %d\n", OUEST);  // affichera 3
    return 0;
}
```

Si vous ne donnez aucune valeur, par défaut la **première valeur sera zéro (0)**.

Il est possible de donner une valeur entière à ces constantes, si vous ne donnez qu'une seule valeur à une seule constante les suivantes prendront la même valeur incrémentée de **1**.

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

int main() {
    enum DIRECTION {NORD= 10, SUD, EST= 20, OUEST};

    printf("Nord : %d\n", NORD);    // affichera 10
    printf("Sud : %d\n", SUD);      // affichera 11
    printf("Est : %d\n", EST);      // affichera 20
    printf("Ouest : %d\n", OUEST);  // affichera 21
    return 0;
}
```

VI. Les opérateurs

	NOTATION	DESCRIPTION	EXEMPLE
Opérateurs arithmétiques	+	Addition	int nb= 5 + 4;
	-	Soustraction	int nb= 9 - 5;
	/	Division entière	int nb= 9 / 2;
	*	Multiplication	int nb= 5 * 4;
	%	Modulo (reste de la division entière)	int nb= 9 % 2;
Opérateurs logiques	&&	ET	(9 > 5) && (3 > 8) //retourne false
	||	OU	(9 > 5) || (3 > 8) //retourne true
	!	NON	!(3 < 8) //retourne false
Opérateurs de comparaison	==	Egal	(9 = = 10); //retourne false
	!=	Différent de	(9 != 10); //retourne true
	>	Strictement supérieur	(9 > 10); //retourne false
	<	Strictement inférieur	(9 <10); //retourne true
	>=	Supérieur ou égal	(9 >= 10); //retourne false
	<=	Inférieur ou égal	(9 <= 10); //retourne true
Opérateurs d'affectation	=	Affectation	int nb= 3;
	+=		int nb+= 3; → int nb= nb + 3;
	-=		int nb-= 3; → int nb= nb - 3;
	*=		int nb*= 3; → int nb= nb * 3;
	/=		int nb/= 3; → int nb= nb / 3;
	%=		int nb%= 3; → int nb= nb % 3;
Opérateur d'incrémentation	++	Il peut être utilisé soit de manière préfixé, soit de manière post fixé.	++n ; (manière préfixé)
			n++ ; (manière post fixé)
Opérateur de décrémentation	--		--n ; (manière préfixé)
			n-- ; (manière post fixé)

VII. Afficher un texte à l'écran (sorties formatées)

Il existe une fonction standard permettant de réaliser des sorties formatées : il s'agit de la fonction **printf** issue de la bibliothèque « **stdio.h** ». Cette fonction effectue les sorties suivant un modèle (ou format).

Syntaxe :

```
printf("<format>",<Expr1>,<Expr2>, ... )
```

- **"<format>"** : correspond au **format de représentation des données**
- **<Expr1>, <Expr2>, ...** : correspondent aux **variables et expressions** dont les valeurs sont à représenter

Le **format de représentation** est en fait une chaîne de caractères qui peut contenir:

- Du texte
- Des séquences d'échappement (**exemple : ' \n ', ' \t ', …**)
- **Des spécificateurs de format** : qui indiquent la manière dont les valeurs des expressions **<Expr1>, <Expr2>, ...** sont imprimées. La partie **"<format>"** contient exactement un spécificateur de format pour chaque expression **<Expr1>, <Expr2>, ...** Les spécificateurs de format commencent toujours par le symbole **%** et se terminent par **un ou deux caractères** qui indiquent le format d'impression. Les spécificateurs de format sont encore appelés **symboles de conversion** car ils impliquent une conversion d'un nombre en chaîne de caractères.

Il existe une pléthore de spécificateurs de format pour l'impression d'une donnée à savoir :

SYMBOLE	TYPE	IMPRESSION COMME
%d ou %i	int	Entier relatif
%u	unsigned int	Entier naturel non signé
%o	int	Entier exprimé en Octal
%x	int	Entier exprimé en Hexadécimal
%c	char	Caractère
%f	double	Nombre réel en notation décimal
%e	double	Nombre réel en notation scientifique
%s	char*	Chaîne de caractères

NB : Les spécificateurs **%d, %i, %u, %o, %x** peuvent seulement représenter des valeurs du **type int ou unsigned int**. Une valeur trop grande pour être codée dans deux octets est **coupée sans avertissement** si nous utilisons **%d**. Ainsi, pour pouvoir traiter correctement les arguments du type long, il faut utiliser les spécificateurs **%ld, %li, %lu, %lo, %lx**.

Exemple :

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

int main() {
    int nb= 101;

    printf("NB = %d \n", nb);     // affichera « NB = 101 »
    printf("NB = %o \n", nb);     // affichera « NB = 145 »
    printf("NB = %x \n", nb);     // affichera « NB = 65 »
    printf("NB = %e \n", nb);     // affichera « NB = e »

    return 0;
}
```

VIII. Récupérer une saisie (lecture de données)

Les variables vont en fait commencer à devenir intéressantes maintenant. On va apprendre à demander à l'utilisateur de taper un nombre dans la console. Ce nombre, on va le récupérer et le stocker dans une variable.

Pour demander à l'utilisateur d'entrer quelque chose dans la console, on va utiliser la fonction toute prête : **scanf** accessible depuis la bibliothèque standard « **stdio.h** ».

La fonction **scanf** reçoit ses données à partir du **fichier d'entrée standard stdin (par défaut le clavier)**.

Cette fonction ressemble beaucoup à **printf**. Vous devez mettre un format pour indiquer ce que l'utilisateur doit entrer (un int, un float, ...). Puis vous devez ensuite indiquer le nom (adresse mémoire) de la variable qui va recevoir le nombre.

Syntaxe :

```
scanf("<format>",<AdrVar1>,<AdrVar2>, ...)
```

- **"<format>"** : correspond au **format de lecture des données**. Il détermine comment les données reçues doivent être interprétées. Il est constitué principalement de **spécificateurs de format**.
- **<AdrVar1>, <AdrVar2>, ...** : correspondent aux **adresses des variables** auxquelles les données seront attribuées. Les données reçues correctement sont mémorisées successivement aux adresses indiquées par **<AdrVar1>, <AdrVar2>, ...** L'adresse d'une variable est indiquée par le **nom de la variable** précédé de l'opérateur **&**.

Il existe une pléthore de spécificateurs de format pour la lecture d'une donnée à savoir :

SYMBOLE	LECTURE D'UN(E)	TYPE
%d ou %i	Entier relatif	int*
%u	Entier naturel non signé	int*
%o	Entier exprimé en Octal	int*
%b	Entier exprimé en Hexadécimal	int*
%c	Caractère	char*
%s	Chaîne de caractères	char*
%f ou %e	Nombre réel en notation décimale ou scientifique	float*

Attention : Le **symbole *** indique que l'argument n'est pas une variable, mais l'adresse d'une variable de ce type (c'est-à-dire un **pointeur** sur une variable de ce type – nous verrons les pointeurs plutard).

NB :

1. **Le type long** : Si nous voulons lire une donnée du type long, nous devons utiliser les spécificateurs **%ld, %li, %lu, %lo, %lx**. (Sinon, le nombre est simplement coupé à la taille de int).
2. **Le type double** : Si nous voulons lire une donnée du type double, nous devons utiliser les spécificateurs **%le ou %lf**.
3. **Le type long double** : Si nous voulons lire une donnée du type long double, nous devons utiliser les spécificateurs **%Le ou %Lf**.

Exemple :

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

int main() {
    int nb;
    printf("Entrez une valeur : ");
    scanf("%d", &nb);
    printf("Le double de %d est : %d", nb, (nb*2));

    return 0;
}
```

Indication de la largeur maximale

Pour tous les spécificateurs, nous pouvons indiquer la largeur maximale du champ à évaluer pour une donnée. Les chiffres qui passent au-delà du champ défini sont attribués à la prochaine variable qui sera lue !

Exemple : Soit le code C suivant :

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

int main() {
    int nb1, nb2;
    printf("Entrez deux valeurs : ");
    scanf("%4d %2d", &nb1, &nb2);
    printf("nb1= %d \n", nb1);
    printf("nb2= %d \n", nb2);

    return 0;
}
```

Si nous entrons le nombre **1234567** lors de l'exécution, nous obtiendrons les affectations suivantes:

- **nb1=1234**
- **nb2=56**

Le **chiffre 7** sera gardé pour la prochaine instruction de lecture.

Formats spéciaux

Si la chaîne de format contient des caractères tels que des signes d'espacement ou autres, alors ces symboles doivent être introduits exactement dans l'ordre indiqué selon le format définit.

Exemple :

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

int main() {
    int jour, mois, annee;
    printf("Entrez la date au format JJ/MM/AAAA : ");
    scanf("%d/%d/%d", &jour, &mois, &annee);

    printf("La date est : %d/%d/%d", jour, mois, annee);
    return 0;
}
```

Entrées acceptées	Entrées rejetées
13/09/2019	13 09 2019
13/9/2019	13 /9 /2019

Nombre de valeurs lues

Lors de l'évaluation des données, **scanf** s'arrête si la chaîne de format a été travaillée jusqu'à la fin ou si une donnée ne correspond pas au format indiqué. **scanf** retourne comme résultat le nombre d'arguments correctement reçus et affectés.

Exemple :

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

int main() {
    int jour, mois, annee, nb;
    printf("Entrez une date au format JJ/MM/AAAA : ");
    nb= scanf("%d %d %d", &jour, &mois, &annee);

    printf("Nombre de valeurs lues : %d \n", nb);
    printf("La date est : %d %d %d", jour, mois, annee);
    return 0;
}
```

Entrées		Nombre de valeurs lues	Jour	Mois	Année
13 09 2019	➔	3	13	09	2019
13/09/2019	➔	1	13	-	-
13.09 2019	➔	1	13	-	-
13 09 20.19	➔	3	12	09	20

Tests de connaissances

1. Définir : Identificateur, Objet, Variable, Constante, Enumération, Préprocesseur.
2. Quelle est la seule mémoire qui n'est pas vidée lorsque l'ordinateur est éteint ?
3. Quelles sont les caractéristiques d'une variable ?
4. Quelle est la différence entre une constante « const » et une constante de préprocesseur ?
5. Quelle est la différence entre une incrémentation et une décrémentation ? Donner un exemple
6. Quelle est la différence entre opérateurs arithmétiques et opérateurs logiques ?
7. A quoi sert « & » devant une variable ?
8. Quelle fonction permet d'afficher un texte à l'écran ? Donner sa syntaxe.
9. Quelle fonction permet de récupérer une valeur entrée par un utilisateur ? Donner sa syntaxe ?
10. Quel symbole permet d'effectuer un retour à la ligne à l'écran ?

CHAPITRE III : LES CONDITIONS ET BOUCLES

Objectif du chapitre

Construire des programmes en langage C comportant des traitements conditionnels et itératifs.

Objectifs spécifiques

- Définition et manipulation des structures conditionnelles (if..else, switch)
- Définition et manipulation des structures itératives (for, while, do..while)

I. Structures conditionnelles

Ces structures sont utilisées pour décider de l'exécution d'un bloc d'instruction : est-ce-que ce bloc est exécuté ou non ? Ou bien pour choisir entre l'exécution de deux blocs différents.
Les exemples précédents montrent des programmes dont les instructions doivent s'exécuter dans l'ordre, de la première à la dernière. Ici, nous introduisons une instruction précisant que le déroulement ne sera plus séquentiel. Cette instruction est appelée **une conditionnelle**. Il s'agit de représenter une alternative où, selon les cas, un bloc d'instructions est exécuté plutôt qu'un autre.

La syntaxe de cette instruction est :

```
if (<condition>) {
     <Bloc d'instructions 1>;
}
else {
     <Bloc d'instructions 2>;
}
```

Cette instruction est composée de deux (02) parties distinctes : la condition introduite par la clause « **if** », et la clause « **else** ». La condition est une expression dont la valeur est de type booléen (**« true » ou « false »**). Elle est évaluée. Si elle vaut « **true** », les instructions de la clause « **if** » sont exécutées. Dans le cas contraire, les instructions de la clause « **else** » sont exécutées.

Exemple :

Ecrire un programme C qui prend un nombre entier au clavier et imprime la parité de ce nombre.

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

int main() {
    int nb;
    printf("Entrez un nombre entier : ");
    scanf("%d", &nb);

    if((nb%2) == 0) {
        printf("%d est un nombre pair.", nb);
    }
    else {
        printf("%d est un nombre impair.", nb);
    }

    return 0;
}
```

I.1. Structures conditionnelles imbriquées

Dans l'exemple précédent portant sur la parité d'un nombre, nous n'avions qu'une condition à évaluer pour parvenir au résultat recherché. Mais il peut arriver dans certains cas que nous ayons besoin d'évaluer plusieurs conditions. Dans ce cas, il suffit d'empiler les structures conditionnelles les uns en dessous des autres (**imbrication**) comme suit :

```
if (<condition 1>) {
      <Bloc d'instructions 1>;
}
else if (<condition 2>) {
      <Bloc d'instructions 2>;
}
else if (<condition 3>) {
      <Bloc d'instructions 3>;
}
else if (<condition 4>) {
      <Bloc d'instructions 4>;
}
...
...
```

Exemple :

Un magasin de reprographie facture 50 FCFA les dix premières photocopies, 25 FCFA les vingt suivantes et 10 FCFA au-delà. Ecrivez un programme C qui demande à l'utilisateur le nombre de photocopies effectuées et qui affiche la facture correspondante.

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

int main() {
    int nb, facture;

    printf("Entrez le nombre de photocopies : ");
    scanf("%d", &nb);
    if(nb <= 0) {
        facture= 0;
    }
    else if(nb <= 10) {
        facture= nb*50;
    }
    else if(nb <= 30) {
        facture= 10*50 + (nb-10)*25;
    }
    else {
        facture= 10*50 + 20*25 + (nb-30)*10;
    }

    printf("Votre facture est de %d", facture);
    return 0;
}
```

I.2. La condition switch

La condition « **if... else** » que l'on vient de voir est le type de condition le plus souvent utilisé. En fait, il n'y a pas 36 façons de faire une condition en C. Le « **if... else** » permet de gérer tous les cas. Toutefois, le « **if... else** » peut s'avérer quelque peu… répétitif. Imaginez un instant que vous ayez à implémenter un programme qui nécessite prés de 25 conditions qui portent sur une même variable. Croyez moi dans ce cas de figure le « **if... else** » va devenir très vite ennuyeux.

En général, les informaticiens détestent faire des choses répétitives. Alors, pour éviter d'avoir à faire des répétitions quand on teste la valeur d'une seule et même variable, ils ont inventé une autre structure que le « **if... else** ». Cette structure particulière s'appelle « **switch** » donc la syntaxe est la suivante :

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

L'idée c'est donc d'écrire « **switch(<MaVariable>)** » pour dire « je vais tester la valeur de la variable **MaVariable** ». Vous ouvrez ensuite des accolades que vous refermez tout en bas. Ensuite, à l'intérieur de ces accolades, vous gérez tous les différents cas possibles : « **case <valeur 1>, case <valeur 2>, …** ».

Vous devez mettre une instruction « **break;** » obligatoirement à la fin de chaque cas. Si vous ne le faites pas, alors l'ordinateur ira lire les instructions en dessous censées être réservées aux autres cas ! L'instruction « **break;** » commande en fait à l'ordinateur de « sortir » des accolades.

Enfin, le cas « **default** » correspond en fait au « **else** ». Si la variable ne vaut aucune des valeurs précédentes, l'ordinateur ira lire le « **default** ».

Exemple :

Ecrire un programme C qui demande deux (02) nombres entiers non nuls au clavier et imprime le résultat de l'opération arithmétique réalisée entre les deux (02) nombres en fonction de l'opération choisie par l'utilisateur : addition, soustraction, multiplication et division.

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

int main() {
    int nb1, nb2, op;
    printf("Entrez deux (02) nombres entiers non nuls : ");
    scanf("%d %d", &nb1, &nb2);

    printf("Sélectionnez l'opération à réalisée : \n");
    printf("1. ADDITION \n");
    printf("2. SOUSTRACTION \n");
    printf("3. MULTIPLICATION \n");
    printf("4. DIVISION \n");
    printf("Quel est votre choix ? (1, 2, 3 ou 4) : ");
    scanf("%d", &op);

    switch(op) {
        case 1 :
            printf("\n %d + %d = %d \n", nb1, nb2, nb1+nb2);
            break;
        case 2 :
            printf("\n %d - %d = %d \n", nb1, nb2, nb1-nb2);
            break;
        case 3 :
            printf("\n %d x %d = %d \n", nb1, nb2, nb1*nb2);
            break;
        case 4 :
            printf("\n %d / %d = %f \n", nb1, nb2, (double) nb1/nb2);
            break;
        default :
            printf("\n Votre choix est invalide \n");
    }
    return 0;
}
```

I.3. Les ternaires : des conditions condensées

Il existe une troisième façon de faire des conditions, plus rare. On appelle cela des **expressions ternaires**. Concrètement, c'est comme un « **if... else** », sauf qu'on fait tout tenir sur une seule ligne.

Comme un exemple vaut mieux qu'un long discours, je vais vous donner deux fois une même condition : la première avec un « **if... else** », et la seconde, identique, mais sous forme d'une **expression ternaire**.

Supposons qu'on ait une variable booléenne « **majeur** » qui vaut **vrai (1)** si on est majeur, et **faux (0)** si on est mineur. On veut changer la valeur d'une variable « **age** » en fonction du booléen, pour mettre **"18"** si on est majeur, **"17"** si on est mineur. C'est un exemple complètement stupide je suis d'accord, mais ça me permet de vous montrer comment on peut se servir des expressions ternaires.

Avec « **if... else** » on aura :

```
if (majeur) {
      age= 18;
}
else {
      age= 17;
}
```

Alors qu'avec une **expression ternaire** on aura plutôt :

```
age = (majeur) ? 18 : 17;
```

Les **ternaires** permettent, sur une seule ligne, de changer la valeur d'une variable en fonction d'une condition. Ici la condition est tout simplement « **majeur** », mais ça pourrait être n'importe quelle condition plus longue bien entendu.

Le point d'interrogation permet de dire « **est-ce que tu es majeur ?** ». Si oui, alors on met la valeur **18** dans « **age** ». Sinon (le deux-points « **:** » signifie « **else** » ici), on met la valeur **17**.

Les ternaires ne sont pas du tout indispensables, personnellement je les utilise peu car ils peuvent rendre la lecture d'un code source un peu difficile. Ceci étant, il vaut mieux que vous les connaissiez pour le jour où vous tomberez sur un code plein de ternaires dans tous les sens !

II. Structures itératives

Il arrive souvent dans un programme qu'une même action soit répétée plusieurs fois, avec éventuellement quelques variations dans les paramètres qui précisent le déroulement de l'action. Il est alors fastidieux d'écrire un programme qui contient de nombreuses fois la même instruction. De plus, ce nombre peut dépendre du déroulement du programme. Il est alors impossible de savoir à l'avance combien de fois la même instruction doit être décrite. Pour gérer ces cas, on fait appel à des instructions en boucle qui ont pour effet de répéter plusieurs fois une même instruction.

Deux formes existent :

- La première, si le nombre de répétitions est connu avant l'exécution de l'instruction de répétition,
- La seconde s'il n'est pas connu.

Quelques définitions

- Une **boucle** permet de parcourir une partie d'un programme un certain nombre de fois.
- Une **itération** est la répétition d'un même traitement plusieurs fois.
- Un **indice de boucle** est une variable de boucle qui varie d'une valeur minimale (initiale) jusqu'à une valeur maximale (finale).

II.1. Répétitions inconditionnelles

Il est fréquent que le nombre de répétitions soit connu à l'avance, et que l'on ait besoin d'utiliser le numéro de l'itération afin d'effectuer des calculs ou des tests. Le mécanisme permettant cela est la boucle « **for** » qui signifie « **pour** » en français.

Syntaxe de la boucle :

```
for(<Initialisation>, <Condition>, <Incrémentation>){
     <liste d'instructions>;
}
```

- **<Initialisation>** : instruction de départ de la boucle
- **<Condition>** : condition de fin de boucle
- **<Incrémentation>** : expression d'évolution de la boucle

Exemple :

Ecrire un programme C qui demande un nombre de départ, et qui ensuite affiche les dix nombres suivants. Par exemple, si l'utilisateur entre le nombre 17, le programme affichera les nombres de 18 à 27.

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

int main() {
    int nb, i;
    printf("Entrez le nombre de depart : ");
    scanf("%d", &nb);

    for(i= 1; i<= 10; i++) {
        printf(" %d ", nb+i);
    }
    return 0;
}
```

II.2. Répétitions conditionnelles

II.2.1. La Boucle « while »

L'utilisation d'une boucle « **for** » nécessite de connaître à l'avance le nombre d'itérations désiré, c'est-à-dire la valeur finale du compteur. Dans beaucoup de cas, on souhaite répéter une instruction tant qu'une certaine condition est remplie, alors il est à priori impossible de savoir à l'avance au bout de combien d'itérations cette condition cessera d'être satisfaite. Le mécanisme permettant cela est la boucle « **while** » qui signifie « **tantque** » en français.

Syntaxe de la boucle :

```
while(<condition>) {
     <liste d'instructions>;
}
```

Cette boucle a une condition de poursuite dont la valeur est de type booléen et une liste d'instructions qui est répétée si la valeur de la condition de poursuite est vraie : la liste d'instructions est répétée autant de fois que la condition de poursuite à la valeur vraie.

Etant donné que la condition est évaluée avant l'exécution des instructions à répéter, il est possible que celles-ci ne soient jamais exécutées. Il faut que la liste des instructions ait une incidence sur la condition afin qu'elle puisse être évaluée à faux et que la boucle se termine. Il faut toujours s'assurer que la condition devienne fausse au bout d'un temps fini.

Exemple :

Un poissonnier sert un client qui a demandé 1Kg (= 1000g) de poisson. Il pèse successivement différents poissons et s'arrête dès que le poids total égale ou dépasse 1Kg. Ecrire un programme C qui imprime le nombre de poissons servis.

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

int main() {
    double poids_poisson= 0.0, poids_total= 0.0;
    int nb_poisson= 0;

    while(poids_total < 1000) { // poids en grammes (g)
        printf("Entrez le poids du poisson en gramme : ");
        scanf("%lf", &poids_poisson);
        poids_total= poids_total + poids_poisson;
        nb_poisson= nb_poisson + 1;
    }

    printf("Le nombre de poissons vendus est : %d", nb_poisson);
    return 0;
}
```

II.2.2. La Boucle « do .. while »

Ici, une instruction ou un groupe d'instructions sont exécutés répétitivement jusqu'à ce qu'une condition soit vérifiée. Avec la boucle « **while** », la condition doit d'abord être vérifiée avant exécution des instructions alors qu'avec la boule « **do .. while** » les instructions sont exécutées d'abord avant vérification de la condition.

Syntaxe de la boucle :

```
do {
     <liste d'instructions>;
} while(<condition>);
```

La vérification de la condition s'effectue après les instructions. Celles-ci sont donc exécutées au moins une fois.

Exemple :

Ecrire un programme qui demande successivement 10 nombres à l'utilisateur, et qui lui dit ensuite quel était le plus grand parmi ces 10 nombres.

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

int main() {
    int i= 1, nb= 0, plus_grand= 0;

    do {
        printf("Entrez le nombre numero %d : ", i);
        scanf("%d", &nb);

        if(nb > plus_grand) {
            plus_grand= nb;
        }

        i= i+1;
    }while(i <= 10);

    printf("Le plus grand nombre des 10 est : %d", plus_grand);
    return 0;
}
```

NB : Il y a une particularité dans la boucle « **do .. while** » qu'on a tendance à oublier quand on débute : il y a un point-virgule « **;** » tout à la fin ! N'oubliez pas d'en mettre un après le « **while(<condition>)** », sinon votre programme plantera à la compilation !

Tests de connaissances

1. Définir : Boucle, Itération.
2. Quelle différence faites-vous entre une structure conditionnelle et une structure itérative ?
3. Quelle est la syntaxe de déclaration d'une ternaire ?

CHAPITRE IV : LES FONCTIONS

Objectif du chapitre

Les fonctions en programmation C, comme dans la plupart des langages de programmation, facilitent la visibilité du code et surtout empêche la réécriture du même code plusieurs fois, permet de décomposer un problème **P** en sous problèmes **P1, P2……Pn** et surtout d'écrire plusieurs fois la même suite d'instruction. Il faut également mentionner que les fonctions faciliteront la mise au point et la maintenance des programmes (c'est-à-dire la possibilité de faire écrire par plusieurs personnes les différentes parties d'un programme).

Nous allons apprendre ici à structurer nos programmes en petits bouts à l'aide **des fonctions**.

Objectifs spécifiques

- Définition d'une fonction
- Déclaration et appel d'une fonction
- Mettre en exergue la différence entre fonction et procédure
- Notion de paramètres effectifs et formels
- Modes de transmission des paramètres
- Liste de quelques fonctions prédéfinies usuelles

I. Définition et manipulation des fonctions

Nous avons vu précédemment qu'un programme en C commençait par une fonction appelée « **main** ». Je vais vous faire un schéma récapitulatif, pour vous rappeler quelques mots de vocabulaire.

En haut, on y trouve les **directives de préprocesseur**. Ces directives sont faciles à identifier : elles commencent par un « # » et sont généralement mises tout en haut des fichiers sources.

Puis en dessous, on n'a ce que j'avais déjà appelé plus haut « **une fonction** ». Ici, sur le schéma, vous voyez une fonction « **main** ».

Je vous avais dit qu'un programme en langage C commençait par la fonction « **main** ». Je vous rassure, c'est toujours vrai ! Seulement, jusqu'ici nous sommes restés à l'intérieur de la fonction « **main** ». Nous n'en sommes jamais sortis. Revoyez vos codes sources et vous verrez : nous sommes toujours restés à l'intérieur des accolades de la fonction « **main** ».

Ce n'est pas « mal », mais ce n'est pas ce que les programmeurs en C font dans la réalité. Quasiment aucun programme n'est écrit uniquement à l'intérieur des accolades de la fonction « **main** ». Jusqu'ici nos programmes étaient courts, donc ça ne posait pas de gros problèmes, mais imaginez des plus gros programmes qui font des milliers de lignes de code ! Si tout était concentré dans la fonction « main ».

I.1. Déclarer et appeler une fonction

Nous allons donc maintenant apprendre à nous organiser. Nous allons en fait découper nos programmes en petits bouts. Chaque « petit bout de programme » sera ce qu'on appelle **une fonction**.

Une **fonction** exécute des actions et renvoie un résultat. C'est un morceau de code qui sert à faire quelque chose de précis. On dit qu'une fonction possède **une entrée et une sortie**.

Lorsqu'on appelle une fonction, il y a trois (03) étapes :

1. **L'entrée** : on fait « rentrer » des informations dans la fonction (en lui donnant des informations avec lesquelles travailler).
2. **Les calculs** : grâce aux informations qu'elle a reçues en entrée, la fonction travaille.
3. **La sortie** : une fois qu'elle a fini ses calculs, la fonction renvoie un résultat. C'est ce qu'on appelle la sortie, ou encore le retour.

Concrètement, on peut imaginer par exemple une fonction appelée « **triple** » qui calcule le triple du nombre qu'on lui donne, en le multipliant par 3.

La fonction « triple » multiplie le nombre en entrée par 3

Le but des fonctions est donc de simplifier le code source, pour ne pas avoir à retaper le même code plusieurs fois d'affilée.

Syntaxe de déclaration d'une fonction

```
type identificateur(<liste des paramètres>) {
    <Liste des instructions>;
}
```

- **type (correspond à la sortie)** : c'est le type de la fonction (encore appelé **type de retour**). Comme les variables, les fonctions ont un type. Ce type dépend du résultat que la fonction renvoie : si la fonction renvoie un nombre décimal, vous mettrez sûrement « **double** », si elle renvoie un entier vous mettrez « **int** » ou « **long** » par exemple. Mais il est aussi possible de créer des fonctions qui ne renvoient rien !

  Il y a donc **deux (02) sortes de fonctions** :
  - **Les fonctions qui renvoient une valeur** : on leur met un des types que l'on connaît (**char, int, double, etc.**). La particularité de ces fonctions est qu'elles disposent toujours dans la **<Liste des instructions>**, une instruction « **return <valeur de retour>** » placé en dernier permettant de renvoyer une valeur en fonction du type de retour ;
  - **Les fonctions qui ne renvoient pas de valeur (qui correspondent aux procédures en algorithmique)** : on leur met un type de retour spécial « **void** » (qui signifie « vide »). Ces fonctions n'ont pas d'instruction « **return <valeur de retour>** ».

- **identificateur** : c'est le nom de votre fonction. Vous pouvez appeler votre fonction comme vous voulez, du temps que vous respectez les mêmes règles que pour les variables (pas d'accents, pas d'espaces, etc.).
- **<liste des paramètres> (correspond à l'entrée)** : entre parenthèses, vous pouvez envoyer des paramètres à la fonction. Ce sont des valeurs avec lesquelles la fonction va travailler.
  ***NB** : Vous pouvez envoyer autant de paramètres que vous le voulez. Vous pouvez aussi n'envoyer aucun paramètre à la fonction.*
- Ensuite vous avez les accolades qui indiquent le début et la fin de la fonction. À l'intérieur de ces accolades vous mettrez les instructions que vous voulez.

Appel d'une fonction

```
identificateur(<liste des paramètres>);
```

L'appel d'une fonction s'effectue dans le « **main** ». Il est possible également d'effectuer l'appel d'une fonction dans une autre fonction mais tant que celle-ci n'est pas appeler dans le « **main** », elle ne s'exécutera jamais.

Le type de retour et la liste des paramètres lors de l'appel de la fonction dépendent de la syntaxe de déclaration utilisée.

Exemple :

Ecrire un programme C qui permet de calculer la somme de deux nombres entiers naturels à l'aide d'une fonction.

Solution à l'aide d'une fonction qui ne renvoie pas de valeur

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

void addition(int a, int b) {
    printf("\n %d + %d = %d \n", a, b, a+b);
}

int main() {
    int x, y;
    printf("Entrez deux (02) nombres entiers naturels : ");
    scanf("%d %d", &x, &y);

    addition(x, y);
    return 0;
}
```

Solution à l'aide d'une fonction qui renvoie une valeur

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

int addition(int a, int b) {
    return a+b;
}

int main() {
    int x, y;
    printf("Entrez deux (02) nombres entiers naturels : ");
    scanf("%d %d", &x, &y);

    printf("\n %d + %d = %d \n", x, y, addition(x, y));
    return 0;
}
```

I.2. Notion de paramètres formels et paramètres effectifs

Les **paramètres formels (fictifs)** : ce sont des paramètres qui figurent dans l'entête de la déclaration d'une fonction et sont utilisés dans les instructions de la fonction et la fonction seulement. Ils correspondent à des variables locales.

Les **paramètres effectifs (réels)** : ce sont des paramètres qui figurent dans l'instruction d'appel de la fonction et sont substitués aux paramètres formels au moment de l'appel de la fonction.

Si nous reprenons l'exemple précédent portant sur la somme de deux nombres, on aura :

Paramètres formels	Paramètres effectifs
a, b	x, y

I.3. Modes de transmission des paramètres

Il existe deux modes de transmission des paramètres à une fonction :

Par valeur

Le paramètre transmis n'est jamais affecté par les modifications dans la fonction (on ne récupère pas les résultats !). Le passage de paramètres par valeur permet au programme appelant de transmettre une valeur à la fonction appelée.
Avant d'exécuter les instructions de la fonction la valeur du paramètre effectif est affectée au paramètre formel. Dans ce cas, le paramètre formel est considéré comme une variable locale qui est initialisée lors de l'appel.
Ainsi,

- Les paramètres effectifs et les paramètres formels doivent s'accorder du point de vue nombre et ordre.
- Leurs types doivent être identiques selon le mode de passage des paramètres.

Remarque :

- Le transfert d'information est effectué dans un seul sens, du programme principal vers la procédure.
- Toute modification du paramètre formel est sans conséquence sur le paramètre effectif.

Par variable ou adresse ou référence

Le paramètre transmis dans ce cas peut être modifié et on récupère les résultats. Le passage de paramètres par variable permet au programme appelant de transmettre une valeur à la fonction appelée et vice-versa.
Pendant l'exécution des instructions de la procédure, le paramètre formel fait référence au même contenu de la variable que celui désigné par le paramètre effectif.

Remarque :

- Le transfert d'information est effectué dans les deux sens, du programme principal vers la fonction et vice-versa.
- Toute modification du paramètre formel entraîne automatiquement la modification de la valeur du paramètre effectif.

Illustration de la différence entre passage par valeur et passage par adresse

On suppose ici qu'on n'a deux (02) variables entières **x qui vaut 10, et y qui vaut 20** et on voudrait permuter les deux variables de tel sorte que **x vaut 20 et y vaut 10**.

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

// Passage par valeur
void permut1(int a, int b) {
    int tmp= a;
    a= b;
    b= tmp;
}

// Passage par adresse
void permut2(int *a, int *b) {
    int tmp= *a;
    *a= *b;
    *b= tmp;
}

int main() {
    int x= 10, y= 20;

    printf("\n Passage par valeur \n");
    permut1(x, y);
    printf("x= %d, y= %d", x, y); // affiche : x= 10, y= 20

    printf("\n Passage par adresse \n");
    permut2(&x, &y);
    printf("x= %d, y= %d", x, y); // affiche : x= 20, y= 10

    return 0;
}
```

II. Quelques fonctions prédéfinies usuelles

En langage C, il existe ce qu'on appelle des **bibliothèques « standard »**, c'est-à-dire des bibliothèques toujours utilisables. Ce sont en quelque sorte des bibliothèques « de base » qu'on utilise très souvent.

Les bibliothèques sont, je vous le rappelle, des ensembles de fonctions toutes prêtes. Ces fonctions ont été écrites par des programmeurs avant vous, elles vous évitent en quelque sorte d'avoir à réinventer la roue à chaque nouveau programme.
Vous avez déjà utilisé les fonctions « **printf** » et « **scanf** » de la bibliothèque « **stdio.h** ».

Il faut savoir qu'il existe d'autres bibliothèques qui contiennent de nombreuses fonctions toutes prêtes.

Nous apprendrons ici quelques fonctions très utilisées par les programmeurs issues de diverses bibliothèques.

II.1. La bibliothèque « stdlib.h »

Cette bibliothèque renferme les routines[1] standards.

Procédure ou Fonction	Prototype	Description
Abort	void **abort**(void)	Cette procédure permet d'interrompre l'exécution du programme de façon anormale.
Abs	int **abs**(int a)	Cette fonction retourne la valeur absolue.
Atof	double **atof**(const char *str)	Cette fonction convertie une chaîne de caractères en une valeur « **float** ».
Atoi	int **atoi**(const char *str)	Cette fonction convertie une chaîne de caractères en une valeur entière « **int** ».
Atol	long **atol**(const char *str)	Cette fonction convertie une chaîne de caractères en une valeur entière « **long** ».
Exit	void **exit**(int etat)	Cette procédure met fin à l'exécution d'un programme avec une valeur de retour.
Getenv	int **getenv**(const char *vnom)	Cette fonction permet d'effectuer la lecture d'une variable d'environnement système.
Malloc	void ***malloc**(size_t n)	Cette fonction permet une allocation de mémoire dynamique de « *n* » octets.
Rand	int **rand**(void)	Cette fonction retourne un nombre pseudo-aléatoire entier.
System	int **system**(const char *commande)	Cette fonction permet de lancer une commande dans le système d'exploitation.

Pour en savoir plus sur les fonctions liées à cette bibliothèque voici un lien que peu vous être utile :
https://www.gladir.com/CODER/C/referencestdlib.htm

[1] **Une routine** est une entité informatique qui encapsule une portion de code (une séquence d'instructions) effectuant un traitement spécifique bien identifié (asservissement, tâche, calcul, etc.) relativement indépendant du reste du programme, et qui peut être réutilisé dans le même programme, ou dans un autre. Dans ce cas, on range souvent la routine dans une bibliothèque pour la rendre disponible à d'autres projets de programmation, tout en préservant l'intégrité de son implémentation

II.2. La bibliothèque mathématique « math.h »

Cette bibliothèque renferme les routines de traitement mathématique.

Pour pouvoir utiliser les fonctions de la bibliothèque mathématique, il est indispensable de mettre la directive de préprocesseur suivante en haut de votre programme :

```
#include <math.h>
```

Une fois que c'est fait, vous pouvez utiliser toutes les fonctions de cette bibliothèque.

Procédure ou Fonction	Prototype	Description
Acos	double **acos**(double a)	Cette fonction trigonométrique retourne l'«ArcCosinus».
Asin	double **asin**(double a)	Cette fonction trigonométrique retourne l'«ArcSinus».
Atan	double **atan**(double a)	Cette fonction trigonométrique retourne l'«ArcTangente».
Ceil	double **ceil**(double a)	Cette fonction retourne la valeur maximale d'un nombre, soit l'entier le plus proche supérieur ou égal au nombre. (en gros, il permet d'arrondi)
Cos	double **cos**(double a)	Cette fonction trigonométrique retourne le «Cosinus».
Cosh	double **cosh**(double a)	Cette fonction trigonométrique retourne le «Cosinus» hyperbolique.
Exp	double **exp**(double x)	Cette fonction calcul l'exponentiel de la valeur «x».
Fabs	double **fabs**(double a)	Cette fonction calcul la valeur absolue d'un nombre réel.
Floor	double **floor**(double a)	Cette fonction retourne la valeur minimale d'un nombre, soit l'entier le plus proche inférieur ou égal au nombre.
Fmod	double **fmod**(double a,double b)	Cette fonction retourne le reste d'une division de a/b.
Labs	long **labs**(long a)	Cette fonction retourne la valeur absolue d'un entier de type «long».
Log	double **log**(double a)	Cette fonction retourne le logarithme naturel ou népérien.
log10	double **log10**(double a)	Cette fonction retourne le logarithme décimal.
Pow	double **pow**(double x,double y)	Cette fonction retourne le calcul de x à la puissance y.
Sin	double **sin**(double x)	Cette fonction trigonométrique retourne le «Sinus».
Sinh	double **sinh**(double a)	Cette fonction trigonométrique retourne le «Sinus» hyperbolique.
Sqrt	double **sqrt**(double a)	Cette fonction retourne la racine carrée du nombre «a».
Tan	double **tan**(double x)	Cette fonction trigonométrique retourne la «tangente».
Tanh	double **tanh**(double a)	Cette fonction trigonométrique retourne la «tangente» hyperbolique.

Pour en savoir plus sur les fonctions liées à cette bibliothèque voici un lien que peu vous être utile :
https://www.gladir.com/CODER/C/referencemath.htm

Tests de connaissances

1. Définir : Bibliothèque, fonction, procédure.
2. A quoi servent les bibliothèques « stdio.h », « stdlib.h » et « math.h » ? Citer quelques fonctions dans ces bibliothèques.
3. Dans quel cas l'instruction « **return** » n'est pas obligatoire ?
4. Quelle différence faites-vous entre passage par valeur et passage par adresse ?

PARTIE II : TECHNIQUES AVANCEES DU LANGAGE C

OBJECTIF DE LA PARTIE

Dans cette partie, nous allons découvrir des concepts plus avancés du langage C. Lorsque vous serez arrivés à la fin de cette partie, vous serez capables de vous débrouiller dans la plupart des programmes écrits en C.

OBJECTIFS SPECIFIQUES

Au terme de cette partie, l'apprenant sera capable de définir ou utiliser quelques concepts avancés du langage C à savoir :

- La programmation modulaire
- Les pointeurs
- Les tableaux
- Les chaînes de caractères
- Le préprocesseur
- Les types composés
- Lire et écrire dans des fichiers

CHAPITRE V : PROGRAMMATION MODULAIRE

Objectif du chapitre

Mieux structuré le code source de votre programme.

Objectifs spécifiques

- Description du principe de fonctionnement de la programmation modulaire
- Définition et manipulation des prototypes et des headers
- Description du principe de fonctionnement de la compilation (compilation séparée)
- La portée des fonctions et des variables

I. Définition et principe

Jusqu'ici nous n'avons travaillé que dans un seul fichier appelé « **main.c** ». Pour le moment c'était acceptable car nos programmes étaient tout petits, mais ils vont bientôt être composés de dizaines, que dis-je de centaines de fonctions, et si vous les mettez toutes dans un même fichier celui-là va finir par devenir très long !

C'est pour cela que l'on a inventé ce qu'on appelle la **programmation modulaire**. Le principe est tout bête : plutôt que de placer tout le code de notre programme dans un seul fichier « **main.c** », nous le « séparons » en plusieurs petits fichiers.

I.1. Les prototypes

Depuis le chapitre précédent, nous placions nos fonctions avant la fonction « **main** ». Pourquoi ?

Parce que l'ordre a une réelle importance ici : si vous mettez votre fonction avant le « **main** » dans votre code source, votre ordinateur l'aura lue et la connaîtra. Lorsque vous ferez un appel à la fonction dans le « **main** », l'ordinateur connaîtra la fonction et saura où aller la chercher.

En revanche, si vous mettez votre fonction après le « **main** », ça ne marchera pas car l'ordinateur ne connaîtra pas encore la fonction.

Ici, nous allons donc apprendre comment positionner nos fonctions dans n'importe quel ordre dans le code source. Pour cela, nous allons annoncer nos fonctions à l'ordinateur en écrivant ce qu'on appelle « **des prototypes** ».

Dans le souci de faciliter la compréhension des prototypes, prenons un exemple : Supposons qu'il nous ai demandé d'écrire un programme C qui permet de calculer l'aire d'un rectangle sachant que l'aire du rectangle est évaluée à l'aide d'une fonction qui prend en paramètre la longueur et la largeur du rectangle et retourne l'aire. On peut avoir le code C suivant :

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

/*
La ligne suivante est le prototype de la fonction aireRectangle
:
*/
double aireRectangle(double longueur, double largeur);

int main() {
    double longueur, largeur;
    printf("Entrez la longueur et la largeur du Rectangle : ");
    scanf("%lf %lf", &longueur, &largeur);

    printf("Aire = %f \n", aireRectangle(longueur, largeur));
    return 0;
}

/*
Notre fonction aireRectangle peut maintenant être mise n'importe
où dans le code source :
*/
double aireRectangle(double longueur, double largeur) {
    return longueur * largeur;
}
```

Si vous regardez attentivement le code, vous verrez que ce qui a changé ici, c'est l'ajout du **prototype** en haut du code source.

Un prototype, c'est en fait une indication pour l'ordinateur. Cela lui indique qu'il existe une fonction appelée « **aireRectangle** » qui prend tels paramètres en entrée et renvoie une sortie du type que vous indiquez. Cela permet à l'ordinateur de s'organiser.

Grâce à cette ligne, vous pouvez maintenant placer vos fonctions dans n'importe quel ordre sans vous prendre la tête !

Comme vous le voyez, la fonction « **main** » n'a pas de prototype. En fait, c'est la seule qui n'en nécessite pas, parce que l'ordinateur la connaît.

Pour être tout à fait exact, il faut savoir que dans la ligne du prototype il est facultatif d'écrire les noms de variables en entrée. L'ordinateur a juste besoin de connaître les types des variables.

On aurait donc pu simplement écrire :

```
double aireRectangle(double, double);
```

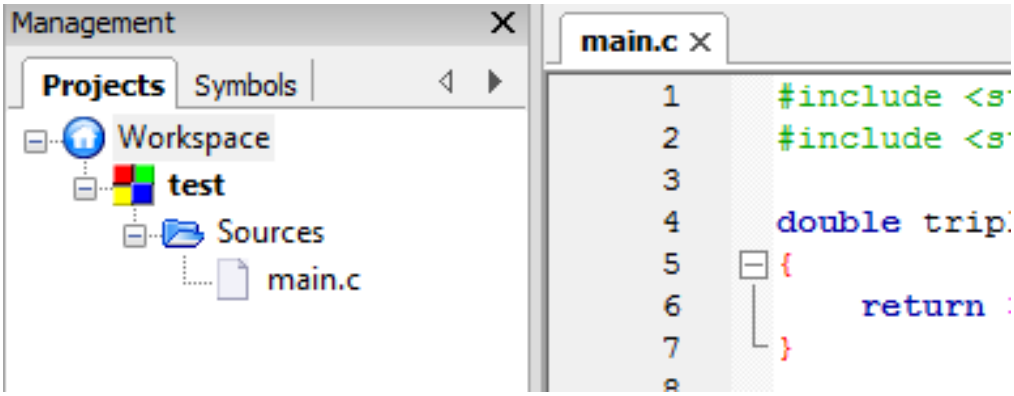
Toutefois, l'autre méthode vu tout à l'heure fonctionne aussi bien. L'avantage avec cette méthode, c'est que vous avez juste besoin de copier-coller la première ligne de la fonction et de rajouter un point-virgule.

NB : N'oubliez JAMAIS de mettre un point-virgule à la fin d'un prototype. C'est ce qui permet à l'ordinateur de différencier un prototype du véritable début d'une fonction. Si vous ne le faites pas, vous risquez d'avoir des erreurs incompréhensibles lors de la compilation.

I.2. Les headers

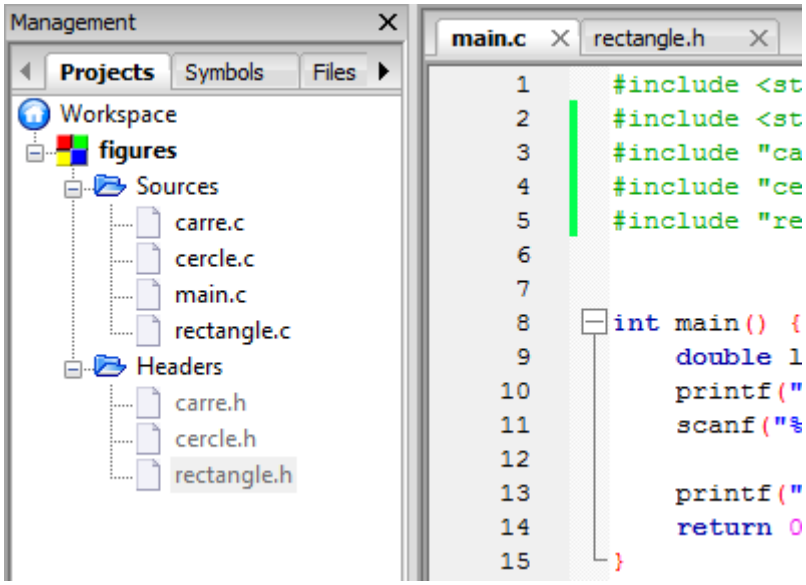
Jusqu'ici, nous n'avions qu'un seul fichier source dans notre projet, le « **main.c** ». Dans la pratique, vos programmes ne seront pas tous écrits dans ce même fichier. Bien sûr, il est possible de le faire, mais ce n'est jamais très pratique de se balader dans un fichier de 10 000 lignes. C'est pour cela qu'en général on crée plusieurs fichiers par projet.

Un projet, c'est l'ensemble des fichiers source de votre programme. Pour le moment, nos projets n'étaient composés que d'un fichier source. Regardez dans votre IDE, généralement c'est sur la gauche.



Comme vous pouvez le voir à gauche sur cette capture d'écran, ce projet n'est composé que d'un fichier « **main.c** ».

Laissez-moi maintenant vous montrer un vrai projet.



Comme vous le voyez, il y a plusieurs fichiers. Un vrai projet ressemblera à ça : vous verrez plusieurs fichiers dans la colonne de gauche. Vous reconnaissez dans la liste le fichier « **main.c** » : c'est celui qui contient la fonction « **main** ». En général dans mes programmes, je ne mets que le « **main** » dans « **main.c** ». Pour information, ce n'est pas du tout une obligation, chacun s'organise comme il veut. Pour bien me suivre, je vous conseille néanmoins de faire comme moi.

Comment savoir combien de fichiers créer pour son projet ?

C'est vous qui choisissez. En général, on regroupe dans un même fichier des fonctions ayant le même thème. Ainsi, dans le fichier « **carre.c** » on peut regrouper toutes les fonctions concernant un carré (aire, périmètre, …) ; dans le fichier « **cercle.c** », on peut regrouper toutes les fonctions concernant un cercle, etc.

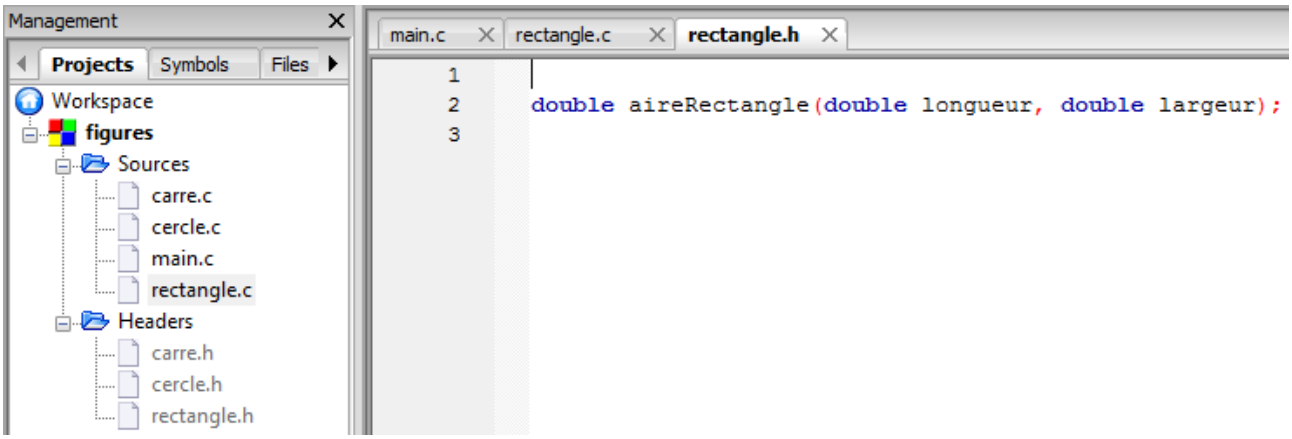
Les fichiers « .h » et « .c »

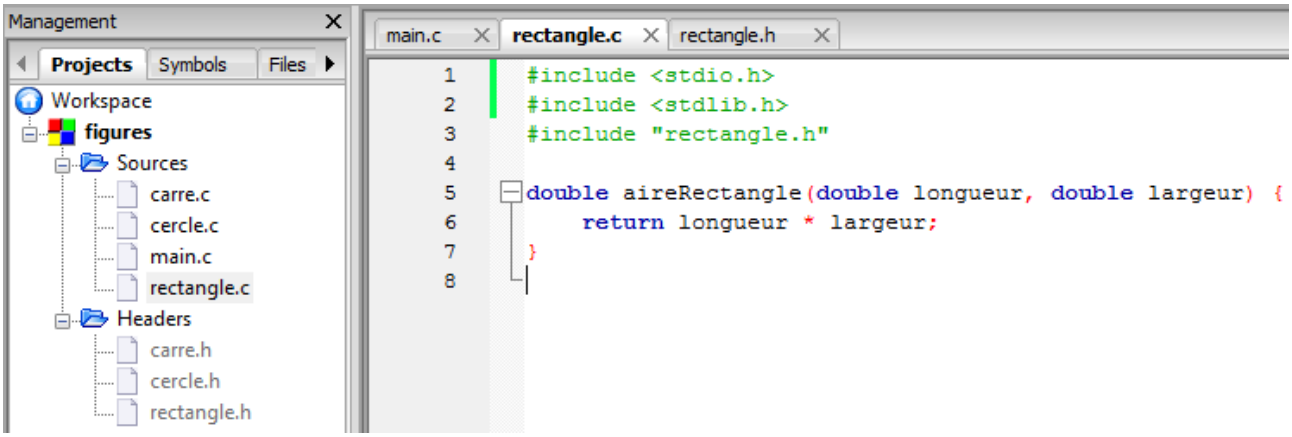
Comme vous le voyez, il y a deux types de fichiers différents sur la figure.

- Les « **.h** », appelés **fichiers headers**. Ces fichiers contiennent les prototypes des fonctions.
- Les « **.c** » : appelés **fichiers sources**. Ces fichiers contiennent les fonctions elles-mêmes.

En général, on met donc rarement les prototypes dans **les fichiers « .c »** comme on l'a fait tout à l'heure dans le « **main.c** » (sauf si votre programme est tout petit). Donc, pour chaque « **fichier.c** », on aura son équivalent « **.h** » qui contient les prototypes des fonctions.

Pour que l'ordinateur sache que les prototypes sont dans un autre fichier que le « **.c** », Il faut inclure le fichier « **.h** » grâce à la directive de préprocesseur « **#include** » (voir les figures suivantes).





On n'a inclut trois fichiers « **.h** » : « **stdio** », « **stdlib** » et « **rectangle** ».
Notez une différence : les fichiers que vous avez créés et placés dans le répertoire de votre projet doivent être inclus avec des guillemets (**"rectangle.h"**) tandis que les fichiers correspondant aux bibliothèques (qui sont généralement installés, eux, dans le répertoire de votre IDE) sont inclus entre chevrons (**<stdio.h>**).

Vous utiliserez donc :

- **Les chevrons < >** pour inclure un fichier se trouvant dans le répertoire « **include** » de votre IDE (généralement **C:\Program Files\CodeBlocks\MinGW\include**) ;
- **Les guillemets " "** pour inclure un fichier se trouvant dans le répertoire de votre projet.

La commande « **#include** » demande d'insérer le contenu du fichier dans le « **.c** ». C'est donc une commande qui dit « **Insère ici le fichier rectangle.h** » par exemple.

Dans le fichier « **rectangle.h** », on trouve simplement les prototypes des fonctions du fichier « **rectangle.c** ».

Quel est l'intérêt de mettre les prototypes dans des fichiers « .h » ?

La raison est en fait assez simple. Quand dans votre code vous faites appel à une fonction, votre ordinateur doit déjà la connaître, savoir combien de paramètres elle prend, etc. C'est à ça que sert un prototype : c'est le mode d'emploi de la fonction pour l'ordinateur.

Tout est une question d'ordre : si vous placez vos prototypes dans des « **.h** » **(headers)** inclus en haut des fichiers « **.c** », votre ordinateur connaîtra le mode d'emploi de toutes vos fonctions dès le début de la lecture du fichier.

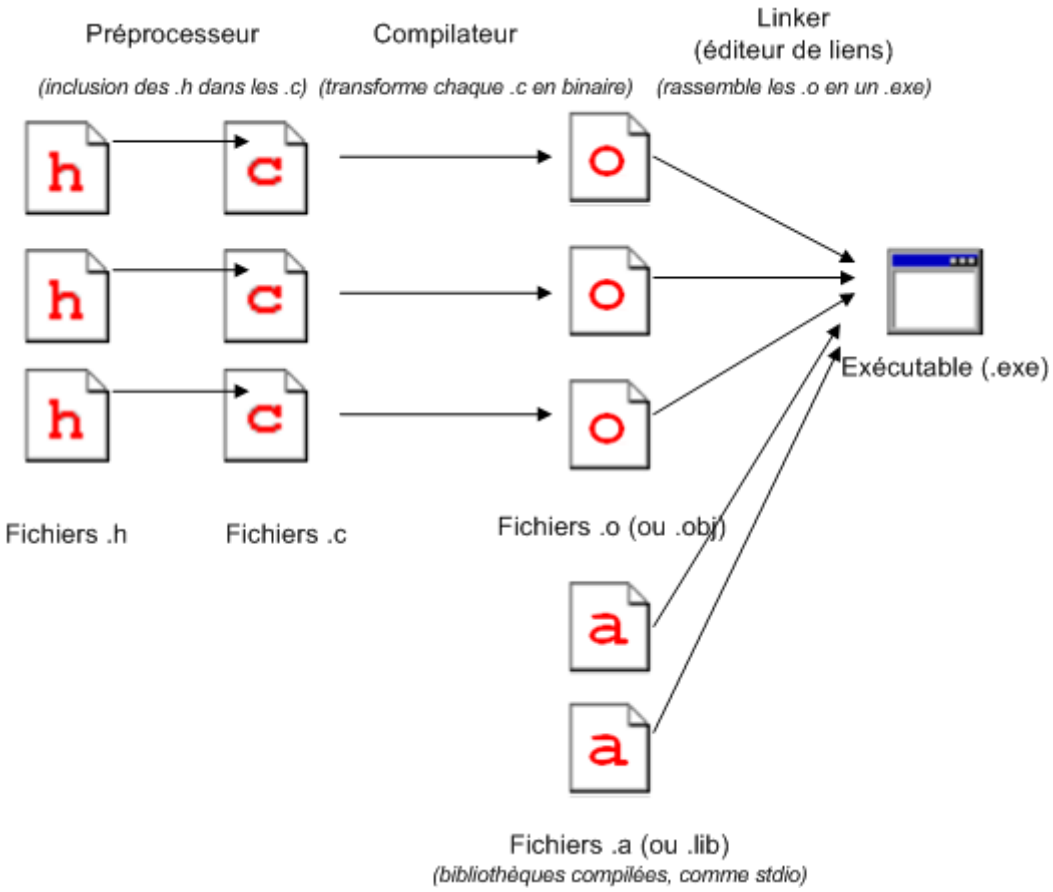
En faisant cela, vous n'aurez ainsi pas à vous soucier de l'ordre dans lequel les fonctions se trouvent dans vos fichiers « **.c** ». Si maintenant vous faites un petit programme contenant deux ou trois fonctions, vous vous rendrez peut-être compte que les prototypes semblent facultatifs. Mais ça ne durera pas longtemps ! Dès que vous aurez un peu plus de fonctions, si vous ne mettez pas vos prototypes de fonctions dans des « **.h** », la compilation échouera sans aucun doute.

Attention : Lorsque vous appellerez une fonction située dans « **rectangle.c** » depuis le fichier « **main.c** », vous aurez besoin d'inclure les prototypes de « **rectangle.c** » dans « **main.c** ». Il faudra donc mettre un « **#include "rectangle.h"** » en haut du « **main.c** ».

Souvenez-vous de cette règle : à chaque fois que vous faites appel à une fonction X dans un fichier, il faut que vous ayez inclus les prototypes de cette fonction dans votre fichier. Cela permet au compilateur de vérifier si vous l'avez correctement appelée.

II. Compilation séparée

Maintenant que vous savez qu'un projet est composé de plusieurs fichiers source, nous pouvons rentrer plus en détail dans le fonctionnement de la compilation.
La figure suivante est un schéma bien plus précis de la compilation. C'est le genre de schéma qu'il est fortement conseillé de comprendre et de connaître par cœur !



1. **Préprocesseur** : le préprocesseur est un programme qui démarre avant la compilation. Son rôle est d'exécuter les instructions spéciales qu'on lui a données dans des directives de préprocesseur, ces fameuses lignes qui commencent par un « # ».

   Prenons par exemple la directive de préprocesseur que l'on utilise généralement « **#include** », qui permet d'inclure un fichier dans un autre. Le préprocesseur « remplace » donc les lignes « **#include** » par le fichier indiqué. Il met à l'intérieur de chaque fichier « **.c** » le contenu des fichiers « **.h** » qu'on a demandé d'inclure. À ce moment-là de la compilation, votre fichier « **.c** » est complet et contient tous les prototypes des fonctions que vous utilisez.

2. **Compilation** : cette étape très importante consiste à transformer vos fichiers source en code binaire compréhensible par l'ordinateur. Le compilateur compile chaque fichier « **.c** » un à un. Il compile tous les fichiers source de votre projet, d'où l'importance d'avoir bien ajouté tous vos fichiers au projet.

   Le compilateur génère un fichier « **.o** » (ou « **.obj** », ça dépend du compilateur) par fichier « **.c** » compilé : ce sont des **fichiers binaires temporaires**. Généralement, ces fichiers sont supprimés à la fin de la compilation, mais selon les options de votre IDE, vous pouvez choisir de les conserver. Bien qu'inutiles puisque temporaires, on peut trouver un intérêt à conserver les « **.o** ». En effet, si parmi les 10 fichiers « **.c** » de votre projet seul l'un d'eux a

changé depuis la dernière compilation, le compilateur n'aura qu'à recompiler seulement ce fichier « **.c** ». Pour les autres, il possède déjà les « **.o** » compilés.

3. **Édition de liens : le linker (ou « éditeur de liens » en français)** est un programme dont le rôle est d'assembler les fichiers binaires « **.o** » avec les bibliothèques compilées dont vous avez besoin (« **.a** » ou « **.lib** » selon le compilateur). Il les assemble en un seul gros fichier : l'exécutable final ! Cet exécutable a **l'extension « .exe » sous Windows**. Si vous êtes sous un autre OS, il devrait prendre l'extension adéquate.

Je le dis et je le répète, ce schéma (figure précédente) est très important. Il fait la différence entre un programmeur du dimanche qui copie sans comprendre des codes source et un autre qui sait et comprend ce qu'il fait.

III. La portée des fonctions et variables

Pour clore ce chapitre, il nous faut impérativement découvrir la notion de portée des fonctions et des variables. Nous allons voir quand les variables et les fonctions sont accessibles, c'est-à-dire quand on peut faire appel à elles.

III.1. Les variables propres aux fonctions (variables locales)

Lorsque vous déclarez une variable dans une fonction, celle-ci est supprimée de la mémoire à la fin de la fonction. Une variable déclarée dans une fonction n'existe donc que pendant que la fonction est exécutée. C'est-à-dire que vous ne pouvez pas y accéder depuis une autre fonction.

Ainsi le code suivant permettant de faire l'addition de deux nombres, ne marchera jamais :

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

void addition(int a, int b) {
    int som= a+b;
}

int main() {
    int x, y;
    printf("Entrez deux (02) nombres entiers naturels : ");
    scanf("%d %d", &x, &y);

    addition(x, y);

    printf("\n %d + %d = %d \n", x, y, som);
    return 0;
}
```

Car la variable « **som** » a été déclarée dans la fonction « **addition** ».

Donc en conclusion, une variable déclarée dans une fonction n'est accessible qu'à l'intérieur de cette fonction. On dit que c'est une **variable locale**.

III.2. Les variables globales

III.2.1. Variable globale accessible dans tous les fichiers

Il est possible de déclarer des variables qui seront accessibles dans toutes les fonctions de tous les fichiers du projet. Je vais vous montrer comment faire pour que vous sachiez que ça existe, mais généralement il faut éviter de le faire. Ça aura l'air de simplifier votre code au début, mais ensuite vous risquez de vous retrouver avec de nombreuses variables accessibles partout, ce qui risquera de vous créer des soucis.

Pour déclarer une variable globale accessible partout, vous devez faire la déclaration de la variable en dehors des fonctions. Vous ferez généralement la déclaration tout en haut du fichier, après les « **#include** ».

Si nous reprenons le code précédent portant sur l'addition de deux nombres on aura :

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

int som= 0;

void addition(int a, int b) {
    som= a+b;
}

int main() {
    int x, y;
    printf("Entrez deux (02) nombres entiers naturels : ");
    scanf("%d %d", &x, &y);

    addition(x, y);

    printf("\n %d + %d = %d \n", x, y, som);
    return 0;
}
```

La variable « **som** » sera accessible dans tous les fichiers du projet, on pourra donc faire appel à elle dans TOUTES les fonctions du programme.

Attention : Ce type de choses est généralement à bannir dans un programme en C. Utilisez plutôt le retour de la fonction (return) pour renvoyer un résultat.

III.2.2. Variable globale accessible uniquement dans un fichier

La variable globale que nous venons de voir était accessible dans tous les fichiers du projet. Il est possible de la rendre accessible uniquement dans le fichier dans lequel elle se trouve. Ça reste une variable globale quand même, mais disons qu'elle n'est globale qu'aux fonctions de ce fichier, et non à toutes les fonctions du programme.

Pour créer une variable globale accessible uniquement dans un fichier, rajoutez simplement le mot-clé « **static** » devant la déclaration de la variable.

Exemple :

```
static int som
```

III.3. Variable statique à une fonction

Attention c'est un peu plus délicat, ici. Si vous rajoutez le mot-clé « **static** » devant la déclaration d'une variable à l'intérieur d'une fonction, ça n'a pas le même sens que pour les variables globales. En fait, la variable « **static** » n'est plus supprimée à la fin de la fonction. La prochaine fois qu'on appellera la fonction, la variable aura conservé sa valeur. Cela signifie qu'on pourra rappeler la fonction plus tard et la variable contiendra toujours la valeur de la dernière fois.

Voici un petit exemple pour bien comprendre :

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

int increment() {
    static int nombre= 0;
    nombre++;
    return nombre;
}

int main() {
    printf("%d \n", increment()); // affiche 1
    printf("%d \n", increment()); // affiche 2
    printf("%d \n", increment()); // affiche 3
    printf("%d \n", increment()); // affiche 4

    return 0;
}
```

Ici, la première fois qu'on appelle la fonction « **increment** », la variable « **nombre** » est créée. Elle est incrémentée à **1**, et une fois la fonction terminée la variable n'est pas supprimée. Lorsque la fonction est appelée une seconde fois, la ligne de la déclaration de variable est tout simplement « **sautée** ». On ne recrée pas la variable, on réutilise la variable qu'on avait déjà créée.

Comme la variable valait 1, elle vaudra maintenant 2, puis 3, puis 4, etc.

III.4. Les fonctions locales à un fichier

Pour en finir avec les portées, nous allons nous intéresser à la portée des fonctions. Normalement, quand vous créez une fonction, celle-ci est globale à tout le programme. Elle est accessible depuis n'importe quel autre fichier « **.c** ».

Il se peut que vous ayez besoin de créer des fonctions qui ne seront accessibles que dans le fichier dans lequel se trouve la fonction.

Pour faire cela, rajoutez le mot-clé « **static** » devant la déclaration de la fonction :

Exemple :

```
static int addition(int a, int b){
    // Instructions
}
```

Tests de connaissances

1. Définir : Programmation modulaire, prototype, header, préprocesseur, compilateur, linker, variable globale, variable locale
2. Expliquer brièvement le principe de la programmation modulaire
3. Expliquer brièvement le principe de fonctionnement de la compilation d'un programme C
4. Quelle différence faites-vous entre variable propre à une fonction et variable statique à une fonction ?
5. Quand dit-t-on qu'une fonction est statique ?
6. A quoi renvoi les extensions suivantes : « .c », « .o », « .obj », « .a », « .lib », « .exe », « .out ».

CHAPITRE VI : LES POINTEURS

Objectif du chapitre

L'heure est venue pour vous de découvrir les pointeurs. Prenez un grand bol d'air avant car ce chapitre ne sera probablement pas une partie de plaisir. Les pointeurs représentent en effet une des notions les plus délicates du langage C. Si j'insiste autant sur leur importance, c'est parce qu'il est impossible de programmer en langage C sans les connaître et bien les comprendre. Les pointeurs sont omniprésents, nous les avons d'ailleurs déjà utilisés sans le savoir.

Nombre de ceux qui apprennent le langage C titubent en général sur les pointeurs. Nous allons faire en sorte que ce ne soit pas votre cas. Ainsi au terme de ce chapitre, nous allons apprendre à définir et manipuler les pointeurs.

Objectifs spécifiques

- Création des pointeurs ;
- Utilisation des pointeurs ;
- Mode de transmission des pointeurs.

I. Définition et utilisation des pointeurs

Un des plus gros problèmes avec les pointeurs, en plus d'être assez délicats à assimiler pour des débutants, c'est qu'on a du mal à comprendre à quoi ils peuvent bien servir.

Alors bien sûr, je me permets de vous rappeler la notion de **passage par adresse** vu dans l'un des chapitres précédents (chapitre sur les fonctions) qui permettait de modifier la valeur d'une variable passer en paramètre à une fonction contrairement au **passage par valeur**. Pour ceux qui l'auraient oublié, je vous invite donc fortement à relire ce chapitre avant de poursuivre…

Jusqu'ici, nous avons uniquement créé des variables faites pour contenir des nombres. Maintenant, nous allons apprendre à créer des variables faites pour contenir des adresses : ce sont justement ce qu'on appelle **des pointeurs**.

Pour créer une variable de type pointeur, on doit rajouter le symbole « ***** » devant le nom de la variable.

```
int *monPointeur;
```

NB : Notez qu'on peut aussi écrire « **int* monPointeur;** ». Cela revient exactement au même. Cependant, la première méthode est à préférer. En effet, si vous voulez déclarer plusieurs pointeurs sur la même ligne, vous serez obligés de mettre l'étoile devant le nom :

```
int *pointeur1, *pointeur2, *pointeur3;
```

Comme les variables, il est possible d'initialiser les pointeurs lors de leur déclaration. Pour **initialiser un pointeur**, c'est-à-dire lui donner une valeur par défaut, on n'utilise généralement pas le nombre 0 comme pour les entiers mais le mot-clé « **NULL** » (veillez à l'écrire en majuscules) :

```
int *monPointeur= NULL;
```

Là, vous avez un pointeur initialisé à **NULL**. Comme ça, vous saurez dans la suite de votre programme que votre pointeur ne contient aucune adresse.

Que se passe-t-il ? Ce code va réserver une case en mémoire comme si vous aviez créé une variable normale. Cependant (et c'est ce qui change), la **valeur du pointeur** est faite pour contenir une adresse (l'adresse d'une autre variable).

```
int nb = 10;
int *monPointeur= &nb;
```

La première ligne signifie : « Créer une variable de type **int** dont la valeur vaut **10** ». La seconde ligne signifie : « Créer une variable de type pointeur dont la valeur vaut l'adresse de la variable **nb** ».

Vous avez remarqué qu'il n'y a pas de type « **pointeur** » comme il y a un type **int** et un type **double**. Au lieu de ça, on utilise le symbole « ***** », mais on continue à écrire **int**. Qu'est-ce que ça signifie ? En fait, on doit indiquer quel est le type de la variable dont le pointeur va contenir l'adresse. Comme notre pointeur « **monPointeur** » va contenir l'adresse de la variable « **nb** » (qui est de **type int**), alors mon pointeur doit être de type « **int*** » ! Si ma variable « **nb** » avait été de type **double**, alors j'aurais dû écrire « **double *monPointeur** ».

Vocabulaire : on dit que le pointeur « **monPointeur** » pointe sur la variable « **nb** ».

Lorsque mon pointeur est créé, le système d'exploitation réserve une case en mémoire comme il l'a fait pour « **nb** ». La différence ici, c'est que la valeur de « **monPointeur** » est un peu particulière : elle contient l'adresse de la variable « **nb** ».

Ceci, chers lecteurs, est le secret absolu de tout programme écrit en langage C. On y est, nous venons de rentrer dans le monde merveilleux des pointeurs !

Certes, ceci n'est pas très époustouflant mais maintenant, on a un pointeur « **monPointeur** » qui contient l'adresse de la variable « **nb** ». Essayons de voir ce que contient le pointeur à l'aide d'un **printf** :

```
int nb = 10;
int *monPointeur= &nb;
printf("%d", monPointeur); //affiche 177454 qui est l'adresse de « nb »
```

Vu ainsi on pourrait se demander comment faire pour avoir la valeur de la variable « **nb** » à partir du pointeur « **monPointeur** » ? Eh ben, Il faut placer le symbole « ***** » devant le nom du pointeur :

```
int nb = 10;
int *monPointeur= &nb;
printf("%d", *monPointeur); //affiche 10 qui est la valeur de « nb »
```

Si au contraire on avait utilisé le symbole « **&** » devant le nom du pointeur, on aurait obtenu l'adresse à laquelle se trouve le pointeur.

À retenir absolument

Voici ce qu'il faut avoir compris et ce qu'il faut retenir pour la suite de ce chapitre :

- Sur une variable, comme la variable « nb » :
  - « **nb** » signifie : « Je veux la valeur de la variable **nb** »,
  - « **&nb** » signifie : « Je veux l'adresse à laquelle se trouve la variable **nb** » ;
- Sur un pointeur, comme « **monPointeur** » :
  - « **monPointeur** » signifie : « Je veux la valeur de **monPointeur** » (cette valeur étant une adresse),
  - « ***monPointeur** » signifie : « Je veux la valeur de la variable qui se trouve à l'adresse contenue dans **monPointeur** ».

Contentez-vous de bien retenir ces quatre points. Faites des tests et vérifiez que ça marche.

Attention : Ne pas confondre les différentes significations de l'étoile ! Lorsque vous déclarez un pointeur, l'étoile sert juste à indiquer qu'on veut créer un pointeur : **int *monPointeur;**.
En revanche, lorsqu'ensuite vous utilisez votre pointeur en écrivant « **printf("%d", *monPointeur);** », cela ne signifie pas « Je veux créer un pointeur » mais : « Je veux la valeur de la variable sur laquelle pointe **monPointeur** ».

II. Envoyer un pointeur à une fonction

Le gros intérêt des pointeurs (mais ce n'est pas le seul) est qu'on peut les envoyer à des fonctions pour qu'ils modifient directement une variable en mémoire, et non une copie comme on l'a vu.

Comment ça marche ? Il y a en fait plusieurs façons de faire. Voici un premier exemple :

```
void triplePointeur(int *pointeurSurNombre);

int main(int argc, char *argv[]) {
    int nombre = 5;

    /* On envoie l'adresse de nombre à la fonction */
    triplePointeur(&nombre);

    /* On affiche la variable nombre. La fonction a directement modifié la
    valeur de la variable car elle connaissait son adresse */
    printf("%d", nombre);

    return 0;
}

void triplePointeur(int *pointeurSurNombre){
    *pointeurSurNombre *= 3; // On multiplie par 3 la valeur de nombre
}
```

La fonction « **triplePointeur** » prend un paramètre de type « **int*** » (c'est-à-dire un pointeur sur **int**). Voici ce qu'il se passe dans l'ordre, en partant du début du **main** :

1. Une variable « **nombre** » est créée dans le **main**. On lui affecte la valeur **5**. Ça, vous connaissez ;
2. On appelle la fonction « **triplePointeur** ». On lui envoie en paramètre l'adresse de notre variable « **nombre** »;
3. La fonction « **triplePointeur** » reçoit cette adresse dans « **pointeurSurNombre** ». À l'intérieur de la fonction « **triplePointeur** », on a donc un pointeur « **pointeurSurNombre** » qui contient l'adresse de la variable « **nombre** »;
4. Maintenant qu'on a un pointeur sur « **nombre** », on peut modifier directement la variable « **nombre** » en mémoire ! Il suffit d'utiliser « ***pointeurSurNombre** » pour désigner la variable « **nombre** » ! Pour l'exemple, on fait un simple test : on multiplie la variable « **nombre** » par 3 ;
5. De retour dans la fonction **main**, notre nombre vaut maintenant **15** car la fonction « **triplePointeur** » a modifié directement la valeur de « **nombre** ».

Bien sûr, j'aurais pu faire un simple « **return** » comme on a appris à le faire dans le chapitre sur les fonctions. Mais l'intérêt, là, c'est que de cette manière, en utilisant des pointeurs, on peut modifier la valeur de plusieurs variables en mémoire (on peut donc « renvoyer plusieurs valeurs »). Nous ne sommes plus limités à une seule valeur !

Dans le code source qu'on vient de voir, il n'y avait pas de pointeur dans la fonction **main**. Juste une variable « **nombre** ». Le seul pointeur qu'il y avait vraiment était dans la fonction « **triplePointeur** » (de type **int***).

Il faut absolument que vous sachiez qu'il y a une autre façon d'écrire le code précédent, en ajoutant un pointeur dans la fonction **main** :

```
void triplePointeur(int *pointeurSurNombre);

int main(int argc, char *argv[]) {
    int nombre = 5;

    /* pointeur prend l'adresse de nombre */
    int *pointeur = &nombre;

    /* On envoie pointeur (l'adresse de nombre) à la fonction */
    triplePointeur(pointeur);

    /* On affiche la valeur de nombre avec *pointeur */
    printf("%d", *pointeur);

    return 0;
}

void triplePointeur(int *pointeurSurNombre) {
    *pointeurSurNombre *= 3; // On multiplie par 3 la valeur de nombre
}
```

Tests de connaissances

1. Quelle est la différence entre un pointeur et une variable ?
2. Quelle est la différence entre passage par adresse et par passage par valeur ? Donner un exemple.
3. Selon vous quel l'intérêt des pointeurs ?
4. Quel est la syntaxe de déclaration d'un pointeur ?

CHAPITRE VI : LES TABLEAUX

Objectif du chapitre

Ce chapitre est la suite directe des pointeurs et va vous faire comprendre un peu plus leur utilité. Vous comptiez y échapper ? C'est raté ! Les pointeurs sont partout en C, vous avez été prévenus !

Dans ce chapitre, nous apprendrons à créer des variables de type « tableaux ». Les tableaux sont très utilisés en C car ils sont vraiment pratiques pour organiser une série de valeurs.

Objectifs spécifiques

- Décrire brièvement le fonctionnement des tableaux ;
- Définir et utiliser les tableaux ;
- Parcourir un tableau ;
- Passer un tableau à une fonction.

I. Les tableaux dans la mémoire

« Les tableaux sont une suite de variables de même type, situées dans un espace contigu en mémoire. ». Concrètement, il s'agit de « grosses variables » pouvant contenir plusieurs nombres du même type (long, int, char, double…).

Un tableau a une dimension bien précise. Il peut occuper 2, 3, 10, 150, 2 500 cases, c'est vous qui décidez. La figure suivante est un schéma d'un tableau de 4 cases en mémoire qui commence à l'adresse 1600.

Adresse	Valeur
1600	10
1601	23
1602	505
1603	8

Lorsque vous demandez à créer un tableau de 4 cases en mémoire, votre programme demande à l'OS la permission d'utiliser **4 cases en mémoire**. Ces 4 cases doivent être contiguës, c'est-à-dire les unes à la suite des autres. Comme vous le voyez, les adresses se suivent : 1600, 1601, 1602, 1603. Il n'y a pas de « trou » au milieu.

Enfin, chaque case du tableau contient un nombre du même type. Si le tableau est de type int, alors chaque case du tableau contiendra un int. On ne peut pas faire de tableau contenant à la fois des **int** et des **double** par exemple.

En résumé, voici ce qu'il faut retenir sur les tableaux.

- Lorsqu'un tableau est créé, il prend un espace contigu en mémoire : les cases sont les unes à la suite des autres.
- Toutes les cases d'un tableau sont du même type. Ainsi, un tableau de int contiendra uniquement des int, et pas autre chose.

II. Définition et utilisation des tableaux

Pour commencer nous allons voir la syntaxe de déclaration des tableaux :

```
<type de base> nomDuTableau[<taille>];
```

- **<type de base>** : correspond au type de valeurs pouvant être stocké dans le tableau. Exemple : int, double, float, …
- **nomDuTableau** : correspond au nom du tableau à déclarer
- **<taille>** : correspond au nombre de cases que voulez mettre dans votre tableau. Exemple : 2, 4, 20, 15, …

Ainsi, si vous voulez créer un tableau d'entiers de taille 4 par exemple on aura :

```
int tableau[4];
```

Si maintenant on veut mettre des valeurs (**initialiser**), on devrait donc écrire :

```
int tableau[4];

tableau[0] = 10;
tableau[1] = 23;
tableau[2] = 505;
tableau[3] = 8;
```

Attention : Un tableau commence à l'indice **n° 0** ! Notre tableau de **4 int** a donc les indices 0, 1, 2 et 3. Il n'y a pas **d'indice 4** dans un tableau de 4 cases ! C'est une source d'erreurs très courantes, souvenez-vous-en.

Il existe une autre façon d'initialiser le tableau à savoir :

```
int tableau[4]= {10, 23, 505, 8};
```

II.1. Rapport entre les tableaux et les pointeurs

En effet, beaucoup se demanderait quelle est donc l'intérêt des pointeurs dans l'utilisation des tableaux comme mentionné dans les objectifs de ce chapitre ?
En fait, si vous reprenez l'exemple précédent et que vous écrivez juste **tableau** dans un **printf**, vous obtenez un pointeur. C'est un pointeur sur la première case du tableau. Faites le test :

```
int tableau[4];
tableau[0] = 10;
tableau[1] = 23;
tableau[2] = 505;
tableau[3] = 8;

printf("%d", tableau); // affiche l'adresse 1600
```

En revanche, si vous indiquez l'indice de la case du tableau entre crochets, vous obtenez la valeur :

```
int tableau[4];
tableau[0] = 10;
tableau[1] = 23;
tableau[2] = 505;
tableau[3] = 8;

printf("%d", tableau[0]); // affiche la valeur 10
```

Notez que comme « **tableau** » est un pointeur, on peut utiliser le symbole « ***** » pour connaître la première valeur :

```
int tableau[4];
tableau[0] = 10;
tableau[1] = 23;
tableau[2] = 505;
tableau[3] = 8;

printf("%d", *tableau); // affiche la valeur 10
```

Il est aussi possible d'obtenir la valeur de la seconde case avec « ***(tableau + 1)** » (adresse de **tableau + 1**).

```
int tableau[4];
tableau[0] = 10;
tableau[1] = 23;
tableau[2] = 505;
tableau[3] = 8;

printf("%d", *(tableau + 1)); // affiche la valeur 23
```

En clair, quand vous écrivez **tableau[0]**, vous demandez la valeur qui se trouve à l'adresse **tableau + 0** case (c'est-à-dire 1600). Si vous écrivez **tableau[1]**, vous demandez la valeur se trouvant à l'adresse **tableau + 1** case (c'est-à-dire 1601). Et ainsi de suite pour les autres valeurs.

II.2. Les tableaux à taille dynamique

Le langage C existe en plusieurs versions. Une version récente, appelée le **C99**, autorise la création de tableaux à taille dynamique, c'est-à-dire de tableaux dont la taille est définie par une variable :

```
int taille = 5;
int tableau[taille];
```

Or cela n'est pas forcément reconnu par tous les compilateurs, certains planteront sur la seconde ligne. Le langage C que je vous enseigne depuis le début (appelé le **C11**) autorise ce genre de choses.

Les versions antérieures du C tels que **C89** ne supporte pas la seconde ligne. Ainsi, dans cette version, vous n'avez pas le droit d'utiliser une variable entre crochets pour la définition de la taille du tableau, même si cette variable est une constante ! Le tableau doit avoir une dimension fixe, c'est-à-dire que vous devez écrire noir sur blanc le nombre correspondant à la taille du tableau. Du coup pour pallier à ce problème dans cette version, il faut utiliser l'**allocation dynamique** à l'aide des mots clés :

- **malloc (« Memory ALLOCation », c'est-à-dire « Allocation de mémoire »)** : qui demande au système d'exploitation la permission d'utiliser de la mémoire ;
- **free (« Libérer »)** : qui permet d'indiquer à l'OS que l'on n'a plus besoin de la mémoire qu'on avait demandée. La place en mémoire est libérée, un autre programme peut maintenant s'en servir au besoin.

III. Parcourir un tableau

Supposons que je veuille maintenant afficher les valeurs de chaque case d'un tableau. Je pourrais faire autant de **printf** qu'il y a de cases. Mais bon, ce serait répétitif et lourd, et imaginez un peu la taille de notre code si on devait afficher le contenu de chaque case du tableau une à une ! Le mieux est de se servir **d'une boucle**. Pourquoi pas d'une **boucle for** ? Les **boucles for** sont très pratiques pour parcourir un tableau.

Si nous reprenons l'exemple précédent portant sur la création d'un tableau de taille 4, on pourrait avoir :

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

int main(){
    int tableau[4];

    tableau[0] = 10;
    tableau[1] = 23;
    tableau[2] = 505;
    tableau[3] = 8;

    for (int i = 0 ; i < 4 ; i++) {
        printf("%d \n", tableau[i]);
    }

    return 0;
}
```

Notre boucle parcourt le tableau à l'aide d'une variable appelée **i** (c'est le nom très original que les programmeurs donnent en général à la variable qui leur permet de parcourir un tableau !).

La **variable i**, qui vaut successivement **0, 1, 2, et 3**. De cette façon, on va donc afficher la valeur de **tableau[0], tableau[1], tableau[2] et tableau[3]**.

Attention : Ne tenter pas d'afficher la valeur de **tableau[4]**! Un tableau de 4 cases possède les indices **0, 1, 2 et 3**, point barre. Si vous tentez d'afficher **tableau[4]**, vous aurez soit n'importe quoi, soit une belle erreur, l'OS coupant votre programme car il aura tenté d'accéder à une adresse ne lui appartenant pas.

IV. Passage de tableaux à une fonction

Il peut arriver à un moment donné que vous ayez besoin d'afficher tout le contenu de votre tableau à partir d'une fonction. Pour cela, il va falloir envoyer deux informations à la fonction : le **tableau (l'adresse du tableau)** et aussi et surtout **sa taille** !

En effet, notre fonction doit être capable d'initialiser un tableau de n'importe quelle taille. Or, dans votre fonction, vous ne connaissez pas la taille de votre tableau. C'est pour cela qu'il faut envoyer en plus une variable que vous appellerez par exemple « **tailleTableau** ».

Comme je vous l'ai dit, **tableau** peut être considéré comme un pointeur. On peut donc l'envoyer à la fonction comme on l'aurait fait avec un vulgaire pointeur :

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

// Prototype de la fonction d'affichage
void affiche(int *tableau, int tailleTableau);

int main() {
    int tableau[4] = {10, 15, 3};

    // On affiche le contenu du tableau
    affiche(tableau, 4);
    return 0;
}

void affiche(int *tableau, int tailleTableau) {
    for (int i = 0 ; i < tailleTableau ; i++){
        printf("%d \n", tableau[i]);
    }
}
```

La fonction n'est pas différente de celles que l'on a étudiées dans le chapitre sur les pointeurs. Elle prend en paramètre un **pointeur sur int** (notre tableau), ainsi que la **taille du tableau** (très important pour savoir quand s'arrêter dans la boucle !). Tout le contenu du tableau est affiché par la fonction via une boucle. Notez qu'il existe une autre façon d'indiquer que la fonction reçoit un tableau. Plutôt que d'indiquer que la fonction attend un **int *tableau**, mettez ceci :

```
void affiche(int tableau[], int tailleTableau);
```

Cela revient exactement au même, mais la présence des crochets permet au programmeur de bien voir que c'est un tableau que la fonction prend, et non un simple pointeur. Cela permet d'éviter des confusions.

J'utilise personnellement tout le temps les crochets dans mes fonctions pour bien montrer que la fonction attend un tableau. Je vous conseille de faire de même. Il n'est pas nécessaire de mettre la taille du tableau entre les crochets cette fois.

Tests de connaissances

1. Quelle est la différence entre un tableau et un variable ?
2. Quelle est la syntaxe de déclaration d'un tableau ?
3. Selon vous, quel boucle il est préférable d'utiliser pour parcourir un tableau ?
4. Ecrire une fonction « **sommeTableau** » qui renvoie la somme des valeurs contenues dans le tableau
5. Ecrire une fonction « **moyenneTableau** » qui calcule et renvoie la moyenne des valeurs.
6. Ecrire une fonction « **copierTableau** » qui prend en paramètre deux tableaux.
7. Ecrire une fonction « **ordonnerTableau** » qui classe les valeurs d'un tableau dans l'ordre croissant.

CHAPITRE VII : LES CHAINES DE CARACTERES

Objectif du chapitre

Manipuler les chaînes de caractères.

Objectifs spécifiques

- Définir et utiliser les chaînes de caractère ;
- Fonctions de manipulation des chaînes de caractère.

I. Le type char

Dans ce chapitre, nous allons porter une attention particulière au type **char**. Si vous vous souvenez bien, le type **char** permet de stocker des nombres compris entre **-128 et 127**.

Si ce type **char** permet de stocker des nombres, il faut savoir qu'en C on l'utilise rarement pour ça. En général, même si le nombre est petit, on le stocke dans un **int**. Certes, ça prend un peu plus de place en mémoire, mais aujourd'hui, la mémoire, ce n'est vraiment pas ce qui manque sur un ordinateur.

Le type **char** est en fait prévu pour stocker… **une lettre** ! Attention, j'ai bien dit : **UNE lettre**.

Comme la mémoire ne peut stocker que des nombres, on a inventé une table qui fait la conversion entre les nombres et les lettres. Cette table indique ainsi par exemple que le nombre **65** équivaut à la lettre **A**.

Le langage C permet de faire très facilement la traduction **lettre <=> nombre** correspondant. Pour obtenir le nombre associé à une lettre, il suffit d'écrire cette lettre entre apostrophes, comme ceci : **'A'**. À la compilation, **'A'** sera remplacé par la valeur correspondante. Ainsi on peut avoir le code suivant :

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

int main() {
    char lettre = 'A';
    printf("%d \n", lettre); // affiche 65
    return 0;
}
```

On sait donc que la lettre A majuscule est représentée par le nombre 65. B vaut 66, C vaut 67, etc. Testez avec des minuscules et vous verrez que les valeurs sont différentes. En effet, la lettre 'a' n'est pas identique à la lettre 'A', l'ordinateur faisant la différence entre les majuscules et les minuscules (on dit qu'il « **respecte la casse** »).
La plupart des caractères « de base » sont codés entre les nombres **0 et 127**. Une table fait la conversion entre les nombres et les lettres : **la table ASCII (prononcez « Aski »)**. Le site **AsciiTable.com** est célèbre pour proposer cette table mais ce n'est pas le seul, on peut aussi la retrouver sur **Wikipédia** et bien d'autres sites encore.

II. Définition et utilisation des chaînes de caractères

Une chaîne de caractères n'est rien d'autre qu'un tableau de type **char**. Ainsi, si on veut par exemple initialiser un tableau « **chaine** » avec le texte « **Salut** », on peut utiliser différentes méthodes à savoir :

Méthode 1

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

int main() {
    char chaine[6]; // Tableau de 6 char pour stocker S-a-l-u-t + le \0
    // Initialisation de la chaîne (on écrit les caractères un à un en mémoire)
    chaine[0] = 'S';
    chaine[1] = 'a';
    chaine[2] = 'l';
    chaine[3] = 'u';
    chaine[4] = 't';
    chaine[5] = '\0';

    // Affichage de la chaîne grâce au %s du printf
    printf("%s", chaine);
    return 0;
}
```

Vous remarquerez que c'est un peu fatigant et répétitif de devoir écrire les caractères un à un comme on l'a fait dans le tableau « **chaine** ». Pour initialiser une chaîne, il existe heureusement une méthode plus simple :

Méthode 2

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

int main() {
    /* La taille du tableau chaine est automatiquement calculée */
    char chaine[] = "Salut";
    printf("%s", chaine);
    return 0;
}
```

Comme vous le voyez à la première ligne, je crée une variable de type « **char[]** ». J'aurais pu écrire aussi « **char*** », le résultat aurait été le même. En tapant entre guillemets la chaîne que vous voulez mettre dans votre tableau, le compilateur C calcule automatiquement la taille nécessaire. C'est-à-dire qu'il compte les lettres et ajoute 1 pour placer le caractère **\0**. Il écrit ensuite une à une les lettres du mot « Salut » en mémoire et ajoute le **\0** comme on l'a fait nous-mêmes manuellement quelques instants plus tôt. Bref, c'est bien plus pratique.

Il y a toutefois un défaut : ça ne marche que pour l'initialisation ! Vous ne pouvez pas écrire plus loin dans le code :

```
chaine = "Salut";
```

Cette technique est donc à réserver à l'initialisation. Après cela, il faudra écrire les caractères manuellement un à un en mémoire comme on l'a fait au début.

III. Récupérer et afficher une chaîne de caractères

En fait, on l'a déjà vu, la récupération au clavier (scanf) et l'affichage à l'écran (printf) d'une chaîne de caractères se fait à l'aide du spécificateur « **%s** ». Prenons un exemple banal : supposons qu'il nous ait demandé d'écrire un programme C permettant d'afficher le prénom d'un utilisateur entré au clavier.

Problème : vous ne savez pas combien de caractères l'utilisateur va entrer. Si vous lui demandez son prénom, il s'appelle peut-être Luc (3 caractères), mais qui vous dit qu'il ne s'appelle pas Jean-Paul (beaucoup plus de caractères).

Eh ben, il n'y a pas 36 solutions. Il va falloir créer un tableau de **char** très grand, suffisamment grand pour pouvoir stocker le prénom. On va donc créer un **char[100]**. Vous avez peut-être l'impression de gâcher de la mémoire, mais souvenez-vous encore une fois que de la place en mémoire, ce n'est pas ce qui manque (et il y a des programmes qui gâchent la mémoire de façon bien pire que cela !).

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

int main() {
    char prenom[100];

    printf("Comment t'appelles-tu ? ");
    scanf("%s", prenom);
    printf("Salut %s, je suis heureux de te rencontrer !", prenom);
    return 0;
}
```

IV. Fonctions de manipulation des chaîne de caractères

Les chaînes de caractères sont, vous vous en doutez, fréquemment utilisées. Tous les mots, tous les textes que vous voyez sur votre écran sont en fait des tableaux de **char** en mémoire qui fonctionnent comme je viens de vous l'expliquer. Afin de nous aider un peu à manipuler les chaînes, on nous fournit dans la bibliothèque « **string.h** » une pléthore de fonctions dédiées aux calculs sur des chaînes.

Procédure ou Fonction	Prototype	Description
STRCAT	char *strcat(char *s1, char *s2)	Cette fonction permet la concaténation de chaînes de caractères.
STRCHR	char *strchr(char *str, int c)	Cette fonction effectue la recherche du premier caractère «c» dans la chaîne de caractères «str».
STRCMP	int strcmp(const char *str1, const char *str2)	Cette fonction effectue la comparaison de deux chaînes de caractères.
STRCPY	char *strcpy(const char *str1, const char *str2)	Cette fonction effectue la copie d'une chaîne de caractères dans une autre chaîne de caractères.
STRCSPN	char *strcspn(const char *str1, const char *str2)	Cette fonction effectue la recherche de la sous-chaîne «str1» ne contenant aucun des caractères contenu dans «str2».
STRLEN	size_t strlen(const char *str)	Cette fonction permet de calculer la longueur de la chaîne de caractères.
STRNCAT	char *strncat(const char *str1, const char *str2, size_t n)	Cette fonction permet d'ajouter les «n» premiers caractères de la chaîne de caractères «str1» à la chaîne de caractères «str2».
STRNCMP	int strncmp(const char *str1, const char *str2, size_t n)	Cette fonction permet de comparer les «n» premiers caractères de la chaîne de caractères «str1» à la chaîne de caractères «str2».
STRNCPY	char *strncpy(const char *str1, const char *str2, size_t n)	Cette fonction permet de copier les «n» premiers caractères de la chaîne de caractères «str1» à la chaîne de caractères «str2».
STRPBRK	char *strpbrk(const char *source, const char *accept)	Cette fonction effectue la recherche dans la chaîne de caractères «source» de la chaîne de caractères «accept».
STRRCHR	char *strrchr(const char *str, int c)	Cette fonction effectue la recherche du dernière caractère «c» dans la chaîne de caractères «str».
STRSPN	int strspn(const char *str1, const char *str2)	Cette fonction effectue le calcul de la longueur de la chaîne de caractères «str1» dans lequel sont compris des caractères de la chaîne de caractères «str2».
STRSTR	char *strstr(const char *str1, const char *str2)	Cette fonction permet de rechercher la chaîne de caractères «str2» dans la chaîne de caractères «str1».
STRTOK	char *strtok(const char *str1, const char *str2)	Cette fonction permet de couper la chaîne de caractères «str1» en symbole élémentaire (Token) en les séparant

		par les caractères de la chaîne de caractères «str2».

Pour en savoir plus sur les fonctions liées à cette bibliothèque voici un lien que peu vous être utile :
https://www.gladir.com/CODER/C/referencestring.htm

Bon à savoir :

size_t est un type entier non signé. Il est suffisamment grand pour contenir les valeurs représentant les tailles, les dimensions de tableau, les index croissants et non négatifs... Ce type est défini dans **<stddef.h>** qui est inclus dans la plupart des headers standards courants (**<stdio.h>, <stdlib.h> <string.h> etc.**)

En résumé

- Un ordinateur ne sait pas manipuler du texte, il ne connaît que les nombres. Pour régler le problème, on associe à chaque lettre de l'alphabet un nombre correspondant dans une table appelée la **table ASCII**.
- Le **type char** est utilisé pour stocker une et une seule lettre. Il stocke en réalité un nombre mais ce nombre est automatiquement traduit par l'ordinateur à l'affichage.
- Pour créer un mot ou une phrase, on doit construire une chaîne de caractères. Pour cela, on utilise un **tableau de char**.
- Toute chaîne de caractère se termine par un caractère spécial appelé **\0** qui signifie « **fin de chaîne** ».
- Il existe de nombreuses fonctions toutes prêtes de manipulation des chaînes dans **la bibliothèque string**. Il faut inclure **string.h** pour pouvoir les utiliser.

Tests de connaissances

1. Quelle est la syntaxe de déclaration d'une chaîne de caractères ?
2. A quoi sert la table ASCII ?
3. Donner le prototype des fonctions permettant de réaliser les actions suivantes :
    a. Obtenir la longueur d'une chaîne de caractères
    b. Comparer deux chaînes de caractères
    c. Joindre deux chaînes de caractères
    d. Recherche un caractère dans une chaîne de caractères
4. Ecrire un programme C permettant de compter le nombre mot d'une phrase terminé par le marqueur « **.** ». On suppose que chaque mot est séparé par le caractère « espace ».

CHAPITRE VIII : LES STRUCTURES

Objectif du chapitre

Le langage C nous permet de faire quelque chose de très puissant : créer nos propres types de variables. Créer de nouveaux types de variables devient indispensable quand on cherche à faire des programmes plus complexes. Ainsi au terme de chapitre, nous allons apprendre à manipuler les structures.

Objectifs spécifiques

- Définir et utiliser les structures ;
- Définir et utiliser des pointeurs de structures.

I. Définition et utilisation des structures

I.1. Définition des structures

Une structure est un assemblage de variables qui peuvent avoir différents types. Contrairement aux tableaux qui vous obligent à utiliser le même type dans tout le tableau, vous pouvez créer une structure comportant des variables de types **long, char, int et double** à la fois.

Les structures sont généralement définies dans les fichiers « **.h** », au même titre donc que **les prototypes et les define.** Voici un exemple de structure :

```
struct NomDeVotreStructure {
    int variable1;
    int variable2;
    int autreVariable;
    double nombreDecimal;
};
```

Une définition de structure commence par le mot-clé **struct**, suivi du nom de votre structure (par exemple **Fichier**, ou encore **Ecran**).

J'ai personnellement l'habitude de nommer mes structures en suivant les mêmes règles que pour les noms de variables, excepté que je mets la première **lettre en majuscule** pour pouvoir faire la différence. Ainsi, quand je vois le mot **ageDuCapitaine** dans mon code, je sais que c'est une variable car cela commence par une **lettre minuscule**. Quand je vois **MorceauAudio** je sais qu'il s'agit d'une structure (un type personnalisé) car cela commence par une majuscule.

Après le nom de votre structure, vous ouvrez les accolades et les fermez plus loin, comme pour une fonction.

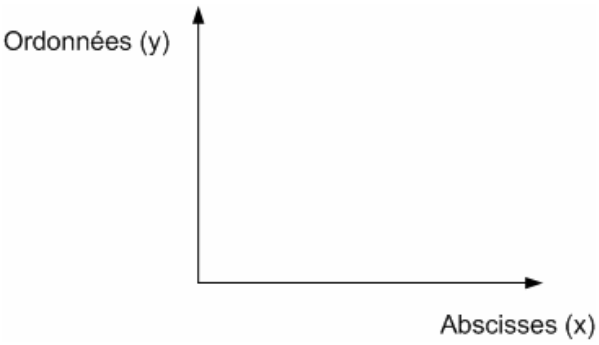
Attention : Ici c'est particulier : vous DEVEZ mettre un point-virgule après l'accolade fermante. C'est obligatoire. Si vous ne le faites pas, la compilation plantera.

Et maintenant, que mettre entre les accolades ? C'est simple, vous y placez les variables dont est composée votre structure. Une structure est généralement composée d'au moins deux « sous-variables », sinon elle n'a pas trop d'intérêt.

Comme vous le voyez, la création d'un type de variable personnalisé n'est pas bien complexe. Toutes les structures que vous verrez sont en fait des « assemblages » de variables de type de base, comme **long, int, double, etc**.

Exemple pratique de structure

Imaginons par exemple que vous vouliez créer une variable qui stocke les coordonnées d'un point à l'écran. Pour ceux chez qui le mot « **géométrie** » provoque des apparitions de boutons inexplicables sur tout le visage, la figure suivante va faire office de petit rappel fondamental.



On a **deux axes** : **l'axe des abscisses** (de gauche à droite) et **l'axe des ordonnées** (de bas en haut). On a l'habitude d'exprimer les abscisses par une variable appelée **x**, et les ordonnées par **y**.

Il est donc question ici d'écrire une structure nommée « **Coordonnees** » qui permette de stocker à la fois la valeur de l'abscisse (x) et celle de l'ordonnée (y) d'un point. On a donc le code suivant :

```
struct Coordonnees {
    int x; // Abscisses
    int y; // Ordonnées
};
```

Notre structure s'appelle « **Coordonnees** » et est composée de deux variables entières **x** et **y**, c'est-à-dire de l'**abscisse** et de l'**ordonnée**.

I.2. Tableaux dans une structure

Les structures peuvent contenir également des tableaux. Imaginons une structure « **Personne** » qui stockerait diverses informations sur une personne :

```
struct Personne {
    char nom[100];
    char prenom[100];
    char adresse[1000];

    int age;
    int genre; // Booléen : 1 = garçon, 0 = fille
};
```

Cette structure est composée de cinq sous-variables. Les trois premières sont des chaînes qui stockeront le nom, le prénom et l'adresse de la personne. Les deux dernières stockent l'âge et le sexe de la personne. Le sexe est un booléen, 1 = vrai = garçon, 0 = faux = fille.

Cette structure pourrait servir à créer un programme de carnet d'adresses. Bien entendu, vous pouvez rajouter des variables dans la structure pour la compléter si vous le voulez. Il n'y a pas de limite au nombre de variables dans une structure.

I.3. Utilisation d'une structure

Maintenant que notre structure est définie (généralement dans un .h), on va pouvoir l'utiliser dans une fonction (généralement le **main**).

Voici comment créer une variable de type « **Coordonnees** » (la structure qu'on a définie plus haut) :

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

struct Coordonnees {
    int x;
    int y;
};

int main() {
    /* Création d'une variable "point" de type Coordonnees */
    struct Coordonnees point;

    return 0;
}
```

On suppose ici que la définition et l'utilisation de la structure a été fait dans le même fichier .c. Mais on aurait pu (et on aurait même dû) déclarer la structure dans un fichier « **structure.h** » par

exemple, inclure le fichier « **structure.h** » dans le fichier « **main.c** » et enfin utiliser la structure dans la fonction « **main** ».

Nous avons ainsi créé une variable « **point** » de type « **Coordonnees** ». Cette variable est automatiquement composée de deux sous-variables : **x et y** (son abscisse et son ordonnée).

NB : Le mot clé « **struct** » utilisé devant « **Coordonnees** » lors de la définition de la variable personnalisée « **point** » dans la fonction « **main** » est obligatoire. Toutefois, les programmeurs trouvent souvent un peu lourd de mettre le mot clé « **struct** » à chaque définition de variable personnalisée. Pour régler ce problème, ils ont inventé une instruction spéciale nommée : **typedef**. Le **typedef** sert à créer un alias de structure, c'est-à-dire à dire qu'écrire telle chose équivaut à écrire telle autre chose. Vu qu'un exemple vaut toujours mieux qu'un discours, reprenons l'exemple précédent et introduisons le **typedef** :

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

struct Coordonnees {
    int x;
    int y;
};
typedef struct Coordonnees Coordonnees;

int main() {
    /* Création d'une variable "point" de type Coordonnees */
    Coordonnees point;

    return 0;
}
```

Prenons la ligne « **typedef struct Coordonnees Coordonnees;** » :

- **typedef** : indique que nous allons créer un alias de structure ;
- **struct Coordonnees** : c'est le nom de la structure dont vous allez créer un alias (c'est-à-dire un « équivalent ») ;
- **Coordonnees** : c'est le nom de l'équivalent.

En clair, cette ligne dit « Écrire le mot **Coordonnees** est désormais équivalent à écrire **struct Coordonnees** ». En faisant cela, vous n'aurez plus besoin de mettre le mot « **struct** » à chaque définition de variable de type « **Coordonnees** ».

Je vous recommande de faire un **typedef** comme je l'ai fait ici pour « **Coordonnees** ». La plupart des programmeurs font comme cela. Ça leur évite d'avoir à écrire le mot « **struct** » partout. Un bon programmeur est un programmeur « **fainéant** » ! Il en écrit le moins possible.

Maintenant que notre variable « **point** » est créée, on pourrait se demander comment modifier ses coordonnées ? Eh ben, comme ceci :

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

struct Coordonnees {
    int x;
    int y;
};
typedef struct Coordonnees Coordonnees;

int main() {
    /* Création d'une variable "point" de type Coordonnees */
    Coordonnees point;
    point.x= 10;
    point.y= 20;

    return 0;
}
```

On a ainsi modifié la valeur de **point**, en lui donnant une **abscisse de 10** et une **ordonnée de 20**. Notre point se situe désormais à la **position (10 ; 20)** (c'est la notation mathématique d'une coordonnée.

Pour accéder donc à chaque composante de la structure, vous devez écrire :

```
variable.nomDeLaComposante
```

Le caractère point « **.** » fait la séparation entre la variable et la composante.

NB : Il est important de noter que tout comme les variables de type votre structure, il est possible également de définir des tableaux de type votre structure. Ces tableaux auront le même fonctionnement que les tableaux définis dans le chapitre portant sur les tableaux.

I.4. Initialiser une structure

Pour les structures comme pour les variables, tableaux et pointeurs, il est vivement conseillé de les initialiser dès leur création pour éviter qu'elles ne contiennent « n'importe quoi ». En effet, je vous le rappelle, une variable qui est créée prend la valeur de ce qui se trouve en mémoire là où elle a été placée. Parfois cette valeur est 0, parfois c'est un résidu d'un autre programme qui est passé par là avant vous et la variable a alors une valeur qui n'a aucun sens, comme -84570.

Pour rappel, voici comment on initialise :

- **Une variable** : on met sa valeur à 0 (cas le plus simple) ;
- **Un pointeur** : on met sa valeur à **NULL**. **NULL** est en fait un **#define** situé dans **stdlib.h** qui vaut généralement **0**, mais on continue à utiliser **NULL**, par convention, sur les pointeurs pour bien voir qu'il s'agit de pointeurs et non de variables ordinaires ;
- **Un tableau** : on met chacune de ses valeurs à **0**.

Pour les structures, l'initialisation va un peu ressembler à celle d'un tableau. En effet, on peut avoir :

```
Coordonnees point= {0, 0};
```

Cela définira, dans l'ordre, « **point.x = 0** » et « **point.y = 0** ».

Revenons à la structure « **Personne** » (qui contient des chaînes). Vous avez aussi le droit d'initialiser une chaîne en écrivant juste « "" » (rien entre les guillemets). Je ne vous ai pas parlé de cette possibilité dans le chapitre sur les chaînes, mais il n'est pas trop tard pour l'apprendre. On peut donc initialiser dans l'ordre **nom, prenom, adresse, age et garcon** comme ceci :

```
Personne utilisateur = {"", "", "", 0, 0};
```

Toutefois, j'utilise assez peu cette technique, personnellement. Je préfère envoyer par exemple ma variable « **point** » à une fonction « **initialiserCoordonnees** » qui se charge de faire les initialisations pour moi sur ma variable. Pour faire cela il faut envoyer un pointeur de ma variable. En effet si j'envoie juste ma variable, une copie en sera réalisée dans la fonction (comme pour une variable de base) et la fonction modifiera les valeurs de la copie et non celle de ma vraie variable.

II. Les pointeurs de structure

Un pointeur de structure se crée de la même manière qu'un pointeur **de int, de double ou de n'importe quelle autre type** de base :

```
Coordonnees *point = NULL;
```

Ce qui nous intéresse ici, c'est de savoir comment envoyer un pointeur de structure à une fonction pour que celle-ci puisse modifier le contenu de la variable.

On va faire ceci comme exemple : on va simplement créer une variable de type « **Coordonnees** » dans le « **main** » et envoyer son adresse à une fonction « **initialiserCoordonnees** ». Cette fonction aura pour rôle de mettre tous les éléments de la structure à **0**. Notre fonction « **initialiserCoordonnees** » va prendre un paramètre : un pointeur sur une structure de type « **Coordonnees** ».

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

struct Coordonnees {
    int x;
    int y;
};
typedef struct Coordonnees Coordonnees;

void initialiserCoordonnees(Coordonnees *point) {
    (*point).x= 0;
    (*point).y= 0;
}

int main() {
    Coordonnees point;
    initialiserCoordonnees(&point);

    return 0;
}
```

Remarqué bien la différence entre « **(*point).x= 0** » et « ***point.x= 0** ».

Sur « ***point.x= 0** », le point de séparation s'applique sur le mot « **point** » et non sur « ***point** » en entier. Or, nous ce qu'on veut, c'est accéder à « ***point** » pour en modifier la valeur. Pour régler le problème, il faut placer des parenthèses autour de « ***point** ».

Ainsi, ce code fonctionne, vous pouvez tester. La variable de type « **Coordonnees** » a été transmise à la fonction qui a initialisé **x et y** à 0.

NB : En langage **C**, on initialise généralement nos structures avec la méthode simple qu'on a vue plus haut. En **C++** en revanche, les initialisations sont plus souvent faites dans des « fonctions ». Le **C++** n'est en fait rien d'autre qu'une sorte de « super-amélioration » des structures. Bien entendu, beaucoup de choses découlent de cela et il faudrait un livre entier pour en parler (chaque chose en son temps).

Remarque : Au lieu d'écrire « **(*point).x= 0** », vous auriez pu écrire « **point->x= 0** » : les deux expressions ont la même signification.

Tests de connaissances

1. Définir structure
2. Quelle est la différence entre une structure et un tableau ?
3. Quelle est la syntaxe de déclaration d'une structure ?
4. Comment faire pour créer un alias de structure ?
5. Selon vous, quelle est la meilleure méthode pour initialiser une structure ?