**Programmer, c'est quoi ?**

On commence par la question la plus simple qui soit, la plus basique de toutes les questions basiques :p  
Si vous avez l'impression de déjà savoir tout ça, je vous conseille de lire quand même, ça ne peut pas vous faire de mal ;) Je pars de zéro pour ce cours, donc je vais devoir répondre à la question :

Que signifie le mot "programmer" ?

Bon, je vais éviter de vous faire comme mon prof de français : je ne vais pas vous donner l'origine du mot "programmer". Et puis de toute façon si je vous disais que ça vient du latin *programmeus*je crois que vous auriez un peu de mal à me croire :lol:  
Simplement, programmer signifie réaliser des "programmes informatiques". Les programmes demandent à l'ordinateur d'effectuer des actions.  
  
**Votre ordinateur est rempli de programmes en tous genres :**

* La calculatrice est un programme
* Votre traitement de texte est un programme
* Votre logiciel de « Chat » est un programme
* Les jeux vidéo sont des programmes

En bref, les programmes sont partout et permettent de faire à priori tout et n'importe quoi sur un ordinateur. Vous pouvez inventer un logiciel de cryptage révolutionnaire si ça vous chante, ou réaliser un jeu de combat en 3D sur Internet, peu importe. Votre ordinateur peut tout faire (sauf le café, mais j'y travaille ^^ ).

  
*Le célèbre jeu Half-Life 2, programmé en C++*

Attention ! Je n'ai pas dit que réaliser un jeu vidéo se faisait en claquant des doigts. J'ai simplement dit que tout cela était possible, mais soyez sûrs que ça demandebeaucoup de travail

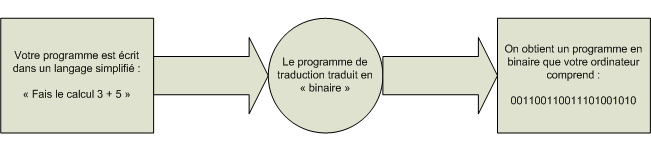
Comme vous débutez, nous n'allons pas commencer par voir comment réaliser un jeu 3D. Ce serait du pur suicide ^^  
Nous allons devoir passer par des choses très simples. Une des premières choses que nous verrons est *comment afficher un message à l'écran*. Oui, je sais ça n'a rien de très transcendant, mais rien que ça croyez-moi, c'est pas si facile que ça en a l'air :D  
  
Bon, c'est vrai que ça impressionne moins les copains, mais on va bien devoir passer par là. Petit à petit, vous apprendrez suffisamment de choses pour commencer à réaliser des programmes de plus en plus complexes. Le but de ce cours est que vous soyez capables de vous débrouiller tous seuls dans n'importe quel programme écrit en C ou C++.  
  
Mais tenez au fait, vous savez ce que c'est vous, cette histoire de "C / C++" ? o_O

**Programmer, dans quel langage ?**

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 :  
  
0010110110010011010011110  
  
*(j’invente hein, je ne connais pas la traduction informatique par cœur :p)*  
Ce que vous voyez là, c’est le langage informatique de votre ordinateur, appelé **langage binaire** (retenez bien ce mot !). Votre ordinateur ne connaît que ce langage-là et, comme vous pouvez le constater, c’est absolument incompréhensible, immonde et imbuvable :D  
  
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, c’est vous dire ^^  
  
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 (ouf ! :D ). 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 :  
"0010110110010011010011110".  
  
Si on fait un schéma de ce que je viens de dire, ça donne quelque chose comme ça :

  
*Schéma ( super-simplifié :p ) de réalisation d’un programme*

**Un peu de vocabulaire**

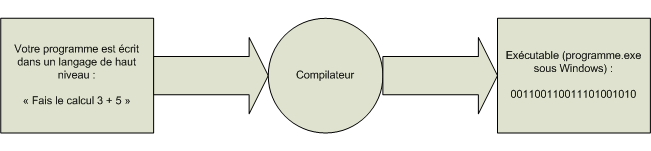
Là j’ai parlé avec des mots simples, mais il faut savoir qu’en informatique il existe un mot pour chacune de ces choses-là. Tout au long de ce cours, vous allez d’ailleurs apprendre pas mal de vocabulaire.  
Non seulement vous aurez l’air de savoir de quoi vous parlez, mais si un jour (et ça arrivera) 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 faudra pas y faire attention :p  
  
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 en fait "**langage de haut niveau**".  
Il existe plusieurs "niveaux" de langages. 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 (chouette ! :) ), mais cela a aussi quelques petits défauts comme nous le verrons plus tard.  
  
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++
* Java
* Visual Basic
* Delphi
* Etc 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 en vrac qui me sont passés par la tête.  
*(et d’avance désolé pour tous les autres langages qui existent, mais faire une liste complète serait vraiment trop long ^^ )*  
  
Certains de ces langages sont plus haut niveau que d’autres (donc en théorie un peu plus faciles à utiliser), on va voir un peu plus loin notamment ce qui différencie le langage C du langage 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 haut niveau. 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 haut niveau (comme le C ou le C++) en binaire. Ce programme a un nom : on l’appelle **le compilateur**. La traduction, elle, s'appelle **la compilation**.  
Très important : il existe un compilateur différent pour chaque langage de haut niveau. 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 ! Il y a le compilateur écrit par Microsoft, le compilateur GNU etc. On verra tout ça dans le chapitre suivant.  
Heureusement, ces compilateurs-là sont quasiment identiques (même s’il y a parfois quelques "légères" différences que nous apprendrons à reconnaître).**

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" comme EXEcutable.  
  
Reprenons notre schéma de tout à l’heure, et utilisons cette fois des vrais mots tordus d’informaticien. Ca donne :

  
*Le même schéma, avec le bon vocabulaire*

**Pourquoi choisir d’apprendre le C / C++ ?**

Comme je vous l’ai dit plus haut, il existe de très nombreux langages de haut niveau. 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 très haut niveau** : c’est facile à utiliser, plutôt "grand public". Parmi eux, on compte Python, Ruby, Visual Basic 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 un peu plus bas niveau** (mais pas trop quand même !) : ils sont peut-être un peu plus difficiles certes, mais avec un langage comme le C (ou 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 et le C++ sont des langages très populaires. Ils sont utilisés 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.  
  
Je vais supposer tout au long de ce cours que c’est votre premier langage de programmation, que vous n’avez jamais fait de programmation avant. Si, par hasard, vous avez déjà un peu programmé, ça ne vous fera pas de mal de reprendre à zéro ;)

Stop, il y a quelque chose que je ne comprends pas… Je vais apprendre un langage appelé "C / C++" ou je vais apprendre 2 langages : l’un appelé "C" et l’autre appelé "C++" ?

La bonne réponse est que vous allez apprendre en fait 2 langages. Non, ça ne va pas faire 2 fois plus de travail :p  
Je m’explique. Le langage C et le langage C++ sont très similaires. **Quand je désigne les 2 à la fois (comme je l’ai fait jusqu’ici), j’écris "C / C++".**  
Voici ce qu’il faut savoir sur la différence entre les 2 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 là :-° )
* Puis, un beau jour, on en est arrivés à créer encore un autre langage qu’on a appelé... le **langage C**. Ce langage, s'il a subi quelques modifications, 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 (quels ajouts ? On verra ça plus tard dans le cours).

Il y a plusieurs façons d’apprendre la programmation, je vous l’ai dit plus haut.  
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++ (enfin, y’a de quoi dire quand même ^^ )

**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.**

Ce n’est PAS parce que Half-Life 2 a été codé en C++ qu’il faut absolument faire du C++ pour réaliser des jeux ou des programmes complexes.  
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.  
  
L’avantage, c’est qu’à la fin de ce cours vous saurez aussi bien programmer en C qu’en C++ selon vos besoins :)

**Programmer, c'est dur ?**

Voilà une question qui doit bien vous torturer l'esprit ^^  
Alors : faut-il être un super mathématicien qui a fait 10 ans d'études supérieures pour pouvoir commencer la programmation ?  
La réponse, que je vous rassure, est non ;)  
Non, un super niveau en maths n'est pas nécessaire. En fait tout ce que vous avez besoin de connaître, ce sont les 4 opérations de base :

* L'addition (bon j'espère que vous maîtrisez :p )
* La soustraction (ouille ouille ouille !)
* La multiplication (argh)
* La division (bah pourquoi y'a plus personne tout à coup ? o_O )

J'espère que vous connaissez tout ça ;) Et histoire d'en être sûr, je vous expliquerai dans un prochain chapitre comment l'ordinateur réalise ces opérations de base.  
  
Bref, niveau maths, il n'y a pas de difficulté insurmontable :D  
En fait, tout dépend du programme que vous allez faire : si vous devez faire un logiciel de cryptage, alors oui il vous faudra connaître des choses en maths. Si vous devez faire un programme qui fait de la 3D, oui il vous faudra quelques connaissances en géométrie de l'espace.  
  
Chaque cas est particulier.  
Pour apprendre le langage C / C++, vous n'avez pas besoin de connaissances pointues en quoi que ce soit.

**Mais alors, où est le piège ? Où est la difficulté ?**

Il faut savoir comment un ordinateur fonctionne pour comprendre ce qu'on fait. De ce point de vue-là, rassurez-vous, je vous apprendrai tout au fur et à mesure.  
  
Un programmeur a aussi certaines qualités comme :

* **La patience** : un programme ne marche jamais du premier coup, il faut savoir persévérer !
* **Le sens de la logique** : pas besoin d'être fort en maths certes, mais ça ne vous empêchera pas d'avoir à réfléchir (*ah ben zut alors !* :lol: )
* **Le calme** : on ne tape pas sur son ordinateur avec un marteau :p Ce n'est pas ça qui fera marcher votre programme ^^

En bref, et pour faire simple, il n'y a pas de véritables connaissances requises pour programmer. Un nul en maths peut s'en sortir sans problème, le tout est d'avoir la patience de réfléchir. Il y en a beaucoup d'ailleurs qui découvrent qu'ils adorent ça ! :)

**Les outils nécessaires au programmeur**

Alors à votre avis, de quels outils un programmeur a-t-il besoin ?  
Si vous avez attentivement suivi le chapitre précédent, vous devez en connaître au moins un !  
  
Vous voyez de quoi je parle ????  
Vraiment pas ? ^^  
  
Eh oui, il s'agit du **compilateur**, ce fameux programme qui permet de traduire votre langage C en langage binaire !  
  
Comme je vous l'avais un peu déjà dit dans le premier chapitre, il existe plusieurs compilateurs pour le langage C / C++. Nous allons voir que le choix du compilateur ne sera pas très compliqué dans notre cas ;)  
  
Bon, de quoi d'autre a-t-on besoin ?  
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 C ou C++). 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 repérer dedans bien plus facilement
* **Un compilateur** pour transformer ("compiler") votre source en binaire.
* **Un débugger** pour vous aider à traquer les erreurs dans votre programme (on n'a malheureusement pas encore inventé le "correcteur", un truc qui corrigerait tout seul nos erreurs :soleil: )

A priori, si vous êtes un casse-cou de l'extrême, vous pouvez vous passer de débugger ? Mais bon, je sais pertinemment que dans moins de 5 minutes vous reviendrez en pleurnichant me demander où on peut trouver un débugger qui marche bien ^^  
  
A partir de maintenant on a 2 possibilités :

* Soit on récupère chacun de ces 3 programmes **séparément**. C'est la méthode la plus compliquée, mais elle fonctionne ^^ Sous Linux en particulier, bon nombre de programmeurs préfèrent utiliser ces 3 programmes séparément. Je ne détaillerai pas cette méthode ici, je vais plutôt vous parler de la méthode simple.
* Soit on utilise un programme "3-en-1" (comme les liquides vaisselle, oui oui) qui combine éditeur de texte, compilateur et débugger. Ces programmes "3-en-1" sont appelés **IDE**, ou encore "Environnements de développement"

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

Il m'a semblé intéressant de vous montrer 3 IDE parmi les plus connus. Tous sont disponibles gratuitement. Personnellement, je navigue un peu entre tous ceux-là et j'utilise l'IDE qui me plaît selon l'humeur du jour ;)

* Vous avez par exemple **Dev C++** qui est très bien. Que son nom ne vous trompe pas : vous pouvez aussi bien faire du C que du C++ avec lui ;)
* Plus récent que Dev C++, l'IDE **Code::Blocks** semble promis à un bel avenir. Il est aussi gratuit et plus tenu à jour que Dev. Il possède en outre quelques fonctionnalités intéressantes et fonctionne sous Windows et Linux. Je conseille d'utiliser celui-ci pour débuter.
* Un des IDE les plus connus, c'est celui de Microsoft : **Visual C++**. Il existe à la base en version payante (chère !), mais heureusement il existe une version gratuite intitulée **Visual C++ Express** qui est vraiment très bien (il y a peu de différences avec la version payante).

Quel est le meilleur de tous ces IDE ?

Personnellement, entre Dev C++, Code::Blocks et Visual C++ j'aurais tendance à préférer Visual. Je trouve son débugger plus puissant. Toutefois, comme vous débutez vous ne serez pas capables de profiter de toute sa puissance. Vous vous y mettrez sûrement, mais plus tard.  
  
Cela nous laisse donc un choix entre Dev C++ et Code::Blocks pour commencer. Grosso modo, il faut retenir que Dev-C++ est un IDE qui a eu beaucoup de succès pendant un moment, mais il n'est plus mis à jour depuis bien trop longtemps à mon goût. Vous en entendrez quand même sûrement parler car beaucoup de personnes l'ont utilisé et l'utilisent encore.  
Je vous recommande donc Code::Blocks pour commencer, mais ce n'est pas une obligation. Quel que soit l'IDE que vous choisissez vous serez capables de faire autant de choses. Vous n'êtes pas limités.  
  
Dans la suite de ce chapitre, je vais présenter chacun de ces 3 IDE histoire que vous puissiez un peu les comparer, ne serait-ce que visuellement.  
  
Au passage, notez que ces IDE sont disponibles sous Windows uniquement, sauf Code::Blocks qui fonctionne aussi sous Linux.

**Attention : si vous êtes sous Windows Vista, il faudra suivre des instructions particulières pour installer votre IDE correctement, sinon vous aurez des bugs à la compilation.  
Je vous conseille d'installer Visual Studio Express (il marche sous Vista sans problème), ou encore d'installer Code::Blocks en suivant les instructions spéciales pour Vista que j'y donne.**

**Ou encore... Code::Blocks**

Code::Blocks est un IDE libre et gratuit, plus récent que Dev-C++.  
Si Dev reste un des IDE gratuits les plus connus, il le doit surtout à son ancienneté.  
  
Code::Blocks étant relativement nouveau, il n'est pas encore aussi connu. Pourtant, n'allez pas penser qu'il est peu avancé pour autant ! Bien au contraire, je dois avouer que j'ai été surpris par cet éditeur et je vous recommande de l'essayer :)  
  
En outre, Code::Blocks est disponible **pour Windows, Mac et Linux**.  
Code::Blocks n'est disponible pour le moment qu'en anglais. Ca ne devrait PAS vous repousser à l'utiliser. Quand vous programmerez vous serez de toute façon confronté bien souvent à des documentations en anglais, donc raison de plus pour s'entraîner à utiliser cette langue. Ca ne complique pas l'utilisation du logiciel de toute manière.  
  
**Télécharger Code::Blocks**

Rendez-vous [sur la page de téléchargements de Code::Blocks](http://www.codeblocks.org/downloads.shtml).  
  
Si vous êtes sous Windows, repérez la section "Windows" un peu plus bas sur cette page. Téléchargez le logiciel en prenant le programme qui contient mingw dans le nom (ex. : codeblocks-8.02mingw-setup.exe). L'autre version étant sans compilateur, vous auriez eu du mal à compiler vos programmes :p  
Si vous êtes sous Linux, il y a un lien en haut pour choisir le package RPM à télécharger.  
  
L'installation est très simple et rapide. Laissez toutes les options par défaut et lancez le programme.

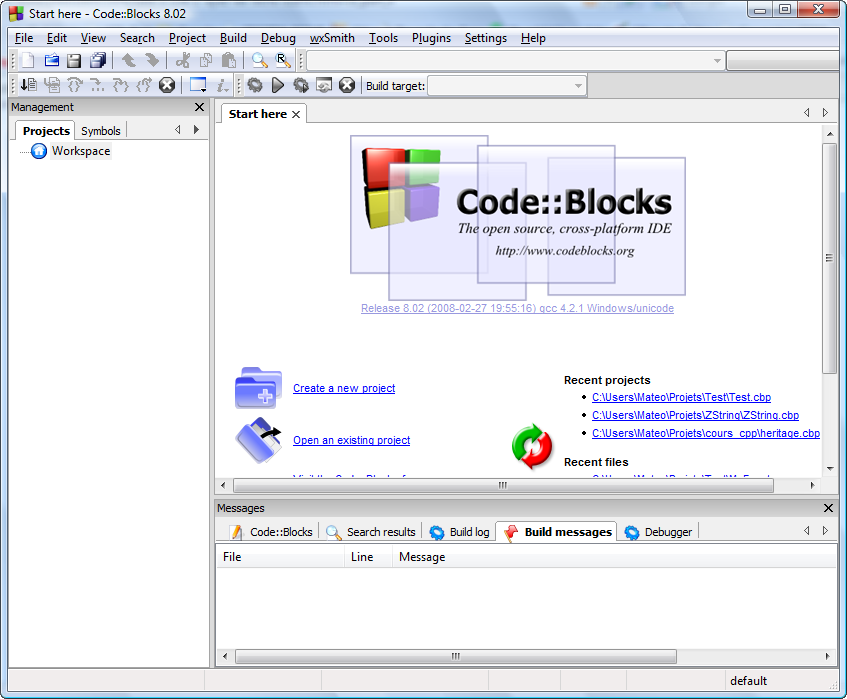
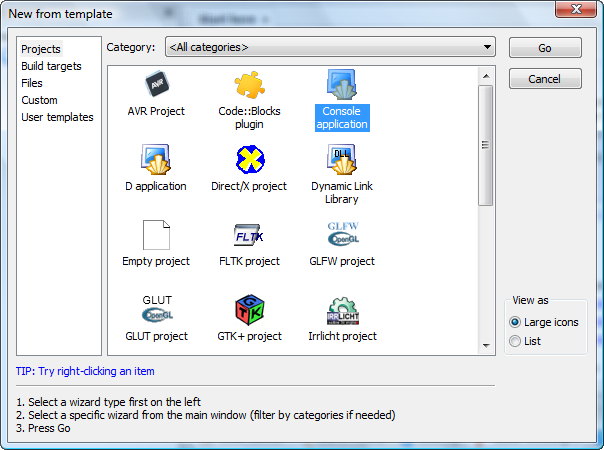


Image utilisateur

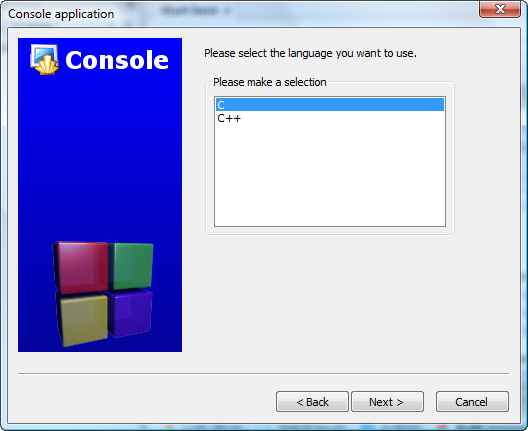
Le fonctionnement du programme est quasiment le même que pour Dev et Visual, vous ne serez pas perdus.  
Vous trouverez là encore dans la barre d'outils les boutons (dans l'ordre) "Compiler", "Exécuter", "Compiler & Exécuter" et "Tout recompiler" (comme Dev ^^ )  
  
**Créer un nouveau projet**

Pour créer un nouveau projet c'est très simple : allez dans le menu File / New / Project.  
Dans la fenêtre qui s'ouvre, choisissez "Console application" :

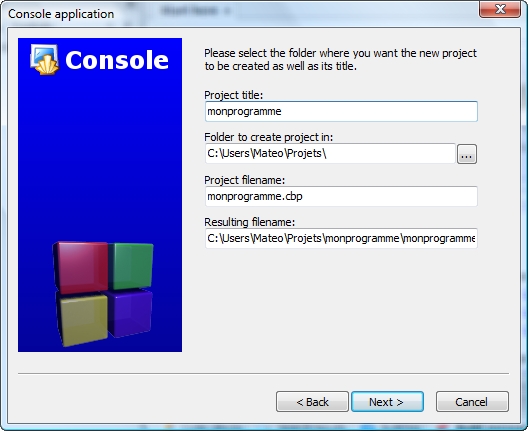


**Comme vous pouvez le voir, Code::Blocks propose de réaliser pas mal de types de programmes différents qui utilisent des librairies connues comme la SDL (2D), OpenGL (3D), QT et wxWidgets (Fenêtres) etc etc... Pour l'instant, ces icônes servent plutôt à faire joli car les librairies ne sont pas installés sur votre ordinateur, vous ne pourrez donc pas les faire marcher.  
Nous nous intéresserons à ces autres types de programmes bien plus tard. En attendant il faudra vous contenter de "Console", car vous n'avez pas encore le niveau nécessaire pour créer les autres types de programmes.**

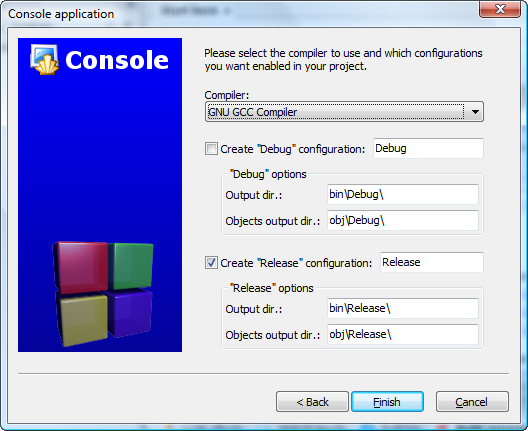
Cliquez sur "Go" pour créer le projet. Un assistant s'ouvre.  
  
Faites "Next", la première page ne servant à rien.  
On vous demande ensuite si vous allez faire du C ou du C++ : répondez C.



On vous demande le nom de votre projet, et dans quel dossier les fichiers source seront enregistrés :



Enfin, la dernière page 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 pour ce que nous allons faire dans l'immédiat (veillez à ce que "Debug" ou "Release" au moins soit coché).



Cliquez sur "Finish", c'est bon !  
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.

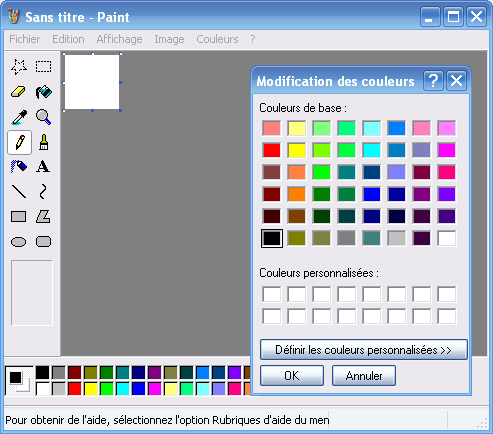
**Console ou fenêtre ?**

Console ou fenêtre ?   
  
Nous en avons rapidement parlé dans le chapitre précédent. Notre IDE (Dev ou Visual) nous demandait quel type de programme nous voulions créer, et je vous avais dit de répondre *console*.  
  
Il faut savoir qu'en fait il existe 2 types de programmes, pas plus :

* Les programmes avec fenêtres
* Les programmes en console

**Les programmes en fenêtres**

Ce sont les programmes que vous connaissez.  
Voici un exemple de programme en fenêtres que vous connaissez sûrement :

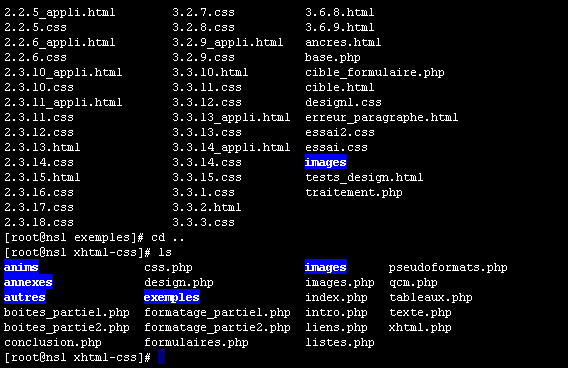
  
*Le programme Paint*

Ca donc, c'est un programme avec des fenêtres.  
Je suppose que vous aimeriez bien créer ce type de programmes, mmh ? Eh ben vous allez pas pouvoir de suite :p  
  
En effet, créer des programmes avec des fenêtres en C / C++ c'est possible, mais... Quand on débute, c'est bien trop compliqué !  
Pour débuter, il vaut mieux commencer par créer des programmes en console.

**Mais au fait, à quoi ça ressemble un programme en console ?**

**Les programmes en console**

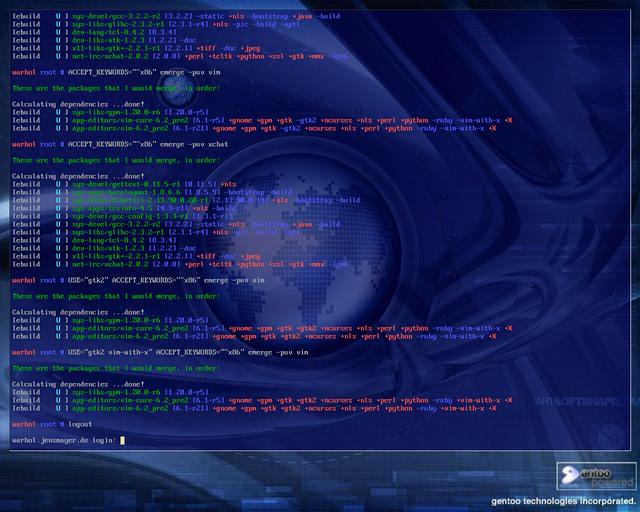
Les programmes console ont été les premiers à apparaître. A cette époque, l'ordinateur ne gérait que le noir et blanc et il n'était pas assez puissant pour créer des fenêtres comme on le fait aujourd'hui.  
  
Bien entendu, le temps a passé depuis. Windows a rendu l'ordinateur "grand public" principalement grâce à sa simplicité et au fait qu'il n'utilisait que des fenêtres. Windows est devenu tellement populaire qu'aujourd'hui presque tout le monde a oublié ce qu'était la console !  
Oui vous là, ne regardez pas derrière vous, je sais que vous vous demandez ce que c'est :lol:  
  
J'ai une grande nouvelle ! **La console n'est pas morte** ! :D  
En effet, Linux a remis au goût du jour l'utilisation de la console. Voici une capture d'écran d'une console sous Linux :

  
*Un exemple de console, ici sous Linux*

Brrr... Terrifiant hein ? :diable:  
Voilà, vous avez maintenant une petite idée de ce à quoi ressemble une console ;)  
  
Plusieurs remarques ceci dit :

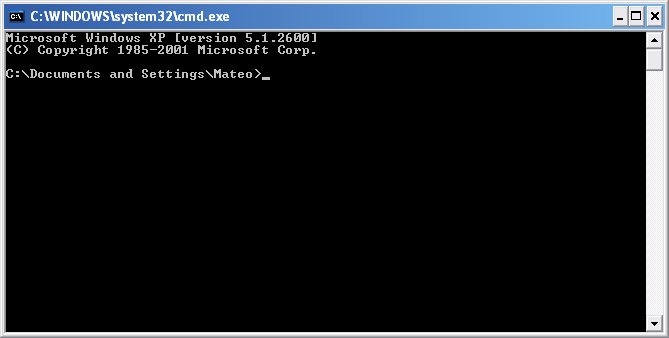
* Comme vous pouvez le voir, aujourd'hui on sait afficher de la couleur, tout n'est donc pas en noir et blanc ^^
* La console est assez peu accueillante pour un débutant
* C'est pourtant un outil puissant quand on sait le maîtriser

Comme je vous l'ai dit plus haut, créer des programmes en mode "console" comme ici, c'est très facile et idéal pour débuter (ce qui n'est pas le cas des programmes en mode fenêtres).  
  
Notez que la console a évolué : elle peut afficher des couleurs, et rien ne vous empêche de mettre une image de fond. Voici une autre capture d'écran de console Linux honteusement pompée sur Internet :-°

  
*La console, ça peut aussi être joli*

**Et sous Windows ? Y'a pas de console ?**

Si, mais elle est un peu... "cachée" on va dire ;)  
Vous pouvez avoir une console en faisant "Démarrer / Accessoires / Invite de commandes", ou bien encore en faisant "Démarrer / Exécuter", et en tapant ensuite "cmd".  
  
Et voici la maaagnifique console de Windows :

  
*La console de Windows*

Si vous êtes sous Windows, sachez donc que c'est dans une fenêtre qui ressemble à ça que nous ferons nos premiers programmes. Si j'ai choisi de commencer par des petits programmes en console, ce n'est pas pour vous ennuyer, bien au contraire ! En commençant par faire des programmes en console, vous apprendrez les bases nécessaires pour ensuite pouvoir créer des fenêtres.  
  
**Soyez donc rassurés** : dès que nous aurons le niveau pour créer des fenêtres, nous verrons comment en faire :)

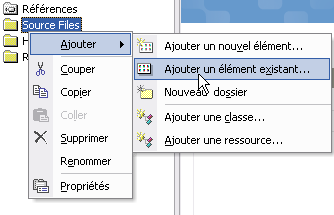
**Un minimum de code**

Pour n'importe quel programme, il faudra taper un minimum de code. Ce code ne fera rien de particulier, mais il est indispensable.  
C'est ce "code minimum" que nous allons découvrir maintenant. Il devrait servir de base pour la plupart de vos programmes en langage C.  
  
Ah oui, je le reprécise quand même au cas où : nous allons maintenant apprendre **le langage C**, comme je vous l'ai dit plus tôt. Tout ce que je vais vous apprendre maintenant, vous le réutiliserez lorsque nous verrons le C++, donc vous avez intérêt à être attentifs tout le temps ;)

**Demandez le code minimal à votre IDE**

Selon l'IDE que vous avez choisi dans le chapitre précédent, la méthode pour créer un nouveau projet n'est pas la même. Reportez-vous à ce chapitre précédent si vous avez oublié comment faire.

* **Sous Dev-C++** : demandez une *console application* que vous appellerez "bonjour". Dev va vous créer le code minimal, vous n'avez rien de plus à faire.
* **Sous Visual C++** : demandez un *projet console Win32* vide que vous appellerez "bonjour".  
  Ajoutez un nouveau fichier à votre projet de type "*Fichier C++ (.cpp)*". Demandez de suite à enregistrer votre fichier dans le répertoire de votre projet sous le nom main.c (et non main.cpp, car l'extension .cpp est plutôt utilisée pour le C++)  
    
  Puis, dans l'onglet "Explorateur de solutions" faites un clic droit sur le dossier "Source Files" et cliquez choisissez "Ajouter / Ajouter un élément existant". On vous demandera d'indiquer des fichiers : vous devrez sélectionner le fichier main.c que vous venez d'enregistrer.

  
*Ajouter un fichier à un projet sous Visual*

Démonstration en images (ou plutôt en vidéo ^^ ) :

[**Ajouter le fichier main.c au projet (200 Ko)**](http://www.siteduzero.com/uploads/fr/ftp/mateo21/vcpp_ajouter_fichier.html)

Bon pour cette fois, la procédure à suivre aura été bien plus simple avec Dev, mais ça ne veut pas dire que Visual est plus compliqué à utiliser hein ;) C'est juste pour la première fois.  
  
Dev a donc généré le minimum de code en langage C dont on a besoin. Le voici :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #include <stdio.h>  #include <stdlib.h>  int main(int argc, char \*argv[])  {    system("PAUSE");  **return** 0;  } |

Si vous êtes sous Visual, copiez-collez ce code source dans votre fichier main.c qui est pour l'instant vide.  
  
Enregistrez le tout. Oui je sais, on n'a encore rien fait, mais enregistrez quand même, c'est une bonne habitude à prendre :p  
Normalement, vous n'avez qu'un seul fichier source appelé *main.c* (le reste c'est des fichiers de projet générés par votre IDE).

**Analysons le code minimal**

Ce code minimal qu'on vient de voir n'est, j'imagine, rien que du charabia pour vous. Et pourtant, moi je vois là un programme console qui s'affiche, qui se met en pause et qui s'éteint.   
Il va falloir apprendre à lire tout ça ^^  
  
Commençons par les 2 premières lignes qui se ressemblent beaucoup :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2 | #include <stdio.h>  #include <stdlib.h> |

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** (un nom compliqué n'est-ce pas ? :p ). Ce sont des lignes qui seront lues par un programme appelé préprocesseur, un programme qui se lance au début de la compilation.  
  
Oui, comme je vous l'ai dit plus tôt, ce qu'on a vu au début n'était qu'un schéma très simplifié de la compilation. Il se passe en réalité plusieurs choses pendant une compilation. On les détaillera plus tard, pour le moment vous avez juste besoin de mettre ces lignes en haut de chacun de vos fichiers.

**Oui mais elles signifient quoi ces lignes ? J'aimerais bien savoir quand même !**

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 sources tout prêts. On verra plus tard qu'on les appelle des **librairies** (ou aussi **bibliothèques** ). En gros, ces fichiers contiennent du code tout prêt qui permet d'afficher du texte à l'écran.

**A noter : le mot anglais est "library" et il se traduit par "bibliothèque". "Librairie" est donc un faux-ami.  
En théorie, on devrait donc dire bibliothèque et non librairie. Mais pour ma part, j'ai pris l'habitude d'écrire librairie (et je ne suis pas le seul ;) ) donc je continuerai à utiliser ce terme. Retenez quand même que la traduction exacte est plutôt "bibliothèque".**

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. Vous voyez la galère dans laquelle on est :p  
  
Bref, les 2 premières lignes incluent les librairies qui vont nous permettre (entre autres) d'afficher du texte à l'écran assez "facilement" :)  
  
Passons à la suite. La suite, c'est tout ça :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int main(int argc, char \*argv[])  {    system("PAUSE");  **return** 0;  } |

Ce que vous voyez là, 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 ! :D  
*(c'est beau la technologie !)*  
  
Sans rentrer dans les détails de la construction d'une fonction (il est trop tôt, on reparlera des fonctions plus tard), analysons quand même ses grandes parties. La première ligne 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. Si vous avez bien suivi, notre fonction main contient 2 lignes :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2 | system("PAUSE");  **return** 0; |

Ces lignes à l'intérieur d'une fonction ont un nom. On les appelle **instructions**(ça en fait du vocabulaire qu'il va falloir retenir ;) ).  
Chaque instruction est une commande à l'ordinateur. Chacune de ces lignes demande à l'ordinateur de faire quelque chose de précis.  
  
Comme je vous l'ai dit un peu plus haut, en regroupant intelligemment (c'est le travail du programmeur) les instructions dans des fonctions, on crée si on veut des "*bouts de programmes tout prêts*". En utilisant les bonnes instructions, rien ne nous empêcherait donc de créer une fonction "ouvrir\_fichier" comme je vous l'ai expliqué tout à l'heure, ou encore une fonction "avancer\_personnage" dans un jeu vidéo par exemple ^^  
  
Un programme, ce n'est en fait au bout du compte rien d'autre qu'une série d'instructions : "fais ceci" "fais cela". Vous donnez des ordres à votre ordinateur et il les exécute (du moins si vous l'avez bien dressé :-° )

**TRES IMPORTANT : toute instruction se termine O-BLI-GA-TOI-RE-MENT par un point-virgule " ; ". C'est d'ailleurs comme ça qu'on reconnaît ce qui est une instruction et ce qui n'en est pas une. Si vous oubliez de mettre un point-virgule à la fin d'une instruction, votre programme ne compilera pas !**

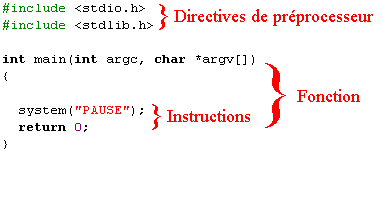
La première ligne :  
system("PAUSE");   
demande à l'ordinateur de mettre en pause le programme (c'est fou, on l'aurait presque deviné tout seul :p ). Quand votre programme arrivera à cette ligne, il va afficher un message à l'écran : " *Appuyez sur une touche pour continuer* " et va attendre que vous appuyiez sur n'importe quelle touche avant de passer à l'instruction suivante.

**L'instruction system("PAUSE") ne fonctionne que sous Windows. Elle est en fait utilisée par Dev C++ pour mettre en pause le programme juste avant qu'il ne se termine.**

Les IDE un peu plus intelligents, comme Code::Blocks, rajoutent une instruction du même type automatiquement à la compilation. Dans ce cas, l'instruction system("PAUSE");est inutile : vous pouvez simplement l'enlever. Vu qu'au début de ce cours je travaillerai principalement sous Dev-C++, vous devriez voir souvent cette instruction. Plus loin dans le cours j'évoluerai vers Code::Blocks et Visual C++ qui sont des IDE un peu plus poussés.  
Si vous êtes sous un autre système d'exploitation que Windows (Linux ou Mac OS), vous lancerez votre programme directement depuis la console et n'aurez pas besoin d'utiliser cette instruction. Si vous tenez à mettre en pause votre programme avant la fin, vous pouvez remplacer l'instruction system("PAUSE"); par l'instruction getchar();(qu'il faudra peut-être écrire 2 fois pour que ça marche :p )  
Passons à l'instruction suivante :  
return 0;  
Bon ben ça en gros, ça veut dire que c'est fini :p (eh oui déjà ^^). Cette ligne indique qu'on arrive à la fin de notre fonction *main*et demande de renvoyer la valeur 0.

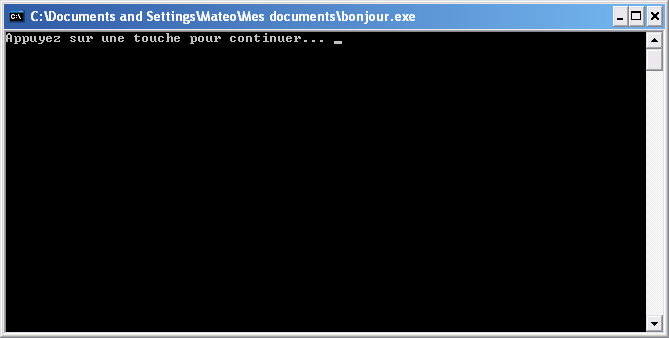
Hein ? Pourquoi mon programme renverrait-il le nombre 0 ?

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 :)  
  
Et voilà ! On vient de détailler un peu le fonctionnement du code minimal.  
  
Certes, on n'a pas vraiment tout vu en profondeur, et vous devez avoir quelques questions en suspens. Soyez rassurés : toutes vos questions trouveront une réponse petit à petit. Je ne peux pas tout vous divulguer d'un coup, sinon c'est l'embrouille assurée ^^  
D'ailleurs, en parlant d'embrouille, ça va vous suivez toujours ? :)  
Si tel n'est pas le cas, rien ne presse. Ne vous forcez pas à lire la suite. Faites une pause et relisez ce début de chapitre à tête reposée. Tout ce que je viens de vous apprendre est fondamental, surtout si vous voulez être sûrs de pouvoir suivre après :)  
  
Tenez, d'ailleurs comme je suis de bonne humeur je vous fais un schéma qui récapitule le vocabulaire qu'on vient d'apprendre :)

  
*Le vocabulaire du programme minimal*

**Testons notre programme**

Tester devrait aller vite. Tout ce que vous avez à faire c'est compiler le projet, puis l'exécuter (cliquez sur "Compiler & Exécuter" sous Dev).  
Si vous ne l'avez pas encore fait, on vous demandera d'enregistrer les fichiers. Faites-le.  
  
Après un temps d'attente insupportable (la compilation :p ), votre premier programme va apparaître sous vos yeux totalement envahis de bonheur o_O

  
*Votre premier programme !*

Comme indiqué à l'écran, appuyez sur une touche. Votre programme s'arrête alors.  
Oui je sais c'est nul, c'est moche, c'est tout ce que vous voulez ^^  
Mais bon, quand même ! C'est un premier programme, un instant dont vous vous souviendrez toute votre vie :-°  
  
... Non ?  
...  
  
Bon, avant que vous me fassiez déjà une première déprime, je propose qu'on passe à la suite sans plus tarder :p

**Ecrire un message à l'écran**

A partir de maintenant, on va ajouter nous-mêmes du code dans ce programme minimal.   
Votre mission, si vous l'acceptez : afficher le message "Bonjour" à l'écran.  
  
Comme tout à l'heure, une console doit s'ouvrir. Le message "Bonjour" doit s'afficher dans la console.

Comment fait-on pour écrire du texte dans la console ?

On va devoir rajouter une ligne dans la fonction main. Vous vous rappelez que ces lignes ont un nom particulier n'est-ce pas ? ^^  
On les appelle **des instructions**. On va donc rajouter l'instruction qui commande à l'ordinateur : "*Affiche-moi un message à l'écran*"  
  
Cette instruction a un nom, elle s'appelle *printf*(retenez-le !).  
En fait, printf est une fonction déjà écrite par d'autres programmeurs avant vous.

Cette fonction, où se trouve-t-elle ? Moi je ne vois que la fonction *main*!

Vous vous souvenez de ces 2 lignes ?  
#include <stdio.h>  
#include <stdlib.h>  
  
Je vous avais dit qu'elles permettaient d'ajouter des librairies dans votre programme.  
Les librairies sont en fait des fichiers avec pleins de fonctions toutes prêtes à l'intérieur. Ces fichiers-là (stdio.h et stdlib.h) contiennent la plupart des fonctions de base dont on a besoin dans un programme. stdio.h en particulier contient des fonctions permettant d'afficher des choses à l'écran (comme *printf*) mais aussi de demander à l'utilisateur de taper quelque chose (ce sont des fonctions que l'on verra plus tard).

**Dis Bonjour au Monsieur**

Dans notre fonction *main*, on va faire appel à la fonction *printf*.  
C'est une fonction qui en appelle une autre (ici, main appelle *printf*). Vous allez voir que c'est tout le temps comme ça que ça se passe en langage C ;)  
Donc, pour faire appel à une fonction, c'est très simple : il suffit d'écrire son nom. Ecrivez donc *printf*sur une ligne au tout début de la fonction main (avant lesystem("PAUSE")).  
C'est bien, mais on n'est pas encore tirés d'affaire ;) Il faut indiquer quoi écrire à l'écran. Pour faire ça, il va falloir donner à la fonction *printf*le texte à afficher. Pour ce faire, ouvrez des parenthèses après le mot printf. Puis, ouvrez des guillemets à l'intérieur des parenthèses. Enfin, tapez le texte à afficher entre les guillemets.  
Dans notre cas, on va donc taper très exactement :  
printf("Bonjour");  
J'espère que vous n'avez pas oublié le point-virgule à la fin, je vous rappelle que c'est très important ! Cela permet d'indiquer que l'instruction s'arrête là.

Voici le code source que vous devriez avoir sous les yeux :  
  
**Code : C**

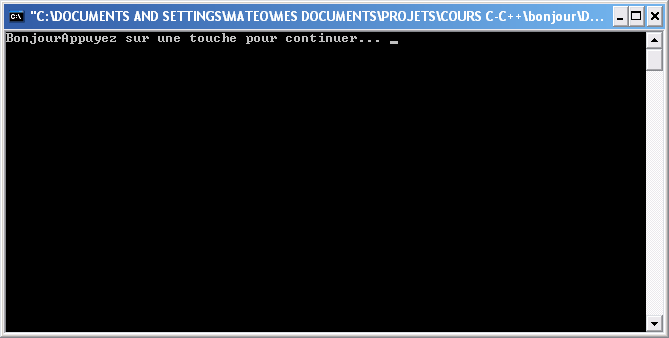
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #include <stdio.h>  #include <stdlib.h>  int main(int argc, char \*argv[])  {  printf("Bonjour");  system("PAUSE");  **return** 0;  } |

On a donc 3 instructions qui commandent dans l'ordre à l'ordinateur :

1. Affiche "Bonjour" à l'écran.
2. Met le programme en pause, affiche le message "Appuyez sur une touche pour continuer" et attend qu'on appuie sur une touche avant de passer à l'instruction suivante.
3. La fonction *main*est terminée, renvoie 0. Le programme s'arrête alors.

**A quoi ça sert de mettre le programme en pause ? On ne pourrait pas enlever l'instruction system("PAUSE") ?**

Si, bien sûr qu'on pourrait ^^ Testez sans cette instruction et vous verrez.  
Le programme ne se met pas en pause. En clair, il affiche le message "Bonjour" et puis s'arrête. Du coup, la fenêtre de la console apparaît et disparaît à la vitesse de l'éclair, vous n'avez pas le temps de lire ce qui est écrit à l'écran.  
Stupide, isn't it ? :p  
Notez qu'avec certains IDE, comme je vous l'ai dit plus tôt, il se peut que la pause soit faite automatiquement. Dans ce cas, l'instruction system("PAUSE") est inutile et vous pouvez l'enlever ;)  
On va donc tester le programme avec une pause, ce qui devrait nous afficher :

  
*Un programme poli qui dit Bonjour... Enfin presque*

**Ouiiiin ! J'arrive même pas à dire bonjour correctement, y'a tout qui s'écrit sur la même ligne :'(**

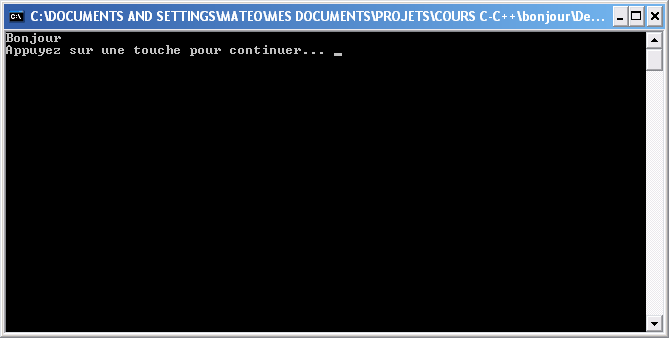
Allons allons, ce n'est pas bien grave, on va apprendre à corriger ça tout de suite :)  
Une des solutions pour rendre notre programme plus présentable serait de faire un retour à la ligne après "Bonjour" (comme si on appuyait sur la touche "Entrée" quoi ^^ )  
Mais bien sûr, ce serait trop simple de taper "Entrée" dans notre code source pour qu'une entrée soit effectuée à l'écran ! Il va falloir utiliser ce qu'on appelle des caractères spéciaux...

**Les caractères spéciaux**

Les caractères spéciaux sont des lettres spéciales qui permettent d'indiquer qu'on veut aller à la ligne, faire une tabulation etc...  
Les caractères spéciaux sont faciles à reconnaître : c'est un ensemble de 2 caractères. Le premier d'entre eux est toujours un anti-slash ( \ ), et le second un nombre ou une lettre. Voici 2 caractères spéciaux courants que vous aurez probablement besoin d'utiliser, ainsi que leur signification :

* **\n** : retour à la ligne (= "Entrée")
* **\t** : tabulation

Dans notre cas, pour faire une entrée, il suffit de taper \n pour créer un retour à la ligne.  
Si je veux donc faire un retour à la ligne juste après le mot Bonjour, je devrai taper :  
  
printf("Bonjour\n");  
  
Votre ordinateur comprend qu'il doit afficher "Bonjour" suivi d'un retour à la ligne.  
  
Votre programme va maintenant avoir une tête un peu plus présentable :)

  
*Ah, voilà un Bonjour un peu plus présentable ! :)*

C'est mieux quand même :D

**Vous pouvez écrire à la suite du \n sans aucun problème. Tout ce que vous écrirez à la suite du \n sera placé sur la deuxième ligne. Vous pourriez donc vous entraîner à écrire :  
printf("Bonjour\nAu Revoir\n");  
Cela affichera "Bonjour" sur la première ligne et "Au revoir" sur la ligne suivante.**

**Le syndrome de Gérard**

**Bonjour, je m'appelle Gérard et j'ai voulu essayer de modifier votre programme pour qu'il me dise "Bonjour Gérard". Seulement voilà, j'ai l'impression que l'accent de Gérard ne s'affiche pas correctement... Que faire ?**

Tout d'abord, bonjour Gérard ^^  
C'est une question très intéressante que vous nous posez là. Je tiens en premier lieu à vous féliciter pour votre esprit d'initiative, c'est très bien d'avoir eu l'idée de modifier un peu le programme. C'est en "bidouillant" les programmes que je vous donne que vous allez en apprendre le plus. Ne vous contentez pas de ce que vous lisez, essayez un peu vos propres modifications des programmes que nous voyons ensemble !  
  
Bien, maintenant pour répondre à la question de notre ami Gérard, j'ai une bien triste nouvelle à vous annoncer : la console de Windows ne gère pas les accents :(  
Par contre la console de Linux oui :)  
  
A partir de là vous avez 2 solutions :

* **Passer à Linux**. C'est une solution un peu radicale et il me faudrait tout un cours entier pour vous expliquer comment vous servir de Linux. Si vous n'avez pas le niveau, oubliez cette possibilité pour le moment :p
* **Ne pas utiliser d'accents**. C'est malheureusement la solution que vous risquez de choisir. La console de Windows a ses défauts que voulez-vous. Il va vous falloir prendre l'habitude d'écrire sans accents. Bien entendu, comme plus tard vous ferez probablement des programmes avec des fenêtres, vous ne devriez pas avoir ce problème-là ;)

Vous devrez donc écrire : printf("Bonjour Gerard\n");  
  
On remercie notre ami Gérard pour nous avoir soulevé ce problème :)  
*ps : si d'aventure vous vous appeliez Gérard, sachez que je n'ai rien contre ce prénom :p C'est simplement le premier prénom avec un accent qui m'est passé par la tête :D  
Et puis bon, il faut toujours que quelqu'un prenne pour les autres, que voulez-vous :lol:*

**Les commentaires, c'est très utile !**

Avant de terminer ce premier chapitre de "véritable" programmation, je dois absolument vous montrer un truc génial qu'on appelle **les commentaires**. Quel que soit le langage de programmation, on a la possibilité d'ajouter des commentaires à son code. Le langage C n'échappe pas à la règle.  
  
Qu'est-ce que ça veut dire "commenter" ?  
Cela signifie : taper du texte au milieu de votre programme pour indiquer ce qu'il fait, à quoi sert telle ligne de code etc. C'est vraiment quelque chose d'indispensable car, même en étant un génie de la programmation, on a besoin de faire quelques annotations par-ci par-là. Cela permet :

* De vous retrouver au milieu d'un de vos codes sources plus tard. On ne dirait pas comme ça, mais on oublie vite comment fonctionnent les programmes qu'on a écrit ^^ Si vous faites une pause ne serait-ce que de quelques jours, vous aurez besoin de vous aider de vos propres commentaires pour vous retrouver dans un gros code.
* Si vous donnez votre projet à quelqu'un d'autre (qui ne connaît pas à priori votre code source), cela lui permettra de se familiariser avec bien plus rapidement.
* Enfin, ça va me permettre à moi de rajouter des annotations dans les codes sources de ce cours. Cela me permettra de mieux vous expliquer à quoi peut servir telle ou telle ligne de code.

Il y a plusieurs manières de rajouter un commentaire. Tout dépend de la longueur du commentaire que vous voulez écrire :

* Votre commentaire est **court**: il tient sur une seule ligne, il ne fait que quelques mots. Dans ce cas, vous devez taper un double slash (//) suivi de votre commentaire. Par exemple :  
  **Code : C**

|  |  |
| --- | --- |
| 1 | *// Ceci est un commentaire* |

* Vous pouvez aussi bien écrire un commentaire seul sur sa ligne, ou bien à droite d'une instruction. C'est d'ailleurs quelque chose de très pratique car ainsi on sait que le commentaire sert à indiquer à quoi sert la ligne sur laquelle il est. Exemple :  
  **Code : C**

|  |  |
| --- | --- |
| 1 | printf("Bonjour"); *// Cette instruction affiche Bonjour à l'écran* |

* Notez que ce type de commentaire a normalement été introduit par le langage C++, mais vous n'aurez pas de problème en l'utilisant pour un programme en langage C (sauf si vous êtes un puriste  )
* Votre commentaire est **long**: vous avez plein de choses à raconter, vous avez besoin d'écrire plusieurs phrases qui tiennent sur plusieurs lignes. Dans ce cas, vous devez taper un code qui signifie "début de commentaire" et un autre code qui signifie "fin de commentaire" :
  + Pour indiquer le *début du commentaire* : tapez un slash suivi d'une étoile (/\*)
  + Pour indiquer la *fin du commentaire* : tapez une étoile suivie d'un slash (\*/)

Vous écrirez donc par exemple :  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3 | */\* Ceci est*  *Un commentaire*  *Sur plusieurs lignes \*/* |

Reprenons notre code source qui écrit "Bonjour", et ajoutons-lui quelques commentaires juste pour s'entraîner :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | */\**  *Ci-dessous, ce sont des directives de préprocesseur.*  *Ces lignes permettent d'ajouter des fichiers au projet, fichiers que l'on appelle "librairies".*  *Grâce à ces librairies, on disposera de fonctions toutes prêtes pour afficher par exemple un message à l'écran*  *\*/*  #include <stdio.h>  #include <stdlib.h>  */\**  *Ci-dessous, vous avez la fonction principale du programme, appelée "main". C'est par cette fonction que tous les programmes commencent.*  *Ici, ma fonction se contente d'afficher "Bonjour" à l'écran, met en pause le programme puis s'arrête*  *\*/*  int main(int argc, char \*argv[])  {  printf("Bonjour"); *// Cette instruction affiche Bonjour à l'écran*  system("PAUSE"); *// Le programme se met en pause*  return 0; *// Le programme renvoie le nombre 0 puis s'arrête*  } |

Voilà ce que donnerait notre programme avec quelques commentaires ;)  
Oui, il a l'air d'être plus gros, mais en fait c'est le même que tout à l'heure. Lors de la compilation, tous les commentaires seront ignorés. Ces commentaires n'apparaîtront pas dans le programme final, ils servent seulement aux programmeurs.  
  
Normalement, on ne commente pas chaque ligne du programme. J'ai dit (et je le redirai) que c'était important de mettre des commentaires dans un code source, mais il faut savoir doser : commenter chaque ligne ne servira la plupart du temps à rien. A force, vous saurez que le *printf*permet d'afficher un message à l'écran, pas besoin de l'indiquer à chaque fois ;)  
  
Le mieux est de commenter plusieurs lignes à la fois, c'est-à-dire d'indiquer à quoi sert une série d'instructions histoire d'avoir une idée. Après, si le programmeur veut se pencher plus en détail dans ces instructions, il est assez intelligent pour y arriver tout seul.  
  
**Retenez donc** : les commentaires doivent guider le programmeur dans son code source, lui permettre de se repérer. Essayez de commenter un ensemble de lignes plutôt que toutes les lignes une par une.  
  
Et pour finir sur une petite touche culturelle, voici une citation tirée de chez IBM :  
  
**Citation : Règle de la maison IBM**

Si après avoir lu uniquement les commentaires d'un programme vous n'en comprenez pas le fonctionnement, jetez le tout !

## Une affaire de mémoire

Ce que je vais vous apprendre dans ce chapitre a donc un rapport direct avec la mémoire de votre ordinateur.  
  
Tout être humain normalement constitué a une mémoire. Eh bien c'est pareil pour un ordinateur... à un détail près : un ordinateur a plusieurs types de mémoire !

Pourquoi un ordinateur aurait-il plusieurs types de mémoire ? Une seule mémoire aurait suffit, non ?

Non, en fait le problème c'est qu'on a besoin d'avoir une mémoire à la fois **rapide**(pour récupérer une information très vite) et **importante**(pour stocker beaucoup de choses). Or, vous allez rire, mais jusqu'ici nous avons été infichus de créer une mémoire qui soit à la fois très rapide et importante. Plus exactement, la mémoire rapide coûte cher, donc on n'en fait qu'en petites quantités.  
Du coup, pour nous arranger, nous avons dû doter les ordinateurs de mémoires très rapides mais pas importantes, et de mémoires importantes mais pas très rapides (vous suivez toujours ? ^^ )

### Les différents types de mémoire

Pour vous donner une idée, voici les différents types de mémoire existant dans un ordinateur, de la plus rapide à la plus lente :

1. Les registres : une mémoire ultrarapide située directement dans le processeur.
2. La mémoire cache : elle fait le lien entre les registres et la mémoire vive.
3. La mémoire vive : c'est la mémoire avec laquelle nous allons travailler le plus souvent.
4. Le disque dur : que vous connaissez sûrement, c'est là qu'on enregistre les fichiers.

Comme je vous l'ai dit, j'ai classé les mémoires de la plus rapide (les registres) à la plus lente (le disque dur). Si vous avez bien suivi, vous avez compris aussi que la mémoire la plus rapide était la plus petite, et la plus lente la plus grosse.  
Les registres sont donc à peine capables de retenir quelques nombres, tandis que le disque dur peut stocker de très gros fichiers.

Quand je dis qu'une mémoire est "lente", c'est à l'échelle de votre ordinateur bien sûr. Eh oui, pour un ordinateur 8 millisecondes pour accéder au disque dur c'est déjà trop long !

Que faut-il retenir dans tout ça ?  
En fait, c'est pour vous situer un peu. Vous savez désormais qu'en programmation, on va surtout travailler avec la mémoire vive. On verra aussi comment lire et écrire sur le disque dur, pour lire et créer des fichiers (mais on ne le fera que plus tard). Quant à la mémoire cache et aux registres, on n'y touchera pas du tout ! C'est votre ordinateur qui s'en occupe.

Dans des langages très bas niveau, comme l'assembleur (abrégé "ASM"), on travaille au contraire plutôt directement avec les registres. Je l'ai fait, et je peux vous dire que faire une simple multiplication dans ce langage est un véritable parcours du combattant ! Heureusement, en langage C (et dans la plupart des autres langages de programmation), c'est beaucoup plus facile.

Il faut ajouter une dernière chose très importante : seul le disque dur retient tout le temps les informations qu'il contient. Toutes les autres mémoires (registres, mémoire cache, mémoire vive) sont des mémoires temporaires : lorsque vous éteignez votre ordinateur ces mémoires se vident !  
  
Heureusement, lorsque vous rallumerez l'ordinateur, votre disque dur sera toujours là pour rappeler à votre ordinateur qui il est ^^

### La mémoire vive en photos

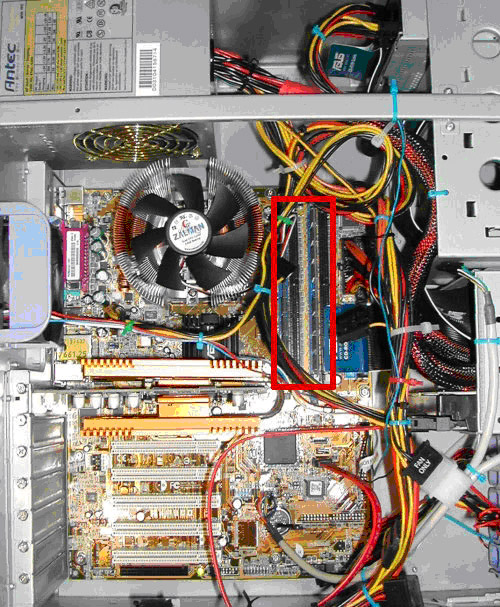
Vu qu'on va travailler pendant un moment avec la mémoire vive, je pense qu'il serait bien de vous la présenter ;)  
  
On va y aller par zooms successifs. Ca, c'est votre ordinateur :



Vous reconnaissez le clavier, la souris, l'écran et l'unité centrale (la tour).   
Intéressons-nous maintenant à l'unité centrale, le cœur de votre ordinateur qui contient toutes les mémoires :



Ce qui nous intéresse, c'est ce qu'il y a à l'intérieur de l'unité centrale, si on l'ouvre :



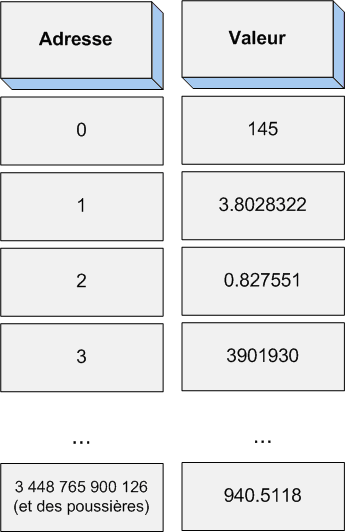
C'est un joyeux petit bazar ^^  
Rassurez-vous, je ne vous demanderai pas de savoir comment tout cela fonctionne. Je veux juste que vous sachiez où se trouve la mémoire vive là-dedans. Je vous l'ai encadrée en rouge.  
Je n'ai pas indiqué les autres mémoires (registres et mémoire cache) car de toute façon elles sont bien trop petites pour être visibles à l'oeil nu ;)  
  
Voici à quoi ressemble une barrette de mémoire vive de plus près :

[](http://www.siteduzero.com/uploads/fr/files/7001_8000/7208.jpg)  
*Cliquez sur l'image si vous voulez voir en plus grand*

La mémoire vive est aussi appelée **RAM**, ne vous étonnez donc pas si par la suite j'utilise plutôt le mot RAM qui est un peu plus court.

### Le schéma de la mémoire vive

En photographiant de plus près la mémoire vive, on n'y verrait pas grand-chose. Pourtant, il est très important de savoir comment ça fonctionne à l'intérieur. C'est d'ailleurs là que je veux en venir depuis tout à l'heure :D  
  
Je vais vous faire un schéma du fonctionnement de la mémoire vive. Il est ultra-simplifié (comme mes schémas de compilation :p ), mais c'est parce que nous n'avons pas besoin de trop de détails. Si vous retenez ce schéma déjà, ça sera très bien ^^



Comme vous le voyez, il faut en gros distinguer 2 colonnes :

* Il y a les **adresses** : une adresse est un nombre qui permet à l'ordinateur de se repérer dans la mémoire vive. On commence à l'adresse 0 (au tout début de la mémoire) et on finit à l'adresse 3 448 765 900 126 et des poussières... Euh, en fait je ne connais pas le nombre d'adresses qu'il y a dans la RAM, je sais juste qu'il y en a beaucoup.  
  En plus ça dépend de la quantité de mémoire vive que vous avez. Plus vous avez de mémoire vive, plus il y a d'adresses, donc plus on peut stocker de choses ;)
* A chaque adresse, on peut stocker une **valeur** (un nombre) : votre ordinateur stocke dans la mémoire vive ces nombres pour pouvoir s'en souvenir par la suite. On ne peut stocker qu'un nombre par adresse !

Notre RAM ne peut stocker que des nombres.

Mais alors, comment fait-on pour retenir des mots ?

Bonne question. En fait, même les lettres ne sont que des nombres pour l'ordinateur ! Une phrase est une simple succession de nombres !  
Il existe un tableau qui fait la correspondance entre les nombres et les lettres. C'est un tableau qui dit par exemple : *le nombre 67 correspond à la lettre Y*. Je ne rentre pas dans les détails, on aura l'occasion de reparler de cela plus loin dans le cours.  
  
Revenons à notre schéma. Les choses sont en fait très simples : si l'ordinateur veut retenir le nombre 5 (qui pourrait être le nombre de vies qu'il reste au joueur), il le met quelque part en mémoire où il y a de la place et note l'adresse correspondante (par exemple 3 062 199 902)  
Plus tard, lorsqu'il veut savoir à nouveau quel est ce nombre, il va chercher à la "case" mémoire n°3 062 199 902 ce qu'il y a, et il trouve la valeur... 5 !  
  
Voilà en gros comment ça fonctionne. C'est peut-être un peu flou pour le moment (quel intérêt de stocker un nombre s'il faut à la place retenir l'adresse ?) mais tout va rapidement prendre du sens dans la suite de ce chapitre je vous le promets :)

## Déclarer une variable

Croyez-moi, cette petite introduction sur la mémoire va nous être plus utile que vous ne le pensez.  
Maintenant que vous savez ce qu'il faut, on peut retourner programmer :D  
  
Alors une variable, c'est quoi ?  
Eh bien c'est une petite information temporaire qu'on stocke dans la RAM. Tout simplement.  
On dit qu'elle est "variable" car c'est une valeur qui peut changer pendant le déroulement du programme. Par exemple, notre nombre 5 de tout à l'heure (le nombre de vies restant au joueur) risque de diminuer au fil du temps. Si ce nombre atteint 0, on saura que le joueur a perdu.  
  
Nos programmes, vous allez le voir, sont remplis de variables. Vous allez en voir partout, à toutes les sauces ^^  
  
En langage C, une variable est constituée de 2 choses :

* Elle a une **valeur**: c'est le nombre qu'elle stocke, par exemple 5.
* Elle a un **nom**: c'est ce qui permet de la reconnaître. En programmant en C, on n'aura pas à retenir l'adresse mémoire (ouf !), on va juste indiquer des noms de variables à la place. C'est le compilateur qui fera la conversion entre le nom et l'adresse. Voilà déjà un souci en moins.

### Donner un nom à ses variables

En langage C, chaque variable doit donc avoir un nom. Pour notre fameuse variable qui retient le nombre de vies, on aimerait bien l'appeler "Nombre de vies" ou quelque chose du genre.  
  
Hélas, il y a quelques contraintes. Vous ne pouvez pas appeler une variable n'importe comment :

* Il ne peut y avoir que des lettres minuscules et majuscules et des chiffres (abcABC012...).
* Votre nom de variable doit commencer par une lettre.
* Les espaces sont interdits. A 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 (éàê etc).

Enfin, et c'est très important à savoir, le langage C (comme le 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 3 variables différentes en langage C, même si pour nous ça a l'air de signifier la même chose !  
  
Voici quelques exemples de noms de variable corrects : nombreDeVies, nombre\_de\_vies, 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 ^^

Quoi que vous fassiez, faites en sorte de donner des noms clairs à vos variables. On aurait pu abréger *nombreDeVies*, en l'écrivant par exemple *ndv*. 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.

### Les types de variables

Notre ordinateur, vous pourrez le constater, n'est en fait rien d'autre qu'une (très grosse) machine à calculer. Il ne sait traiter que des nombres.  
  
Oui mais voilà, j'ai un scoop ! Il existe **plusieurs types de nombres** !  
  
Par exemple, il y a les nombres entiers positifs :

* 45
* 398
* 7650

Mais il y a aussi des nombres décimaux, c'est-à-dire des nombres à virgules :

* 75,909
* 1,7741
* 9810,7

En plus de ça, il y a aussi des nombres entiers négatifs :

* -87
* -916

... Et des nombres négatifs décimaux !

* -76,9
* -100,11

Votre pauvre ordinateur a besoin d'aide ! Lorsque vous lui demandez de stocker un nombre, vous devez dire de quel type il est. Ce n'est pas vraiment qu'il ne soit pas capable de le reconnaître tout seul, mais... Ca l'aide beaucoup à s'organiser, et à faire en sorte de ne pas prendre trop de mémoire pour rien.  
  
Lorsque vous créez une variable, vous allez donc devoir **indiquer son type**.  
Voici les principaux types de variables existant en langage C (il y en aura un autre qui fera son apparition en C++) :

| **Nom du type** | **Nombres stockables** |
| --- | --- |
| char | -128 à 127 |
| int | -2 147 483 648 à 2 147 483 647 |
| long | -2 147 483 648 à 2 147 483 647 |
| float | -3.4 x 10 puissance 38 à 3.4 x 10 puissance 38 |
| double | -1.7 x 10 puissance 308 à 1.7 x 10 puissance 308 |

*(Je suis loin d'avoir mis tous les types, mais j'ai conservé les principaux ;) )*  
  
Les 3 premiers types permettent de stocker des nombres entiers (1, 2, 3, 4...)  
Les 2 derniers permettent de stocker des nombres décimaux (13.8, 16.911...)

Les types float et double permettent de stocker des nombres décimaux extrêmement grands.  
Si vous ne connaissez pas les puissances de 10, dites-vous par exemple que le type double permet de stocker le nombre 1 suivi de 308 zéros derrière !  
C'est-à-dire : 10000000000000000000000000000000000000000000000000000..... (je ne vais quand même pas écrire 308 zéros pour vous :lol: )

Vous remarquerez qu'int et un long ont l'air identiques. Avant ce n'était pas le cas (un int était plus petit qu'un long), mais aujourd'hui les mémoires ont évolué et on a assez de place pour stocker des grands nombres, donc on se moque un peu de la différence entre un int et un long. Le langage C "conserve" tous ces types pour des raisons de compatibilité, même si certains sont un peu de trop.  
En pratique, j'utilise principalement **char, long et double**.

Vous verrez que la plupart du temps on manipule des nombres entiers (tant mieux, parce que c'est plus facile à utiliser ^^ )

Attention avec les nombres décimaux ! Votre ordinateur ne connaît pas la virgule, il utilise le point. Vous ne devez donc pas écrire 54,9 mais plutôt 54.9 !

Ce n'est pas tout ! Pour les types stockant des entiers (char, int, long...), il existe d'autres types dits "unsigned" (non signés) qui eux ne peuvent stocker que des nombres positifs. Pour les utiliser, il suffit d'écrire le mot "unsigned" devant le type :

|  |  |
| --- | --- |
| unsigned char | 0 à 255 |
| unsigned int | 0 à 4 294 967 295 |
| unsigned long | 0 à 4 294 967 295 |

Comme vous le voyez, les unsigned sont des types qui ont le défaut de ne pas pouvoir stocker de nombre négatifs, mais qui ont l'avantage de pouvoir stocker des nombres 2 fois plus grands (char s'arrête à 128, tandis que unsigned char s'arrête à 255 par exemple).

Pourquoi avoir créé 3 types pour les nombres entiers ? Un seul type aurait été suffisant non ?

Oui, mais on a créé plusieurs types à l'origine pour économiser de la mémoire. Ainsi, quand on dit à l'ordinateur qu'on a besoin d'une variable de type "char", on prend moins d'espace en mémoire que si on avait demandé une variable de type "long".  
  
Toutefois, c'était utile surtout à l'époque où la mémoire était limitée. Aujourd'hui, nos ordinateurs ont largement assez de mémoire vive pour que ça ne soit plus vraiment un problème. Il ne sera donc pas utile de se prendre la tête pendant des heures sur le choix d'un type. Si vous ne savez pas si votre variable risque de prendre une grosse valeur, mettez long.  
Et je dis ça sérieusement. Ne vous prenez pas trop la tête sur le choix d'un type pour le moment ;)  
  
En résumé, on fera surtout la distinction entre nombres entiers et décimaux :

* Pour un nombre entier, on utilisera le plus souvent long.
* Pour un nombre décimal, on utilisera généralement double.

### Déclarer une variable

On y arrive. Maintenant, créez un nouveau projet console que vous appellerez "variables".  
On va voir comment déclarer une variable, c'est-à-dire **demander 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 d'indiquer dans l'ordre :

1. Le type de la variable que l'on veut créer
2. Tapez espace
3. Indiquez le nom que vous voulez donner à la variable
4. Et enfin n'oubliez pas le point-virgule ;)

Par exemple, si je veux créer ma variable nombreDeVies de type long, je dois taper la ligne suivante :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | long nombreDeVies; |

Et c'est tout ! :D  
  
Quelques autres exemples stupides pour la forme :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3 | long noteDeMaths;  double sommeArgentRecue;  unsigned long nombreDeZerosEnTrainDeLireUnNomDeVariableUnPeuLong; |

Bon bref, vous avez compris le principe je pense :D  
  
Ce qu'on fait là s'appelle une **déclaration de variable** (un vocabulaire à retenir là ^^ )  
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*), vous allez déclarer la variable comme ceci :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | #include <stdio.h>  #include <stdlib.h>    int main(int argc, char \*argv[])  {  *// Début de la fonction*  long nombreDeVies;    system("PAUSE");  **return** 0;  *// Fin de la fonction*  } |

Si vous lancez ce programme, vous constaterez avec stupeur... qu'il ne fait rien o_O

#### Quelques explications

Alors, avant que vous ne m'étrangliez en croyant que je vous mène en bateau depuis tout à l'heure, laissez-moi juste dire une chose pour ma défense :p  
  
En fait, il se passe des choses, mais vous ne les voyez pas. Lorsque le programme arrive à la ligne de la déclaration de variable, il demande bien gentiment à l'ordinateur s'il peut utiliser un peu d'espace dans la mémoire vive.  
Si tout va bien, l'ordinateur répond "Oui bien sûr, fais comme chez toi". Généralement, cela se passe sans problème.

Le seul problème qu'il pourrait y avoir, c'est qu'il n'y ait plus de place en mémoire... Mais cela arrive rarement heureusement, car pour remplir toute la mémoire rien qu'avec des long il faut vraiment être un bourrin de première ^^

Soyez sans craintes donc, vos variables devraient normalement être créées sans souci.

Une petite astuce à connaître : 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 variable par des virgules sur la même ligne :  
**Code : C**

|  |  |
| --- | --- |
| 1 | long nombreDeVies, niveau, ageDuJoueur; |

Cela créera 3 variables long appelées nombreDeVies, niveau et ageDuJoueur.

Et maintenant ?  
Maintenant qu'on a créé notre variable, on va pouvoir lui donner une valeur :)

### 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 nombreDeVies, il suffit de procéder comme ceci :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | nombreDeVies = 5; |

Rien de plus à faire. Vous indiquez le nom de la variable, un signe égal, puis la valeur que vous voulez mettre dedans.  
Ici, on vient de donner la valeur 5 à la variable nombreDeVies.  
Notre programme complet ressemble donc à ceci :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | #include <stdio.h>  #include <stdlib.h>    int main(int argc, char \*argv[])  {  long nombreDeVies;  nombreDeVies = 5;    system("PAUSE");  **return** 0;  } |

Là encore, rien ne s'affiche à l'écran, tout se passe dans la mémoire.  
Quelque part dans les tréfonds de votre ordinateur, une petite case de mémoire vient de prendre la valeur 5. C'est pas magnifique ça ? :)  
Pour un peu on en pleurerait ^^  
  
On peut s'amuser si on veut à changer la valeur par la suite :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | long nombreDeVies;  nombreDeVies = 5;  nombreDeVies = 4;  nombreDeVies = 3; |

Dans cet exemple, la variable va prendre d'abord la valeur 5, puis 4, et enfin 3. Comme votre ordinateur est très rapide, tout cela se passe extrêmement vite. Vous n'avez pas le temps de cligner des yeux que votre variable vient de prendre les valeurs 5, 4 et 3... et ça y est votre programme est fini :p

### La valeur d'une nouvelle variable

Voici une question très importante que je veux vous soumettre :

Quand on déclare une variable, quelle valeur a-t-elle au départ ?

En effet, quand l'ordinateur lit cette ligne :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | long nombreDeVies; |

Il réserve un petit emplacement en mémoire, d'accord. Mais quelle est la valeur de la variable à ce moment-là ? Y a-t-il une valeur par défaut (par exemple 0) ?  
  
Eh bien, accrochez-vous : la réponse est non. Non non et non, il n'y a pas de valeur par défaut. En fait, l'emplacement est réservé mais la valeur ne change pas. On n'efface pas ce qui se trouve dans la "case mémoire". Du coup, votre variable prend la valeur qui se trouvait là avant dans la mémoire, et cette valeur peut être n'importe quoi !  
  
Si cette zone de la mémoire n'a jamais été modifiée, la valeur est peut-être 0. Mais vous n'en êtes pas sûrs, il pourrait très bien y avoir le nombre 363 ou 18 à la place, c'est-à-dire un reste d'un vieux programme qui est passé par là avant !  
Il faut donc faire très attention à ça si on veut éviter des problèmes par la suite. Le mieux est d'initialiser la variable dès qu'on la déclare. En C, c'est tout à fait possible. En gros, ça consiste à combiner la déclaration et l'affectation d'une variable dans la même instruction :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | long nombreDeVies = 5; |

Ici, la variable nombreDeVies est déclarée et elle prend tout de suite la valeur 5.  
L'avantage, c'est que vous êtes sûrs après que cette variable contient une valeur correcte, et pas du n'importe quoi ;)

### 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.  
  
Pour déclarer une constante, c'est en fait très simple : il faut 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 comme on vient d'apprendre à le faire. Après, il sera trop tard : vous ne pourrez plus changer la valeur de la constante.  
  
Exemple de déclaration de constante :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | **const** long NOMBRE\_DE\_VIES\_INITIALES = 5; |

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.

A part ça, une constante s'utilise comme une variable normale, vous pouvez afficher sa valeur si vous le désirez.  
La seule chose qui change, c'est que si vous essayez de modifier la valeur de la constante plus loin dans le programme, le compilateur vous indiquera qu'il y a une erreur avec cette constante.  
  
Les erreurs de compilation sont affichées en bas de l'écran, dans ce que j'appelle la "zone de la mort", vous vous souvenez ? ^^  
Dans un tel cas, le compilateur vous afficherait un mot doux du genre : *[Warning] assignment of read-only variable ' NOMBRE\_DE\_VIES\_INITIALES'* (traduction : "mais t'es vraiment idiot, pourquoi t'essaies de modifier la valeur d'une constante ? o_O " )

## Afficher le contenu d'une variable

On sait afficher du texte à l'écran avec la fonction *printf*.  
Maintenant, on va voir comment afficher la valeur d'une variable avec cette même fonction.  
  
On utilise en fait printf de la même manière, sauf que l'on rajoute un symbole spécial à l'endroit où on veut afficher la valeur de la variable.  
Par exemple :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | printf("Il vous reste %ld vies"); |

Ce "symbole spécial" dont je viens de vous parler est en fait un % suivi des lettres "ld". Ces lettres permettent d'indiquer ce que l'on doit afficher. "ld" signifie que c'est un nombre entier.  
Il existe plusieurs autres possibilités, mais pour des raisons de simplicité on va se contenter de retenir ces deux-là :

| **Symbole** | **Signification** |
| --- | --- |
| %ld | Nombre entier (ex. : 4) |
| %lf | Nombre décimal (ex. : 5.18) |

Je vous parlerai des autres symboles en temps voulu. Pour le moment, sachez que si vous voulez afficher une variable entière (char, int, long...), vous devez utiliser %ld, et pour un nombre décimal (float, double), vous utiliserez %lf.  
  
On a presque fini. On a indiqué qu'à un endroit précis on voulait afficher un nombre entier, mais on n'a pas précisé lequel ! Il faut donc indiquer à la fonction *printf*quelle est la variable dont on veut afficher la valeur.  
Pour ce faire, vous devez taper le nom de la variable après les guillemets et après avoir rajouté une virgule, comme ceci :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | printf("Il vous reste %ld vies", nombreDeVies); |

Le %ld sera remplacé par la variable indiquée après la virgule, à savoir nombreDeVies.  
On se teste ça un petit coup dans un programme ? :)  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | #include <stdio.h>  #include <stdlib.h>  int main(int argc, char \*argv[])  {  long nombreDeVies = 5; *// Au départ, le joueur a 5 vies*    printf("Vous avez %ld vies**\n**", nombreDeVies);  printf("\*\*\*\* B A M \*\*\*\***\n**"); *// Là il se prend un grand coup sur la tête*  nombreDeVies = 4; *// Il vient de perdre une vie !*  printf("Ah desole, il ne vous reste plus que %ld vies maintenant !**\n\n**", nombreDeVies);    system("PAUSE");  **return** 0;  } |

Ca pourrait presque être un jeu vidéo (il faut juste beaucoup d'imagination ^^ ).  
Ce programme affiche ceci à l'écran :  
  
**Code : Console**

|  |
| --- |
| Vous avez 5 vies  \*\*\*\* B A M \*\*\*\*  Ah desole, il ne vous reste plus que 4 vies maintenant !  Appuyez sur une touche pour continuer... |

Vous devriez reconnaître ce qui se passe dans votre programme :

1. Au départ le joueur a 5 vies, on affiche ça dans un printf
2. Ensuite le joueur prend un coup sur la tête (d'où le BAM)
3. Finalement il n'a plus que 4 vies, on affiche ça aussi avec un printf

Bref, c'est plutôt simple ^^

### Afficher plusieurs variables dans un même printf

Il est possible d'afficher la valeur de plusieurs variables dans un seul printf. Il vous suffit pour cela d'indiquer des %ld ou des %lf là où vous voulez, puis d'indiquer les variables correspondantes dans le même ordre, séparées par des virgules.  
  
Par exemple :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | printf("Vous avez %ld vies et vous etes au niveau n°%ld", nombreDeVies, niveau); |

Veillez à bien indiquer vos variables dans le bon ordre. Le premier %ld sera remplacé par la première variable (nombreDeVies), et le second %ld par la seconde variable (niveau). Si vous vous trompez d'ordre, votre phrase ne voudra plus rien dire ;)

Allez un petit test maintenant. Notez que j'enlève les lignes tout en haut (les directives de préprocesseur commençant par un #), je vais supposer que vous les mettez à chaque fois maintenant :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | int main(int argc, char \*argv[])  {  long nombreDeVies = 5, niveau = 1;    printf("Vous avez %ld vies et vous etes au niveau n°%ld**\n**", nombreDeVies, niveau);    system("PAUSE");  **return** 0;  } |

Ce qui affichera :  
  
**Code : Console**

|  |
| --- |
| Vous avez 5 vies et vous etes au niveau n°1  Appuyez sur une touche pour continuer... |

## Récupérer une saisie

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.  
Une fois que ça sera fait, on pourra faire tout un tas de choses avec, vous verrez ;)  
  
Pour demander à l'utilisateur de rentrer quelque chose dans la console, on va utiliser une autre fonction toute prête : *scanf*  
Cette fonction ressemble beaucoup à *printf*. Vous devez mettre un %ld ou un %lf entre guillemets pour indiquer si vous voulez que l'utilisateur rentre un entier ou un décimal. Puis vous devez indiquer après le nom de la variable qui va recevoir le nombre.  
  
Voici comment faire par exemple  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | scanf("%ld", &age); |

On ne doit mettre que le %ld (ou le %lf) entre les guillemets.  
Par ailleurs, il faut mettre le symbole & devant le nom de la variable qui va recevoir la valeur.

Euh, pourquoi mettre un & devant le nom de la variable o_O ?

Là, il va falloir que vous me fassiez confiance. Si je dois vous expliquer ça tout de suite, on n'est pas sortis de l'auberge croyez-moi ^^  
Que je vous rassure quand même : je vous expliquerai un peu plus tard ce que signifie ce symbole. Pour le moment, je choisis de ne pas vous l'expliquer pour ne pas vous embrouiller, c'est donc plutôt un service que je vous rends là :D  
  
Lorsque le programme arrive à un *scanf*, il se met en pause et attend que l'utilisateur rentre un nombre. Ce nombre sera stocké dans la variable "age".  
  
Voici un petit programme simple qui demande l'âge de l'utilisateur et qui le lui affiche ensuite :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | int main(int argc, char \*argv[])  {  long age = 0; *// On initialise la variable à 0*    printf("Quel age avez-vous ? ");  scanf("%ld", &age); *// On demande d'entrer l'age avec scanf*  printf("Ah ! Vous avez donc %ld ans !**\n\n**", age);    system("PAUSE");  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Quel age avez-vous ? 20  Ah ! Vous avez donc 20 ans !  Appuyez sur une touche pour continuer... |

Le programme se met donc en pause après avoir affiché la question "Quel age avez-vous ?". Le curseur apparaît à l'écran, vous devez taper un nombre entier (votre âge). Tapez ensuite sur Entrée pour valider, et le programme continuera à s'exécuter.  
Ici, tout ce qu'il fait après c'est afficher la valeur de la variable "age" à l'écran (*"Ah ! Vous avez donc 20 ans !"*).  
  
Voilà, vous avez compris le principe :)  
Grâce à la fonction *scanf*, on peut donc commencer à interagir avec l'utilisateur, histoire de lui demander 2-3 informations privées :-°  
  
Notez que rien ne vous empêche de taper autre chose qu'un nombre entier :

* Si vous rentrez un nombre décimal, comme 2.9, il sera automatiquement tronqué, c'est-à-dire que seule la partie entière sera conservée. Dans ce cas, c'est le nombre 2 qui aurait été stocké dans la variable.
* Si vous tapez des lettres au hasard ("éèydf"), la variable ne changera pas de valeur. Ce qui est bien ici, c'est qu'on avait initialisé notre variable à 0 au début. Du coup, le programme affichera "0 ans" si ça n'a pas marché. Si on n'avait pas initialisé la variable, le programme aurait pu afficher n'importe quoi !

**Les calculs de base**

Il faut savoir qu'en plus de n'être qu'une vulgaire calculatrice, votre ordinateur est une calculatrice très basique puisqu'on ne peut faire que des opérations très simples :

* Addition
* Soustraction
* Multiplication
* Division
* Modulo (je vous expliquerai ce que c'est si vous ne savez pas)

Si vous voulez faire des opérations plus compliquées (des carrés, des puissances, des logarithmes et autres joyeusetés) il vous faudra les programmer, c'est-à-dire **expliquer à l'ordinateur comment le faire**.  
Fort heureusement, nous verrons plus loin dans ce chapitre qu'il existe une librairie mathématique livrée avec le langage C qui contient plein de fonctions mathématiques toutes prêtes. Vous n'aurez pas à les réécrire donc, à moins que vous soyez maso (ou prof de maths, ça marche aussi :D )  
  
Voyons voir donc l'addition pour commencer.  
Pour faire une addition, on utilise le signe + (non, sans blague ? o_O ).  
Vous devez mettre le résultat de votre calcul dans une variable. On va donc par exemple créer une variable "resultat" de type long et faire un calcul :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3 | long resultat = 0;  resultat = 5 + 3; |

Pas besoin d'être un pro du calcul mental pour deviner que la variable "resultat" contiendra la valeur 8 après exécution ^^  
Bien sûr, rien ne s'affiche à l'écran avec ce code. Si vous voulez voir la valeur de la variable, rajoutez un printf comme vous savez maintenant si bien le faire :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | printf("5 + 3 = %ld", resultat); |

A l'écran, cela donnera :  
  
**Code : Console**

|  |
| --- |
| 5 + 3 = 8 |

Voilà pour l'addition.  
Pour les autres opérations, c'est pareil, seul le signe utilisé change :

* Addition : +
* Soustraction : -
* Multiplication : \*
* Division : /
* Modulo : %

Si vous avez déjà utilisé la calculatrice sur votre ordinateur, vous devriez connaître ces signes. Le signe "moins" est en fait le tiret, le signe "multiplié" est une étoile, et le signe "divisé" est le slash (la barre oblique).  
  
Il n'y a pas de difficulté particulière pour ces opérations, à part pour les deux dernières (la division et le modulo). Nous allons donc parler un peu plus en détail de chacune d'elles.

**La division**

Les divisions fonctionnent normalement sur un ordinateur quand il n'y a pas de reste. Par exemple, 6 / 3 ça fait 2, votre ordinateur vous donnera la réponse juste. Jusque-là pas de souci.  
  
Prenons maintenant une division avec reste comme 5 / 2.  
5 / 2, si vous calculez bien, ça fait 2.5 :)  
  
Et pourtant ! Regardez ce que fait ce code :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | long resultat = 0;    resultat = 5 / 2;  printf ("5 / 2 = %ld", resultat); |

**Code : Console**

|  |
| --- |
| 5 / 2 = 2 |

Il y a un gros problème. On a demandé 5 / 2, on s'attend à avoir 2.5, et l'ordinateur nous dit que ça fait 2 !  
  
Il y a anguille sous roche. Nos ordinateurs seraient-ils stupides à ce point ?  
En fait, quand il voit les chiffres 5 et 2, votre ordinateur fait une division de nombres entiers. Cela veut dire qu'il tronque le résultat, il ne garde que la partie entière (le 2).

Eh mais je sais ! C'est parce que *resultat*est un long ! Si ça avait été un double, il aurait pu stocker un nombre décimal à l'intérieur !

Même pas :D  
Essayez le même code en transformant juste *resultat* en double, et vous verrez qu'on vous affiche quand même 2.  
  
Si on veut que l'ordinateur affiche le bon résultat, il va falloir transformer les nombres 5 et 2 de l'opération en nombres décimaux, c'est-à-dire écrire 5.0 et 2.0 (ce sont les mêmes nombres, mais pour l'ordinateur ce sont des nombres décimaux, donc il fait une division de nombres décimaux) :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | double resultat = 0;    resultat = 5.0 / 2.0;  printf ("5 / 2 = %lf", resultat); |

**Code : Console**

|  |
| --- |
| 5 / 2 = 2.500000 |

Là le nombre est correct. Bon il affiche plein de zéros derrière si ça lui chante, mais le résultat reste quand même correct.  
  
  
Cette propriété de la division de nombres entiers est super importante. Il faut que vous reteniez que pour un ordinateur :  
5 / 2 = 2  
10 / 3 = 3  
4 / 5 = 0  
Si vous voulez avoir un résultat décimal, il faut que les nombres de l'opération soient décimaux :  
5.0 / 2.0 = 2.5  
10.0 / 3.0 = 3.33333  
4.0 / 5.0 = 0.8  
  
En fait, en faisant une division d'entiers comme "5 / 2", votre ordinateur répond à la question "Combien y a-t-il de fois 2 dans le nombre 5 ?". La réponse est 2 fois. De même, combien de fois y a-t-il le nombre 3 dans 10 ? 3 fois".  
  
Mais alors me direz-vous, comment on fait pour récupérer le reste de la division ?  
C'est là que super-modulo intervient :)

**Le modulo**

Le modulo est une opération mathématique qui permet d'obtenir **le reste d'une division**. C'est peut-être une opération moins connue que les 4 autres, mais pour votre ordinateur ça reste une opération de base... probablement justement pour combler le problème de la "division d'entiers" qu'on vient de voir.  
  
Le modulo, je vous l'ai dit tout à l'heure, se représente par le signe %.  
Voici quelques exemples de modulos :

* 5 % 2 = 1
* 14 % 3 = 2
* 4 % 2 = 0

Le modulo 5 % 2 est le reste de la division 5 / 2, c'est-à-dire 1. L'ordinateur calcule que 5 = 2 \* 2 + 1 (c'est ce 1, le reste, que le modulo renvoie)  
De même, 14 % 3, le calcul est 14 = 3 \* 4 + 2 (modulo renvoie le 2)  
Enfin, pour 4 % 2, la division tombe juste, il n'y a pas de reste, donc modulo renvoie 0.  
  
Voilà, je ne peux pas dire grand-chose d'autre de plus au sujet des modulos. Je tenais juste à l'expliquer pour ceux qui ne connaîtraient pas ;)  
  
En plus j'ai une bonne nouvelle : on a vu toutes les opérations de base. Finis les cours de maths ;)

**Des calculs entre variables**

Ce qui serait intéressant, maintenant que vous savez faire les 5 opérations de base, ce serait de s'entraîner à faire des calculs entre plusieurs variables.  
En effet, rien ne vous empêche de faire :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | resultat = nombre1 + nombre2; |

Cette ligne fait la somme des variables nombre1 et nombre2, et stocke le résultat dans la variable resultat.  
  
Et c'est là que les choses commencent à devenir très intéressantes :)  
Tenez, il me vient une idée. Vous avez maintenant déjà le niveau pour réaliser une mini calculatrice. Si si, je vous assure ! :D  
  
Imaginez un programme qui demande 2 nombres à l'utilisateur. Ces deux nombres, vous les stockez dans des variables.  
Ensuite, vous faites la somme de ces variables, et vous stockez le résultat dans une variable appelée "resultat".  
  
Vous n'avez plus qu'à afficher le résultat du calcul à l'écran, sous les yeux ébahis de l'utilisateur qui n'aurait jamais été capable de calculer cela de tête aussi vite ^^  
  
Essayez de coder vous-même ce petit programme, c'est facile et ça vous entraînera :)  
  
La réponse est ci-dessous :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | int main(int argc, char \*argv[])  {  long resultat = 0, nombre1 = 0, nombre2 = 0;    *// On demande les nombres 1 et 2 à l'utilisateur :*    printf("Entrez le nombre 1 : ");  scanf("%ld", &nombre1);  printf("Entrez le nombre 2 : ");  scanf("%ld", &nombre2);    *// On fait le calcul :*    resultat = nombre1 + nombre2;    *// Et on affiche l'addition à l'écran :*    printf ("%ld + %ld = %ld**\n**", nombre1, nombre2, resultat);    system("PAUSE");  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Entrez le nombre 1 : 30  Entrez le nombre 2 : 25  30 + 25 = 55 |

Mine de rien, on vient de faire là notre premier programme qui a un intérêt. Notre programme est capable d'additionner 2 nombres et d'afficher le résultat de l'opération :)  
  
Vous pouvez essayez avec n'importe quel nombre (du temps que vous ne dépassez pas les limites d'un type long), votre ordinateur effectuera le calcul en un éclair (encore heureux, parce que des opérations comme ça il doit en faire des milliards dans une même seconde ;) )  
  
Je vous conseille de faire la même chose avec les autres opérations pour vous entraîner (soustraction, multiplication...). En plus, vous ne devriez pas avoir trop de mal vu qu'il y a juste un ou deux signes à changer ;)  
Vous pouvez aussi ajouter une troisième variable et faire l'addition de 3 variables à la fois, ça fonctionne sans problème :   
  
**Code : C**

|  |  |
| --- | --- |
| 1 | resultat = nombre1 + nombre2 + nombre3; |

**Les raccourcis**

Comme promis, nous n'avons pas de nouvelles opérations à voir. Et pour cause ! On les a déjà toutes vues ;)  
C'est avec ces simples opérations de base que vous pouvez tout créer. Il n'y a pas besoin d'autres opérations. Je reconnais que c'est difficile à avaler, se dire qu'un jeu 3D ne fait rien d'autre au final que des additions et des soustractions, pourtant c'est la stricte vérité :p  
  
Ceci étant, il existe en C des techniques permettant de raccourcir l'écriture des opérations.  
Pourquoi utiliser des raccourcis ? Parce que, souvent, on fait des opérations répétitives. Vous allez voir ce que je veux dire par là tout de suite, avec ce qu'on appelle l'incrémentation.

**L'incrémentation**

Vous verrez que vous serez souvent amenés à ajouter 1 à une variable. Au fur et à mesure du programme, vous aurez des variables qui augmentent de 1 en 1.  
  
Imaginons que votre variable s'appelle "nombre" (nom très original n'est-ce pas ^^ ). Sauriez-vous comment faire pour ajouter 1 à cette variable, sans savoir quel est le nombre qu'elle contient ?  
  
Voici comment on doit faire :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | nombre = nombre + 1; |

Que se passe-t-il ici ? On fait le calcul nombre + 1, et on range ce résultat dans la variable... nombre ! Du coup, si notre variable nombre valait 4, elle vaut maintenant 5. Si elle valait 8, elle vaut maintenant 9 etc...  
  
Cette opération est justement répétitive. Les informaticiens étant des gens particulièrement fainéants, ils n'avaient guère envie de taper 2 fois le même nom de variable (ben oui quoi, c'est fatigant ! ^^ ).  
Ils ont donc inventé un raccourci pour cette opération qu'on appelle **l'incrémentation**. L'instruction ci-dessous fait exactement la même chose que le code qu'on vient de voir :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | nombre++; |

Cette ligne, bien plus courte que celle de tout à l'heure, signifie "Ajoute 1 à la variable nombre". Il suffit d'écrire le nom de la variable à incrémenter, de mettre 2 signes +, et de ne pas oublier le point-virgule bien entendu.  
  
Mine de rien, cela nous sera bien pratique par la suite car, comme je vous l'ai dit, on sera souvent amenés à faire des incrémentations (c'est-à-dire ajouter 1 à une variable).

Si vous êtes perspicaces, vous avez d'ailleurs remarqué que ce signe ++ se trouve dans le nom du langage "C++". C'est en fait un clin d'oeil des programmeurs, et vous êtes maintenant capables de le comprendre ! C++ signifie que c'est du langage C "incrémenté", c'est-à-dire si on veut "du langage C à 1 niveau supérieur" :p

**La décrémentation**

C'est tout bêtement l'inverse de l'incrémentation : on enlève 1 à une variable.  
Même si on fait plus souvent des incrémentations que des décrémentations, cela reste une opération pratique que vous utiliserez de temps en temps.  
  
La décrémentation, si on l'écrit en forme "longue" :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | nombre = nombre - 1; |

Et maintenant en forme "raccourcie" :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | nombre--; |

On l'aurait presque deviné tout seul ça ^^  
Au lieu de mettre un ++, vous mettez un --. Si votre variable vaut 6, elle vaudra 5 après l'instruction de décrémentation.

**Les autres raccourcis**

Il existe d'autres raccourcis qui fonctionnent sur le même principe. Cette fois, ces raccourcis fonctionnent pour toutes les opérations de base : + - \* / %  
  
Cela permet là encore d'éviter une répétition du nom d'une variable sur une même ligne.  
Ainsi, si vous voulez multiplier par 2 une variable :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | nombre = nombre \* 2; |

Vous pouvez l'écrire d'une façon raccourcie comme ceci :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | nombre \*= 2; |

Si le nombre vaut 5 au départ, il vaudra 10 après cette instruction.  
Pour les autres opérations de base, cela fonctionne de la même manière. Voici un petit programme d'exemple :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | long nombre = 2;  nombre += 4; *// nombre vaut 6...*  nombre -= 3; *// ... nombre vaut maintenant 3*  nombre \*= 5; *// ... nombre vaut 15*  nombre /= 3; *// ... nombre vaut 5*  nombre %= 3; *// ... nombre vaut 2 (car 5 = 1 \* 3 + 2)* |

*(allez boudez pas, un peu de calcul mental n'a jamais tué personne :p )*  
  
L'avantage ici est qu'on peut utiliser toutes les opérations de base, et qu'on peut ajouter, soustraire, multiplier par n'importe quel nombre.  
Ce sont des raccourcis à connaître si vous avez des lignes répétitives à taper un jour dans un programme :)  
  
Retenez quand même que l'incrémentation reste de loin le raccourci le plus utilisé ;)

**La librairie mathématique**

En langage C, il existe ce qu'on appelle des librairies "standard", c'est-à-dire des librairies toujours utilisables. Ce sont en quelque sorte des librairies "de base" qu'on utilise très souvent.  
  
Les librairies 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 librairie stdio.h.  
Il faut savoir qu'il existe une autre librairie, appelée math.h, qui contient de nombreuses fonctions mathématiques toutes prêtes.

En effet, les 5 opérations de base que l'on a vu sont loin d'être suffisantes ! Bon, il se peut que vous n'ayez jamais besoin de certaines opérations complexes comme les exponentielles (si vous ne savez pas ce que c'est, c'est que vous êtes peut-être un peu trop jeune ou que vous n'avez pas assez fait de maths dans votre vie :p ). Toutefois, la librairie mathématique contient de nombreuses autres fonctions dont vous aurez très probablement besoin.

Tenez par exemple, on ne sait pas faire des puissances en C ! Comment calculer un simple carré ? Vous pouvez toujours essayer de taper 5² dans votre programme, mais votre ordinateur ne le comprendra jamais car il ne sait pas ce que c'est... A moins que vous le lui expliquiez en lui indiquant la librairie mathématique !  
  
Pour pouvoir utiliser les fonctions de la librairie mathématique, il est indispensable de mettre la directive de préprocesseur suivante en haut de votre programme :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | #include <math.h> |

Une fois que c'est fait, vous pouvez utiliser toutes les fonctions de cette librairie.  
  
J'ai justement l'intention de vous les présenter :)  
Bon, comme il y a beaucoup de fonctions je ne peux pas faire la liste complète ici. D'une part ça vous ferait trop à assimiler, et d'autre part mes pauvres petits doigts auraient fondu avant la fin de l'écriture du chapitre ^^  
Je vais donc me contenter des principales fonctions, c'est-à-dire celles qui me semblent les plus importantes.

Vous n'avez peut-être pas tous le niveau en maths pour comprendre ce que font ces fonctions. Si c'est votre cas, pas d'inquiétude. Lisez juste, cela ne vous pénalisera pas pour la suite.  
Ceci étant, je vous offre un petit conseil gratuit : soyez attentifs en cours de maths, on dirait pas comme ça mais en fait ça finit par servir :p

**fabs**

Cette fonction retourne la valeur absolue d'un nombre, c'est-à-dire |x| (c'est la notation mathématique).  
La valeur absolue d'un nombre est sa valeur positive :

* Si vous donnez -53 à la fonction, elle vous renvoie 53.
* Si vous donnez 53 à la fonction, elle vous renvoie 53.

En bref, elle renvoie toujours l'équivalent positif du nombre que vous lui donnez.  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3 | double absolu = 0, nombre=-27;  absolu = fabs(nombre); *// absolu vaudra 27* |

Cette fonction renvoie un double, donc votre variable "absolu" doit être de type double.

Il existe aussi une fonction similaire appelée "abs", située dans "stdlib.h" cette fois.  
La fonction "abs" marche de la même manière, sauf qu'elle utilise des entiers (int). Elle renvoie donc un nombre entier de type int et non un double comme fabs.

**ceil**

Cette fonction renvoie le premier nombre entier après le nombre décimal qu'on lui donne.  
  
C'est une sorte d'arrondi. On arrondit en fait toujours au nombre entier supérieur.  
Par exemple, si on lui donne 26.512, la fonction renvoie 27.  
  
Cette fonction s'utilise de la même manière, et renvoie un double :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3 | double dessus = 0, nombre = 52.71;  dessus = ceil(nombre); *// dessus vaudra 53* |

**floor**

C'est l'inverse de la fonction précédente, cette fois elle renvoie le nombre directement en dessous.  
Si vous lui donnez 37.91, la fonction *floor*vous renverra donc 37

**pow**

Cette fonction permet de calculer la puissance d'un nombre. Vous devez lui indiquer 2 valeurs : le nombre, et la puissance à laquelle vous voulez l'élever. Voici le schéma de la fonction :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | pow(nombre, puissance); |

Par exemple, "2 puissance 3" (que l'on écrit habituellement 2^3 sur un ordinateur), c'est le calcul 2 \* 2 \* 2, ce qui fait 8 :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3 | double resultat = 0, nombre = 2;  resultat = pow(nombre, 3); *// resultat vaudra 2^3 = 8* |

Vous pouvez donc utiliser cette fonction pour calculer des carrés. Il suffit d'indiquer une puissance de 2.

**sqrt**

Cette fonction calcule la racine carrée d'un nombre. Elle renvoie un double.  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3 | double resultat = 0, nombre = 100;  resultat = sqrt(nombre); *// resultat vaudra 10* |

**sin, cos, tan**

Ce sont les 3 fameuses fonctions utilisées en trigonométrie.  
Le fonctionnement est le même, ces fonctions renvoient un double.  
  
Ces fonctions attendent une valeur en **radians**.

**asin, acos, atan**

Ce sont les fonctions arc sinus, arc cosinus et arc tangente, d'autres fonctions de trigonométrie.  
Elles s'utilisent de la même manière et renvoient un double.

**exp**

Cette fonction calcule l'exponentielle d'un nombre. Elle renvoie un double (oui oui, elle aussi ;) )

**log**

Cette fonction calcule le logarithme népérien d'un nombre (que l'on note aussi "ln")

**log10**

Cette fonction calcule le logarithme base 10 d'un nombre.

**Conclusion**

Conclusion, ben heureusement que je n'ai pas parlé des autres fonctions ^^ (en fait, je ne comprends même pas à quoi elles servent :-° )  
Déjà, avec ces fonctions-là vous avez de quoi faire si vous vous ennuyez :p  
  
Encore une fois, si vous n'avez pas compris un mot de ce que j'ai dit, ce n'est pas bien grave car on n'en a pas absolument besoin. Tout dépend en fait du programme que vous allez faire : si vous programmez une calculatrice scientifique, c'est sûr que vous vous en servirez :p  
  
Retenez quand même les fonctions **floor, ceil, et pow**, elles vous seront probablement utiles (même si vous ne programmez pas une calculatrice oui oui :p )

## La condition "if... else"

**Les conditions servent à "tester" des variables**. On peut par exemple dire "Si la variable machin est égale à 50, fais ceci"... Mais ce serait dommage de ne pouvoir tester que l'égalité ! Il faudrait aussi pouvoir tester si la variable est inférieure à 50, inférieure ou égale à 50, supérieure, supérieure ou égale...  
  
Ne vous inquiétez pas, le C a tout prévu :D  
*(mais vous n'en doutiez pas hein ;) )*  
  
Pour étudier les conditions "if... else", nous allons suivre le plan suivant :

1. Quelques symboles à connaître avant de commencer
2. Le test if
3. Le test else
4. Le test "else if"
5. Plusieurs conditions à la fois
6. Quelques erreurs courantes à éviter

Avant de voir comment on écrit une condition de type "if... else" en C, il faut donc que vous connaissiez 2-3 symboles de base. Ces symboles sont indispensables pour réaliser des conditions.

### Quelques symboles à connaître

Voici un petit tableau de symboles du langage C à connaître par coeur :)

| **Symbole** | **Signification** |
| --- | --- |
| == | Est égal à |
| > | Est supérieur à |
| < | Est inférieur à |
| >= | Est supérieur ou égal à |
| <= | Est inférieur ou égal à |
| != | Est différent de |

Faites très attention, il y a bien 2 symboles "==" pour tester l'égalité. Une erreur courante que font les débutants et de ne mettre qu'un symbole =, ce qui n'a pas la même signification en C. Je vous en reparlerai un peu plus bas.

### Un if simple

Attaquons maintenant sans plus tarder :)  
Nous allons faire un test simple, qui va dire à l'ordinateur :  
  
**Citation : Test simple**

SI la variable vaut ça  
ALORS fais ceci

En anglais, le mot "si" se traduit par "if". C'est celui qu'on utilise en langage C pour introduire une condition.  
Ecrivez donc un if. Ouvrez ensuite des parenthèses : à l'intérieur de ces parenthèses vous devrez écrire votre condition.  
  
Ensuite, ouvrez une accolade { et fermez-la un peu plus loin }. Tout ce qui se trouve à l'intérieur des accolades sera exécuté uniquement si la condition est vérifiée.  
  
Cela nous donne donc à écrire :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | **if** (*/\* Votre condition \*/*)  {  *// Instructions à exécuter si la condition est vraie*  } |

A la place de mon commentaire "Votre condition", on va écrire une condition pour tester une variable.  
Par exemple, on pourrait tester une variable "age" qui contient votre âge. Tenez pour s'entraîner, on va tester si vous êtes majeur, c'est-à-dire **si votre âge est supérieur ou égal à 18** :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | **if** (age >= 18)  {  printf ("Vous etes majeur !");  } |

Le symbole >= signifie "Supérieur ou égal", comme on l'a vu dans le tableau tout à l'heure.

S'il n'y a qu'une instruction entre les accolades, alors celles-ci deviennent facultatives. Vous pouvez donc écrire :

**Code : C**

|  |  |
| --- | --- |
| 1  2 | **if** (age >= 18)  printf ("Vous etes majeur !"); |

#### Tester ce code

Si vous voulez tester les codes précédents pour voir comment le if fonctionne, il faudra placer le if à l'intérieur d'une fonction main et ne pas oublier de déclarer une variable age à laquelle on donnera la valeur de notre choix.  
Ca peut paraître évident pour certains, mais apparemment ça ne l'était pas pour tout le monde aussi ai-je rajouté cette explication ;)  
  
Voici un code complet **que vous pouvez tester**:  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | #include <stdio.h>  #include <stdlib.h>  int main(int argc, char \*argv[])  {  long age = 20;    **if** (age >= 18)  {  printf ("Vous etes majeur !**\n**");  }    system("PAUSE");  **return** 0;  } |

Ici, la variable age vaut 20, donc le "Vous êtes majeur !" s'affichera.  
Essayez de changer la valeur initiale de la variable pour voir. Mettez par exemple 15 : la condition sera fausse, et donc "Vous êtes majeur !" ne s'affichera pas cette fois :)  
  
Servez-vous de ce code de base pour tester les prochains exemples du chapitre :)

#### Une question de propreté

La façon dont vous ouvrez les accolades n'est pas importante, votre programme marchera aussi bien si vous écrivez tout sur une même ligne. Par exemple :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | **if** (age >= 18) { printf ("Vous etes majeur !"); } |

Pourtant, même si c'est possible d'écrire comme ça, c'est **ultra déconseillé** (notez que quand j'écris plus gros, en gras rouge souligné, c'est généralement parce que c'est vraiment important :p )  
En effet, tout écrire sur une même ligne rend votre code **difficilement lisible**. Si vous ne prenez pas dès maintenant l'habitude d'aérer votre code, plus tard quand vous écrirez de plus gros programmes vous ne vous y retrouverez plus !  
  
Essayez donc de présenter votre code source de la même façon que moi : une accolade sur une ligne, puis vos instructions (précédées d'une tabulation pour les "décaler vers la droite"), puis l'accolade de fermeture sur une ligne.

Il existe plusieurs bonnes façons de présenter son code source. Ca ne change rien au fonctionnement du programme final, mais c'est une question de "style informatique" si vous voulez ;)  
Si vous voyez un code de quelqu'un d'autre présenté un peu différemment, c'est qu'il code avec un style différent. Le principal, c'est que son code reste aéré et lisible.

### Le "else" pour dire "sinon"

Maintenant que nous savons faire un test simple, allons un peu plus loin : si le test n'a pas marché (il est faux), on va dire à l'ordinateur d'exécuter d'autres instructions.  
  
En français, nous allons donc écrire quelque chose qui ressemble à cela :  
  
**Citation : Test avec sinon**

SI la variable vaut ça  
ALORS fais ceci  
SINON fais cela

Il suffit de rajouter le mot **else** après l'accolade fermante du **if**.  
Petit exemple :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **if** (age >= 18) *// Si l'âge est supérieur ou égal à 18*  {  printf ("Vous etes majeur !");  }  **else** *// Sinon...*  {  printf ("Ah c'est bete, vous etes mineur !");  } |

Les choses sont assez simples : si la variable age est supérieure ou égale à 18, on affiche le message "Vous êtes majeur !", sinon on affiche "Vous êtes mineur".

### Le "else if" pour dire "sinon si"

On a vu comment faire un "si" et un "sinon". Il est possible aussi de faire un "sinon si" pour faire un autre test si le premier test n'a pas marché. Le "sinon si" se met entre le if et le else.  
  
On dit dans ce cas à l'ordinateur :  
  
**Citation : Avec un sinon si**

SI la variable vaut ça ALORS fais ceci  
SINON SI la variable vaut ça ALORS fais ça  
SINON fais cela

Traduction en langage C :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **if** (age >= 18) *// Si l'âge est supérieur ou égal à 18*  {  printf ("Vous etes majeur !");  }  **else** **if** ( age > 4 ) *// Sinon, si l'âge est au moins supérieur à 4*  {  printf ("Bon t'es pas trop jeune quand meme...");  }  **else** *// Sinon...*  {  printf ("Aga gaa aga gaaa gaaa"); *// Langage Bébé, vous pouvez pas comprendre ;o)*  } |

L'ordinateur fait les tests dans l'ordre :

1. D'abord il teste le premier if : si la condition est vraie, alors il exécute ce qui se trouve entre les premières accolades.
2. Sinon, il va au "sinon si" et fait à nouveau un test : si ce test est vrai, alors il exécute les instructions correspondantes entre accolades.
3. Enfin, si aucun des tests précédents n'a marché, il exécute les instructions du "sinon".

Le "else" et le "else if" ne sont pas obligatoires. Pour faire une condition, il faut juste au moins un "if" (logique me direz-vous, sinon il n'y a pas de condition ! :lol: )  
Notez qu'on peut mettre autant de "else if" que l'on veut. On peut donc écrire :

**Citation : Plusieurs else if**

SI la variable vaut ça  
ALORS fais ceci  
SINON SI la variable vaut ça ALORS fais ça  
SINON SI la variable vaut ça ALORS fais ça  
SINON SI la variable vaut ça ALORS fais ça  
SINON fais cela

### Plusieurs conditions à la fois

Il peut aussi être utile de faire plusieurs tests à la fois dans votre if. Par exemple, vous voudriez tester si l'âge est supérieur à 18 ET si l'âge est inférieur à 25.  
Pour faire cela, il va falloir utiliser de nouveaux symboles :

|  |  |
| --- | --- |
| && | ET |
| || | OU |
| ! | NON |

#### Test ET

Si on veut faire le test que j'ai mentionné plus haut, il faudra écrire :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | **if** (age > 18 && age < 25) |

Les deux symboles "&&" signifient ET. Notre condition se dirait en français : "*Si l'âge est supérieur à 18 ET si l'âge est inférieur à 25*"

#### Test OU

Pour faire un OU, on utilise les 2 signes ||. Je dois avouer que ce signe n'est pas facilement accessible sur nos claviers. Pour le taper sur un clavier AZERTY français, il faudra faire **Alt Gr + 6**. Sur un clavier belge, il faudra faire Alt Gr + &.  
  
Imaginons un programme débile qui décide si une personne a le droit d'ouvrir un compte en banque. C'est bien connu, pour ouvrir un compte en banque il vaut mieux ne pas être trop jeune (on va dire arbitrairement qu'il faut avoir au moins 30 ans) ou bien avoir plein d'argent (parce que là même à 10 ans on vous acceptera à bras ouverts :D )  
Notre test pour savoir si le client a le droit d'ouvrir un compte en banque pourrait être :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **if** (age > 30 || argent > 100000)  {  printf("Bienvenue chez PicsouBanque !");  }  **else**  {  printf("Hors de ma vue, miserable !");  } |

Ce test n'est valide que si la personne a plus de 30 ans ou si elle possède plus de 100 000 euros ;)

#### Test NON

Le dernier symbole qu'il nous reste à tester est le point d'exclamation. En informatique, le point d'exclamation signifie "Non".  
Vous devez mettre ce signe avant votre condition pour dire "Si cela n'est pas vrai" :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | **if** (!(age < 18)) |

Cela pourrait se traduire par "*Si la personne n'est pas mineure*".  
Si on avait enlevé le "!" devant, cela aurait signifié l'inverse : "*Si la personne est mineure*".

### Quelques erreurs courantes de débutant

#### N'oubliez pas les 2 signes ==

Si on veut tester si la personne a tout juste 18 ans, il faudra écrire :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | **if** (age == 18)  {  printf ("Vous venez de devenir majeur !");  } |

**N'oubliez pas** de mettre 2 signes "égal" dans un if, comme ceci : ==  
Si vous ne mettez qu'un seul signe =, alors votre variable *prendra* la valeur 18 (comme on l'a appris dans le chapitre sur les variables). Nous ce qu'on veut faire ici, c'est tester la valeur de la variable, pas la changer ! Faites très attention à cela, beaucoup d'entre vous n'en mettent qu'un quand ils débutent et forcément... leur programme ne marche pas comme ils voudraient :p

#### Le point-virgule de trop

Une autre erreur courante de débutant : vous mettez parfois un point-virgule à la fin de la ligne d'un if. Or, un if est une condition, et on ne met de point-virgule qu'à la fin d'une instruction et non d'une condition.

Le code suivant ne marchera pas comme prévu car il y a un point-virgule à la fin du if :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | **if** (age == 18); *// Notez le point-virgule ici qui ne devrait PAS être là*  {  printf ("Tu es tout juste majeur");  } |

## Les booléens, le coeur des conditions

Nous allons maintenant rentrer plus en détail dans le fonctionnement d'une condition de type if... else.  
En effet, les conditions font intervenir quelque chose qu'on appelle **les booléens** en informatique.  
  
C'est un concept très important, donc ouvrez grand vos oreilles (euh vos yeux plutôt :p )

### Quelques petits tests pour bien comprendre

En cours de Physique-Chimie, mon prof avait l'habitude de nous faire commencer par quelques petites expériences avant d'introduire une nouvelle notion.  
Je vais l'imiter un peu aujourd'hui ;)  
  
  
Voici un code source très simple que je vous demande de tester :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **if** (1)  {  printf("C'est vrai");  }  **else**  {  printf("C'est faux");  } |

Résultat :  
  
**Code : Console**

|  |
| --- |
| C'est vrai |

Mais ??? On n'a pas mis de condition dans le if, juste un nombre. Qu'est-ce que ça veut dire ça n'a pas de sens ?

Si ça en a, vous allez comprendre :p  
Faites un autre test maintenant en remplaçant le 1 par un 0 :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **if** (0)  {  printf("C'est vrai");  }  **else**  {  printf("C'est faux");  } |

Résultat :  
  
**Code : Console**

|  |
| --- |
| C'est faux |

Faites maintenant d'autres tests en remplaçant le 0 par n'importe quel autre nombre entier, comme 4, 15, 226, -10, -36 etc...  
Qu'est-ce qu'on vous répond à chaque fois ? On vous répond : "C'est vrai".  
  
**Résumé de nos tests :** si on met un 0, le test est considéré comme faux, et si on met un 1 ou n'importe quel autre nombre, le test est vrai.

### Des explications s'imposent

En fait, à chaque fois que vous faites un test dans un if, ce test renvoie la valeur 1 s'il est vrai, et 0 s'il est faux.  
  
Par exemple :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | **if** (age >= 18) |

Ici, le test que vous faites est "age >= 18".  
  
Supposons que age vaille 23. Alors le test est vrai, et l'ordinateur "remplace" en quelque sorte "age >= 18" par 1.  
Ensuite, l'ordinateur obtient (dans sa tête) un "if (1)". Quand le nombre est 1, comme on l'a vu, l'ordinateur dit que la condition est vraie, donc il affiche "C'est vrai" !  
  
De même, si la condition est fausse, il remplace age >= 18 par le nombre 0, et du coup la condition est fausse : l'ordinateur va lire les instructions du "else".

### Un test avec une variable

Testez maintenant un autre truc : envoyez le résultat de votre condition dans une variable, comme si c'était une opération (car pour l'ordinateur, **c'est** une opération !).  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5 | long age = 20;  int majeur = 0;    majeur = age >= 18;  printf("Majeur vaut : %ld**\n**", majeur); |

Comme vous le voyez, la condition age >= 18 a renvoyé le nombre 1 **car elle est vraie**. Du coup, notre variable majeur vaut 1, on vérifie d'ailleurs ça en faisant un printf qui montre bien qu'elle a changé de valeur.  
  
Faites le même test en mettant age = 10 par exemple. Cette fois, majeur vaudra 0.

### Cette variable "majeur" est un booléen

Retenez bien ceci :

On dit qu'une variable à laquelle on fait prendre les valeurs 0 et 1 est **un booléen**.

Et aussi ceci :

0 = Faux  
1 = Vrai

Pour être tout à fait exact, 0 = faux et tous les autres nombres valent vrai (on a eu l'occasion de le tester plus tôt). Ceci dit, pour simplifier les choses on va se contenter de n'utiliser que les chiffres 0 et 1, pour dire si "quelque chose est faux ou vrai".

En langage C, il n'existe pas de type de variable "booléen". Il n'y a pas de type comme "double", "char"...  
En fait, le type booléen n'a été rajouté qu'en C++. En effet, en C++ vous avez un nouveau type "bool" qui a été créé spécialement pour ces variables booléennes.

Comme pour l'instant on fait du C, on n'a donc pas de type spécial. Du coup, on est obligé d'utiliser un autre type.  
Pour ma part, afin de bien différencier dans mon code les variables qui contiennent des nombres de celles qui contiennent un booléen, j'utilise le type "int". Dans la suite de ce cours, tous mes int seront donc des booléens, ce qui les rendra faciles à identifier :)

### Les booléens dans les conditions

Souvent, on fera un test "if" sur une variable booléenne :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | int majeur = 1;  **if** (majeur)  {  printf("Tu es majeur !");  }  **else**  {  printf("Tu es mineur");  } |

Comme majeur vaut 1, la condition est vraie, donc on affiche "Tu es majeur !".  
  
Ce qui est très pratique, c'est que la condition se lit facilement par un être humain. On voit "if (majeur)", ce que peut traduire par "*Si tu es majeur*"  
Les tests sur des booléens sont donc faciles à lire et à comprendre, pour peu que vous ayez donné des noms clairs à vos variables comme je vous ai dit de le faire depuis le début ;)  
  
Tenez, voici un autre test imaginaire :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | **if** (majeur && garcon) |

Ce test signifie "Si tu es majeur ET que tu es un garçon".  
garcon est ici une autre variable booléenne qui vaut 1 si vous êtes un garçon, et 0 si vous êtes... une fille ! Bravo, vous avez tout compris ! :D  
  
Les booléens servent donc à exprimer si quelque chose est vrai ou faux.  
C'est très utile, et ce que je viens de vous expliquer vous permettra de comprendre bon nombre de choses par la suite :)

Petite question : si on fait le test "if (majeur == 1)", ça marche aussi non ?

Tout à fait. Mais le principe des booléens c'est justement de raccourcir l'expression du if et de la rendre plus facilement lisible. Avouez que "if (majeur)" ça se comprend très bien non ? ;)  
  
**Retenez donc** : si votre variable est censée contenir un nombre, faites un test sous la forme "if (variable == 1)".  
Si au contraire votre variable est censée contenir un booléen (c'est-à-dire soit 1 soit 0 pour dire vrai ou faux), faites un test sous la forme "if (variable)".

## 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. Prenons cet exemple :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | **if** (age == 2)  {  printf("Salut bebe !");  }  **else** **if** (age == 6)  {  printf("Salut gamin !");  }  **else** **if** (age == 12)  {  printf("Salut jeune !");  }  **else** **if** (age == 16)  {  printf("Salut ado !");  }  **else** **if** (age == 18)  {  printf("Salut adulte !");  }  **else** **if** (age == 68)  {  printf("Salut papy !");  }  **else**  {  printf("Je n'ai aucune phrase de prete pour ton age ");  } |

### Construire un switch

Les informaticiens détestent faire des choses répétitives, on a eu l'occasion de le vérifier plus tôt ;)  
  
Alors, pour éviter d'avoir à faire des répétitions comme ça 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". Voici un switch basé sur l'exemple qu'on vient de voir :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | **switch** (age)  {  **case** 2:  printf("Salut bebe !");  **break**;  **case** 6:  printf("Salut gamin !");  **break**;  **case** 12:  printf("Salut jeune !");  **break**;  **case** 16:  printf("Salut ado !");  **break**;  **case** 18:  printf("Salut adulte !");  **break**;  **case** 68:  printf("Salut papy !");  **break**;  **default**:  printf("Je n'ai aucune phrase de prete pour ton age ");  **break**;  } |

Imprégnez-vous de mon exemple pour créer vos propres switch. On les utilise plus rarement, mais c'est vrai que c'est pratique car ça fait (un peu) moins de code à taper ;)  
  
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 "cas" : case 2, case 4, case 5, case 45...

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" qu'on connaît bien maintenant. Si la variable ne vaut aucune des valeurs précédentes, l'ordinateur ira lire le default.

### Gérer un menu avec un switch

Le switch est très souvent utilisé pour faire des menus en console.  
Je crois que le moment est venu de pratiquer un peu ;)

#### Au boulot !

En console, pour faire un menu, on fait des printf qui affichent les différentes options possibles. Chaque option est numérotée, et l'utilisateur doit rentrer le numéro du menu qui l'intéresse.  
Voici par exemple ce que la console devra afficher :  
  
**Code : Console**

|  |
| --- |
| === Menu ===  1. Royal Cheese  2. Mc Deluxe  3. Mc Bacon  4. Big Mac  Votre choix ? |

*(Vous aurez compris que j'avais un peu faim lorsque j'étais en train de rédiger ces lignes :lol: )*  
  
  
**Voici votre mission (si vous l'acceptez)** : reproduisez ce menu à l'aide de printf (facile), ajoutez un scanf pour enregistrer le choix de l'utilisateur dans une variable choixMenu (trop facile :p ), et enfin faites un switch pour dire à l'utilisateur "Tu as choisi le menu Royal Cheese" par exemple.  
  
Allez, au travail :diable: 

#### Correction

Voici la solution que j'espère que vous avez trouvée :p  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | #include <stdio.h>  #include <stdlib.h>  int main(int argc, char \*argv[])  {  long choixMenu;    printf("=== Menu ===**\n\n**");  printf("1. Royal Cheese**\n**");  printf("2. Mc Deluxe**\n**");  printf("3. Mc Bacon**\n**");  printf("4. Big Mac**\n**");  printf("**\n**Votre choix ? ");  scanf("%ld", &choixMenu);    printf("**\n**");    **switch** (choixMenu)  {  **case** 1:  printf("Vous avez choisi le Royal Cheese. Bon choix !");  **break**;  **case** 2:  printf("Vous avez choisi le Mc Deluxe. Berk, trop de sauce...");  **break**;  **case** 3:  printf("Vous avez choisi le Mc Bacon. Bon, ca passe encore ca ;o)");  **break**;  **case** 4:  printf("Vous avez choisi le Big Mac. Vous devez avoir tres faim !");  **break**;  **default**:  printf("Vous n'avez pas rentre un nombre correct. Vous ne mangerez rien du tout !");  **break**;  }    printf("**\n\n**");    system("PAUSE");  } |

Et voilà le travail :D  
  
J'espère que vous n'avez pas oublié le "default" à la fin du switch !  
En effet, quand vous programmez vous devez toujours penser à tous les cas. Vous avez beau dire de taper un nombre entre 1 et 4, vous trouverez toujours un imbécile qui ira taper "10" ou encore "Salut" alors que ce n'est pas ce que vous attendez ;)  
  
Bref, soyez toujours vigilants de ce côté-ci : ne faites pas confiance à l'utilisateur, il peut parfois rentrer n'importe quoi. Prévoyez toujours un cas "default" ou un "else" si vous faites ça avec des if.

Je vous conseille de vous familiariser avec le fonctionnement des menus en console, car on en fait souvent dans des programmes console et vous en aurez sûrement besoin ;)

## 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 2 fois la même condition : la première avec un "if... else", et la seconde, identique, mais sous forme de ternaire.

### Une condition if... else bien connue

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 de la variable age en fonction du booléen, pour mettre "18" si on est majeur, "17" si on est mineur. C'est un exemple complètement débile je suis d'accord, mais ça me permet de vous montrer comment on peut se servir des ternaires.  
  
Voici comment faire cela avec un if... else :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | **if** (majeur)  age = 18;  **else**  age = 17; |

Notez que j'ai enlevé dans cet exemple les accolades car elles sont **facultatives**s'il n'y a qu'une instruction, comme je vous l'ai expliqué plus tôt.

### La même condition en ternaire

Voici un code qui fait exactement la même chose que le code précédent, mais écrit cette fois sous forme de ternaire :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | age = (majeur) ? 18 : 17; |

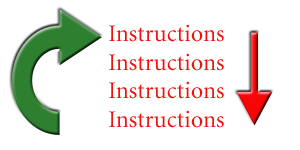
Les ternaires permettent, sur une seule ligne, de changer la valeur d'une variable en fonction d'une condition. Ici la condition est tout simplement "majeur", mais ça pourrait être n'importe quelle condition plus longue hein ;)  
  
Le point d'interrogation permet de dire "Est-ce que tu es majeur ?". Si oui, alors on met la valeur 18 dans age. Sinon (le ":" signifie "else" ici), on met la valeur 17.  
  
Les ternaires ne sont pas du tout indispensables, personnellement je ne les utilise pas trop car ils peuvent rendre la lecture d'un code source un peu difficile.  
Ceci étant, il vaut mieux que vous les connaissiez si, un jour, vous tombez sur un code plein de ternaires dans tous les sens :D

**Qu'est-ce qu'une boucle ?**

Tout comme pour les conditions, il y a plusieurs façons de réaliser des boucles. Au bout du compte, cela revient à faire la même chose : répéter les mêmes instructions un certain nombre de fois.  
Nous allons voir 3 types de boucles courantes en C :

* while
* do... while
* for

Dans tous les cas, le schéma est le même :



Voici ce qu'il se passe dans l'ordre :

1. L'ordinateur lit les instructions de haut en bas (comme d'habitude)
2. Puis, une fois arrivé à la fin de la boucle, il repart à la première instruction
3. Il recommence alors à lire les instructions de haut en bas...
4. ... Et il repart au début de la boucle.

Le problème dans ce système c'est que si on ne l'arrête pas, l'ordinateur est capable de répéter les instructions à l'infini ! Il est pas du genre à se plaindre vous savez, il fait ce qu'on lui dit de faire :p  
  
Et c'est là qu'on retrouve... **des conditions** !  
Quand on crée une boucle, on indique toujours une condition. Cette condition signifiera "*Répète la boucle tant que cette condition est vraie.*".  
  
Il y a plusieurs manières de s'y prendre comme je vous l'ai dit. Voyons voir sans plus tarder comment on réalise une boucle de type while en C :)

**La boucle while**

Voici comment on construit une boucle while :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | **while** (*/\* Condition \*/*)  {  *// Instructions à répéter*  } |

C'est aussi simple que cela :D  
While signifie "Tant que". On dit donc à l'ordinateur "*Tant que la condition est vraie : répète les instructions entre accolades*".  
  
Je vous propose de faire un test simple : on va demander à l'utilisateur de taper le nombre 47. Tant qu'il n'a pas tapé le nombre 47, on recommence à lui demander le nombre. Le programme ne pourra s'arrêter que si l'utilisateur tape le nombre 47 (je sais je sais, je suis diabolique :-° ) :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | long nombreEntre = 0;  **while** (nombreEntre != 47)  {  printf("Tapez le nombre 47 ! ");  scanf("%ld", &nombreEntre);  } |

Voici maintenant le test que j'ai fait. Notez que j'ai fait exprès de me planter 2-3 fois avant de taper le bon nombre :D  
  
**Code : Console**

|  |
| --- |
| Tapez le nombre 47 ! 10  Tapez le nombre 47 ! 27  Tapez le nombre 47 ! 40  Tapez le nombre 47 ! 47 |

Le programme s'est arrêté après avoir tapé le nombre 47.  
Cette boucle while se répète donc tant que l'utilisateur n'a pas tapé 47, c'est assez simple.  
  
Maintenant, essayons de faire quelque chose d'un peu plus intéressant : on veut que notre boucle se répète un certain nombre de fois.  
On va pour cela créer une variable "compteur" qui vaudra 0 au début du programme et que l'on va **incrémenter** au fur et à mesure. Vous vous souvenez de l'incrémentation ? Ca consiste à ajouter 1 à la variable en faisant "*variable++;*".  
  
Regardez attentivement ce bout de code et, surtout, essayez de le comprendre :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | long compteur = 0;  **while** (compteur < 10)  {  printf("Salut les Zeros !**\n**");  compteur++;  } |

Résultat :  
  
**Code : Console**

|  |
| --- |
| Salut les Zeros !  Salut les Zeros !  Salut les Zeros !  Salut les Zeros !  Salut les Zeros !  Salut les Zeros !  Salut les Zeros !  Salut les Zeros !  Salut les Zeros !  Salut les Zeros ! |

Ce code répète 10 fois l'affichage de "Salut les Zeros !".

Comment ça marche exactement ?

1. Au départ, on a une variable compteur initialisée à 0. Elle vaut donc 0 au début du programme.
2. La boucle while ordonne la répétition TANT QUE compteur est inférieur à 10. Comme compteur vaut 0 au départ, on rentre dans la boucle.
3. On affiche la phrase "Salut les Zeros !" via un printf.
4. On **incrémente**la valeur de la variable compteur, grâce à l'instruction "*compteur++;*". Compteur valait 0, il vaut maintenant 1.
5. On arrive à la fin de la boucle (accolade fermante), on repart donc au début, au niveau du while. On refait le test du while : "*Est-ce que compteur est toujours inférieur à 10 ?*". Ben oui, compteur vaut 1 :p Donc on recommence les instructions de la boucle.

Et ainsi de suite... Compteur va valoir progressivement 0, 1, 2, 3, ..., 8, 9, et 10. Lorsque compteur vaut 10, la condition "compteur < 10" est fausse. Comme l'instruction est fausse, on sort de la boucle.  
  
On pourrait voir d'ailleurs que la variable compteur augmente au fur et à mesure dans la boucle, en l'affichant dans le printf :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | long compteur = 0;  **while** (compteur < 10)  {  printf("La variable compteur vaut %ld**\n**", compteur);  compteur++;  } |

**Code : Console**

|  |
| --- |
| La variable compteur vaut 0  La variable compteur vaut 1  La variable compteur vaut 2  La variable compteur vaut 3  La variable compteur vaut 4  La variable compteur vaut 5  La variable compteur vaut 6  La variable compteur vaut 7  La variable compteur vaut 8  La variable compteur vaut 9 |

Voilà, si vous avez compris ça, vous avez tout compris ;)  
Vous pouvez vous amuser à augmenter la limite du nombre de boucles ("< 100" au lieu de "< 10"). Cela m'aurait été d'ailleurs très pratique plus jeune pour rédiger les punitions que je devais réécrire 100 fois :-°

**Attention aux boucles infinies**

Lorsque vous créez une boucle, assurez-vous toujours qu'elle peut s'arrêter à un moment ! Si la condition est toujours vraie, votre programme ne s'arrêtera jamais !  
Voici un exemple de boucle infinie :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | **while** (1)  {  printf("Boucle infinie**\n**");  } |

Souvenez-vous des booléens : 1 = vrai, 0 = faux. Ici, la condition est toujours vraie, donc ce programme affichera "Boucle infinie" sans arrêt !

Pour arrêter un tel programme sous Windows, vous n'avez pas d'autre choix que de fermer la console en cliquant sur la croix en haut à droite. Sous Linux faites Ctrl + C.

Faites donc très attention : évitez à tout prix de tomber dans une boucle infinie.  
  
Notez toutefois que les boucles infinies peuvent s'avérer utiles, notamment, nous le verrons plus tard, lorsque nous réaliserons des jeux.

**La boucle do... while**

Ce type de boucle est très similaire à while, bien qu'un peu moins utilisé en général.  
La seule chose qui change en fait par rapport à while, c'est la position de la condition. Au lieu d'être au début de la boucle, **la condition est à la fin** :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | long compteur = 0;  **do**  {  printf("Salut les Zeros !**\n**");  compteur++;  } **while** (compteur < 10); |

Qu'est-ce que ça change ?  
C'est très simple : la boucle while pourrait très bien ne jamais être exécutée si la condition est fausse dès le départ. Par exemple, si on avait initialisé le compteur à 50, la condition aurait été fausse dès le début et on ne serait jamais rentré dans la boucle.  
Pour la boucle do... while, c'est différent : cette boucle s'exécutera toujours au moins une fois. En effet, le test se fait à la fin comme vous pouvez le voir. Si on initialise compteur à 50, la boucle s'exécutera une fois.  
  
Il est donc parfois utile de faire des boucles de ce type, pour s'assurer que l'on rentre au moins une fois dans la boucle. C'est quand même plus rare ;)

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, ou sinon votre programme plantera à la compilation :p

**La boucle for**

En théorie, la boucle while permet de réaliser toutes les boucles que l'on veut.  
Toutefois, tout comme le switch pour les conditions, il est dans certains cas utiles d'avoir un autre système de boucle plus "condensé", plus rapide à écrire.  
  
Les boucles for sont très très utilisées en programmation. Je n'ai pas de statistiques sous la main, mais sachez que vous utiliserez certainement autant de for que de while, donc il vous faudra savoir manipuler ces deux types de boucles.  
  
  
Comme je vous le disais, les boucles for sont juste une autre façon de faire une boucle while.  
Voici un exemple de boucle while que nous avons vu tout à l'heure :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | long compteur = 0;  **while** (compteur < 10)  {  printf("Salut les Zeros !**\n**");  compteur++;  } |

Voici maintenant l'équivalent en boucle for :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | long compteur;  **for** (compteur = 0 ; compteur < 10 ; compteur++)  {  printf("Salut les Zeros !**\n**");  } |

Quelles différences ?

* Vous noterez qu'on n'a pas initialisé la variable compteur à 0 dès sa déclaration (mais on aurait pu le faire)
* Il y a beaucoup de choses entre les parenthèses après le for (nous allons détailler ça après)
* Il n'y a plus de compteur++; dans la boucle

Intéressons-nous à ce qui se trouve entre les parenthèses, car c'est là que réside tout l'intérêt de la boucle for. Il y a 3 instructions condensées, **séparée chacune par un point-virgule** :

* La première est **l'initialisation** : cette première instruction est utilisée pour préparer notre variable compteur. Dans notre cas, on initialise la variable à 0.
* La seconde est **la condition** : comme pour la boucle while, c'est la condition qui dit si la boucle doit être répétée ou pas. Tant que la condition est vraie, la boucle for continue.
* Enfin, il y a **l'incrémentation** : cette dernière instruction est exécutée à la fin de chaque tour de boucle pour mettre à jour la variable compteur. La quasi-totalité du temps on fera une incrémentation, mais on peut aussi faire une décrémentation (*variable--;*) ou encore n'importe quelle autre opération (*variable += 2;* pour avancer de 2 en 2 par exemple)

Bref, comme vous le voyez la boucle for n'est rien d'autre qu'un condensé ;)  
Sachez vous en servir, vous en aurez besoin plus d'une fois !

## Préparatifs et conseils

### Le principe du programme

Avant toute chose, il faut que je vous explique en quoi va consister notre programme.  
C'est un petit jeu que j'appelle "Plus ou moins".  
  
Le principe est le suivant :

* L'ordinateur tire au sort un nombre entre 1 et 100
* Il vous demande de deviner le nombre. Vous rentrez donc un nombre entre 1 et 100
* L'ordinateur compare le nombre que vous avez rentré avec le nombre "mystère" qu'il a tiré au sort. Il vous dit si le nombre mystère est supérieur ou inférieur à celui que vous avez entré
* Puis, l'ordinateur vous redemande le nombre.
* ... Et il vous indique si le nombre mystère est supérieur ou inférieur.
* Et ainsi de suite, jusqu'à ce que vous ayez trouvé le nombre mystère.

Le but du jeu, bien sûr, est de trouver le nombre mystère en un minimum de coups ;)  
  
Voici une capture d'écran d'une partie, c'est ce que vous devez arriver à faire :  
  
**Code : Console**

|  |
| --- |
| Quel est le nombre ? 50  C'est plus !  Quel est le nombre ? 75  C'est plus !  Quel est le nombre ? 85  C'est moins !  Quel est le nombre ? 80  C'est moins !  Quel est le nombre ? 78  C'est plus !  Quel est le nombre ? 79  Bravo, vous avez trouve le nombre mystere !!! |

### Tirer un nombre au sort

Mais comment tirer un nombre au hasard ? Je ne sais pas le faire !

Certes ^^  
Nous ne savons pas générer un nombre aléatoire. Il faut dire que demander cela à l'ordinateur n'est pas simple : il sait bien faire des calculs, mais lui demander de choisir un nombre au hasard ça il sait pas faire !  
En fait, pour "essayer" d'obtenir un nombre aléatoire, on doit faire faire des calculs complexes à l'ordinateur... ce qui revient au bout du compte à faire des calculs :p  
  
Bon, on a donc 2 solutions :

* Soit on demande à l'utilisateur à rentrer le nombre mystère via un scanf d'abord. Ca implique qu'il y ait 2 joueurs : l'un rentre un nombre au hasard, et l'autre essaie de le deviner ensuite.
* Soit on tente le tout pour le tout, et on essaie quand même de générer un nombre aléatoire automatiquement. L'avantage est qu'on peut jouer tout seul du coup. Le défaut... est qu'il va falloir que je vous explique comment faire :-°

Nous allons tenter la seconde solution, mais rien ne vous empêche de coder la première si vous voulez après ;)  
  
Pour générer un nombre aléatoire, on utilise la fonction rand().  
Cette fonction génère un nombre au hasard. Mais nous, on veut que ce nombre soit compris entre 1 et 100 par exemple (si on ne connaît pas les limites ça va devenir trop compliqué :p )  
  
Pour ce faire, on va utiliser la formule suivante :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2 | srand(time(NULL));  nombreMystere = (rand() % (MAX - MIN + 1)) + MIN; |

*(je ne pouvais pas trop vous demander de la deviner :lol: )*  
  
La première ligne (avec srand) permet d'initialiser le générateur de nombre aléatoires. Oui c'est un peu compliqué je vous avais prévenu :p  
nombreMystere est une variable qui contiendra le nombre au hasard.

L'instruction **srand**ne doit être exécutée qu'une seule fois (au début du programme). Il faut obligatoirement faire un srand une fois, et seulement une fois.  
Vous pouvez ensuite faire autant de rand() que vous voulez pour générer des nombres aléatoires. Mais il ne faut PAS que l'ordinateur lise l'instruction **srand**2 fois par programme, ne l'oubliez pas.

MAX et MIN sont des constantes, le premier est le nombre maximal (100) et le second le nombre minimal (1). Je vous recommande de définir ces constantes au début du programme, comme ceci :  
  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | **const** int MAX = 100, MIN = 1; |

(pour un nombre aléatoire entre 1 et 100)

### Les bibliothèques à inclure

Pour que votre programme marche correctement, vous aurez besoin d'inclure 3 bibliothèques : stdlib, stdio et time (la dernière sert pour les nombres aléatoires).  
Votre programme devra donc commencer par :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3 | #include <stdio.h>  #include <stdlib.h>  #include <time.h> |

### J'en ai assez dit !

Bon allez, j'arrête là parce que sinon je vais vous donner tout le code du programme si ça continue :p

Pour vous faire générer des nombres aléatoires, j'ai été obligé de vous donner des codes "tous prêts", sans vous expliquer totalement comment ils fonctionnaient. En général je n'aime pas faire ça mais là je n'ai pas vraiment le choix car ça compliquerait trop les choses pour le moment.  
Soyez sûrs toutefois que par la suite vous apprendrez de nouvelles choses qui vous permettront de comprendre cela ;)

Bref, vous en savez assez. Je vous ai expliqué le principe du programme, je vous ai fait une capture d'écran du programme au cours d'une partie.  
Avec tout ça, vous êtes tout à fait capables d'écrire le programme ;)

A vous de jouer !  
Bonne chance ! :)

## Correction !

Stop !  
Je ramasse les copies :D  
  
  
Alors, avez-vous réussi à coder le programme ?  
Je veux pas vous mettre la pression mais... vous devriez :p   
Cela n'est pas bien compliqué.  
  
  
Je vais vous donner une correction (la mienne), mais il y a plusieurs bonnes façons de faire le programme. Si votre code source n'est pas identique au mien et que vous avez trouvé une autre façon de le faire, c'est bien aussi hein ;)

### La correction de "Plus ou Moins"

**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | */\**  *Plus ou Moins*  *-------------*  *Réalisé par M@teo21, pour les cours du Site du Zér0*  *siteduzero.com (cours de programmation en C / C++ pour débutants)*  *\*/*  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  int main ( int argc, char\*\* argv )  {  int nombreMystere = 0, nombreEntre = 0;  **const** int MAX = 100, MIN = 1;  *// Génération du nombre aléatoire*  srand(time(NULL));  nombreMystere = (rand() % (MAX - MIN + 1)) + MIN;  */\* La boucle du programme. Elle se répète tant que l'utilisateur*  *n'a pas trouvé le nombre mystère \*/*  **do**  {  *// On demande le nombre*  printf("Quel est le nombre ? ");  scanf("%d", &nombreEntre);  *// On compare le nombre entré avec le nombre mystère*  **if** (nombreMystere > nombreEntre)  printf("C'est plus !**\n\n**");  **else** **if** (nombreMystere < nombreEntre)  printf("C'est moins !**\n\n**");  **else**  printf ("Bravo, vous avez trouve le nombre mystere !!!**\n\n**");  } **while** (nombreEntre != nombreMystere);  } |

### Exécutable et sources

Pour ceux qui le désirent, je mets à votre disposition en téléchargement l'exécutable du programme ainsi que les sources.

L'exécutable (.exe) est compilé pour Windows, donc si vous êtes sous un autre système d'exploitation il faudra recompiler le programme pour qu'il marche chez vous ;)

[Télécharger l'exécutable et les sources de  
"Plus ou Moins" (7 Ko)](http://www.siteduzero.com/uploads/fr/ftp/mateo21/plusoumoins.zip)

Il y a deux dossiers, l'un avec l'exécutable (compilé sous Windows je le rappelle), et l'autre avec les sources.  
  
Dans le cas de "Plus ou moins", les sources sont très simples : il y a juste un fichier main.c.  
**N'ouvrez pas le fichier main.c directement**. Ouvrez d'abord votre IDE favori (Code::Blocks, Visual...) et créez un nouveau projet de type console vide. Une fois que c'est fait, demander à **ajouter au projet le fichier main.c**.  
Vous pourrez alors compiler le programme pour tester, et le modifier si vous le désirez ;)

### Explications

Je vais maintenant vous expliquer mon code, en commençant par le début.

#### Les directives de préprocesseur

Ce sont les lignes commençant par # tout en haut. Elles incluent les bibliothèques dont on a besoin.  
Je vous les ai données tout à l'heure, donc si vous vous êtes plantés là c'est que vous êtes vraiment euh... pas doués :lol:

#### Les variables

On n'en a pas eu besoin de beaucoup.  
Juste une pour le nombre entré par l'utilisateur (nombreEntre) et une autre qui retient le nombre aléatoire généré par l'ordinateur (nombreMystere).  
  
J'ai aussi défini les constantes comme je vous l'ai dit au début de ce chapitre. L'avantage de définir les constantes en haut du programme, c'est que comme ça si vous voulez changer la difficulté (en mettant 1000 pour MAX par exemple) il suffit juste d'éditer cette ligne et de recompiler.

#### La boucle

J'ai choisi de faire une boucle do... while. En théorie, une boucle while simple aurait pu fonctionner aussi, mais j'ai trouvé qu'utiliser do... while était plus logique.  
  
Pourquoi ?  
Parce que, souvenez-vous, do... while est une boucle qui s'exécute **au moins une fois**. Et nous, on sait qu'on veut demander le nombre à l'utilisateur au moins une fois (il ne peut pas trouver le résultat en moins d'un coup, ou alors c'est qu'il est super fort o_O )  
  
  
A chaque passage dans la boucle, on redemande à l'utilisateur le nombre. On stocke le nombre qu'il propose dans nombreEntre.  
Puis, on compare ce nombreEntre au nombreMystere. Il y a 3 possibilités :

* Le nombre mystère est supérieur au nombre entré, on indique donc l'indice "C'est plus !"
* Le nombre mystère est inférieur au nombre entré, on indique l'indice "C'est moins !"
* Et si le nombre mystère n'est ni supérieur ni inférieur ? Ben... c'est qu'il est égal forcément ! D'où le else. Dans ce cas, on affiche la phrase "Bravo vous avez trouvé !"

Il faut une condition pour la boucle. Celle-ci était facile à trouver : on continue la boucle **TANT QUE le nombre entré n'est pas égal au nombre mystère**.  
La fois où le nombre est égal (c'est-à-dire quand on a trouvé), la boucle s'arrête. Le programme est alors terminé :)

## Idées d'amélioration

Non, vous ne croyiez tout de même pas que j'allais m'arrêter là ? ^^  
Les cours du Site du Zér0 c'est comme la mousse au chocolat, quand y'en a plus y'en a encore !  
  
  
Je veux vous inciter à continuer à améliorer ce programme, pour vous entraîner. N'oubliez pas que c'est en vous entraînant comme ceci que vous progresserez ! Ceux qui lisent les cours d'une traite sans jamais faire de tests font une grosse erreur, je l'ai dit et je le redirai ;)  
  
Figurez-vous que j'ai une imagination débordante, et même sur un petit programme comme celui-ci je vois plein d'idées pour l'améliorer ! :-°   
Attention : cette fois je ne vous fournis pas de correction, il faudra vous débrouiller tous seuls ! Si vous avez vraiment des problèmes, n'hésitez pas à aller faire un tour sur[les forums du site](http://www.siteduzero.com/forum.html), faites une recherche pour voir si on n'a pas déjà donné la réponse à vos questions, sinon créez un nouveau sujet pour poser ces questions :)

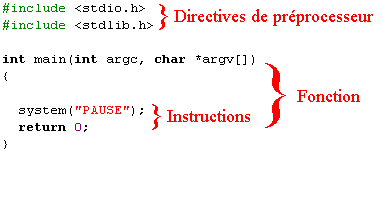
* **Faites un compteur de "coups"**. Ce compteur devra être une variable que vous incrémenterez à chaque fois que vous passez dans la boucle. Lorsque l'utilisateur a trouvé le nombre mystère, vous lui direz "*Bravo, vous avez trouvé le nombre mystère en 8 coups*" par exemple.
* Lorsque l'utilisateur aura trouvé le nombre mystère, le programme s'arrête. Pourquoi ne pas demander s'il veut faire **une autre partie** ?  
  Si vous faites ça, il vous faudra faire une boucle qui englobera la quasi-totalité de votre programme. Cette boucle devra se répéter TANT QUE l'utilisateur n'a pas demandé à arrêter le programme. Je vous conseille de rajouter une variable booléenne "continuerPartie" initialisée à 1 au départ. Si l'utilisateur demande à arrêter le programme, vous mettrez la variable à 0 et le programme s'arrêtera.
* Implémentez **un mode 2 joueurs** ! Attention, je veux qu'on ait le choix entre un mode 1 joueur et un mode 2 joueurs !  
  Vous devrez donc faire un menu au début de votre programme qui demande à l'utilisateur le mode de jeu qu'il veut faire.  
  La seule chose qui changera entre les deux modes de jeu, c'est la génération du nombre mystère. Dans un cas ce sera un rand() comme on a vu, dans l'autre cas ça sera... un scanf ;)
* Créez **plusieurs niveaux de difficulté**. Au début, faites un menu qui demande le niveau de difficulté. Par exemple :
  + 1 = entre 1 et 100
  + 2 = entre 1 et 1000
  + 3 = entre 1 et 10000

Si vous faites ça, vous devrez changer votre constante MAX... Ben oui, ça ne peut plus être une constante si la valeur doit changer au cours du programme ! Renommez donc cette variable en nombreMaximum (vous prendrez soin d'enlever le mot-clé "const" sinon ça sera toujours une constante !). La valeur de cette variable dépendra du niveau qu'on aura choisi.

Voilà, ça devrait vous occuper un petit bout de temps :D  
  
Amusez-vous bien et n'hésitez pas à chercher d'autres idées pour améliorer ce "Plus ou Moins", je suis sûr qu'il y en a !  
N'oubliez pas que [les forums](http://www.siteduzero.com/forum.html) sont à votre disposition si vous avez des questions ;)

## Créer et appeler une fonction

Nous avons vu dans les tous premiers chapitres qu'un programme en C commençait par une fonction appelée "main".  
Je vous avais d'ailleurs même fait un schéma récapitulatif, pour vous rappeler quelques mots de vocabulaire. Attendez que je retrouve ce schéma :euh:   
  
*\* va fouiller dans les archives poussièreuses \**  
  
Ah je l'ai :D  
Souvenirs souvenirs :



C'était au tout début hein ;)  
En haut, les directives de préprocesseur (un nom barbare sur lequel on reviendra d'ailleurs). Ces directives sont faciles à identifier : elles commencent par un # et sont généralement mises tout en haut des fichiers sources.  
  
Puis en-dessous, il y avait ce que j'avais déjà appelé "une fonction". Ici, sur mon schéma, vous voyez une fonction "main" (pas trop remplie il faut le reconnaître ^^ )  
Je vous avais dit qu'un programme en langage C **commençait par la fonction main**. Que je vous rassure, c'est toujours vrai :D  
  
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 le verrez : nous sommes toujours restés à l'intérieur des accolades de la fonction main.

Eh bien, c'est mal d'avoir fait ça ?

Non 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, bonjour le bordel :lol:   
  
Nous allons donc maintenant **apprendre à nous organiser**. Nous allons en fait découper nos programmes en petits bouts (souvenez-vous de l'image des legos que je vous ai donnée tout à l'heure).  
Chaque "petit bout de programme" sera ce qu'on appelle une fonction.

Quel est le but d'une fonction ?

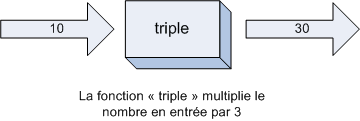
Une fonction exécute des actions et renvoie un résultat. C'est un **morceau de code** qui sert à faire quelque chose de précis.  
On dit qu'une fonction possède une entrée et une sortie. Schématiquement, ça donne quelque chose comme ça :



Lorsqu'on appelle une fonction, il y a 3 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) :



Bien entendu, les fonctions seront en général plus compliquées ;)  
Le but des fonctions est donc de simplifier le code source, pour ne pas avoir à retaper le même code plusieurs fois d'affilée.  
  
Rêvez un peu : plus tard, nous créerons par exemple une fonction "afficherFenetre" qui ouvrira une fenêtre à l'écran. Une fois la fonction écrite (c'est l'étape la plus difficile), on n'aura plus qu'à dire "Hep toi la fonction afficherFenetre, ouvre-moi une fenêtre !" ;)  
On pourra aussi écrire une fonction "deplacerPersonnage" dont le but sera de déplacer le personnage d'un jeu à l'écran etc etc :D  
  
Schéma d'une fonction

Vous avez déjà eu un aperçu de comment est faite une fonction avec la fonction main.  
Cependant pour bien que vous compreniez il va falloir que je vous montre quand même **comment on construit une fonction**.  
  
Voici le schéma d'une fonction, à retenir par coeur :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | type nomFonction(parametres)  {  *// Insérez vos instructions ici*  } |

Vous reconnaissez là un peu la forme de la fonction main.  
Voici ce qu'il faut savoir sur ce schéma :

* **type *(correspond à la sortie)*** : c'est le type de la fonction. 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 2 sortes de fonctions :
  + Les fonctions qui **renvoient une valeur** : on leur met un des types que l'on connaît (char, int, double...)
  + Les fonctions qui **ne renvoient pas de valeur** : on leur met un type spécial "void" (qui signifie "vide").
* **nomFonction**: 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).
* **parametres *(correspond à l'entrée)*** : entre parenthèses, vous pouvez envoyer des paramètres à la fonction. Ce sont des valeurs avec lesquelles la fonction va travailler.  
  Par exemple, pour une fonction "triple", vous envoyez un nombre en paramètre. La fonction "récupère" ce nombre et en calcule le triple, en le multipliant par 3. Elle renvoie ensuite le résultat de ses calculs.

Vous pouvez envoyer autant de paramètres que vous le voulez.  
Vous pouvez aussi n'envoyer aucun paramètre à la fonction, mais ça se fait plus rarement.

* Ensuite vous avez les **accolades**qui indiquent le début et la fin de la fonction. A l'intérieur de ces accolades vous mettrez les instructions que vous voulez. Pour la fonction triple, il faudra taper des instructions qui multiplient par 3 le nombre reçu en entrée.

Une fonction, c'est donc un mécanisme qui reçoit des valeurs en **entrée**(les paramètres) et qui renvoie un résultat en **sortie**.  
Créer une fonction

Voyons voir un exemple pratique sans plus tarder : la fameuse fonction "triple" dont je vous parle depuis tout à l'heure. On va dire que cette fonction reçoit un nombre entier de type long et qu'elle renvoie un nombre entier aussi de type long. Cette fonction calcule le triple du nombre qu'on lui donne :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | long triple(long nombre)  {  long resultat = 0;  resultat = 3 \* nombre; *// On multiplie le nombre qu'on nous a transmis par 3*  **return** resultat; *// On retourne la variable resultat qui vaut le triple de nombre*  } |

Voilà notre première fonction :D  
Une première chose importante : comme vous le voyez, la fonction est de type long. Elle doit donc renvoyer une valeur de type long.  
  
Entre les parenthèses, vous avez les variables que la fonction **reçoit**. Ici, notre fonction triple reçoit une variable de type long appelée "nombre".  
  
La ligne qui indique de "renvoyer une valeur" est celle qui contient le "return". Cette ligne se trouve généralement à la fin de la fonction, après les calculs.  
*return resultat;* dit à la fonction : arrête-toi là et renvoie le nombre "resultat". Cette variable "resultat" DOIT être de type long, car la fonction renvoie un long comme on l'a dit plus haut ;)  
  
La variable resultat est déclarée (= créée) dans la fonction "triple". Cela signifie qu'elle n'est utilisable que dans cette fonction, et pas dans une autre comme la fonction "main" par exemple. C'est donc une variable propre à la fonction "triple".  
  
  
Mais est-ce la façon la plus courte de faire notre fonction triple ?  
Non, on peut faire ça en une ligne en fait ;)  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | long triple(long nombre)  {  **return** 3 \* nombre;  } |

Cette fonction fait exactement la même chose que la fonction de tout à l'heure, elle est juste plus rapide à écrire :)  
Généralement, vos fonctions contiendront plusieurs variables pour effectuer leurs calculs et leurs opérations, rares seront les fonctions aussi courtes que "triple" ;)  
  
Plusieurs paramètres, aucun paramètre

#### Plusieurs paramètres

Notre fonction "triple" contient un paramètre, mais il est possible de créer des fonctions prenant plusieurs paramètres.  
Par exemple, une fonction addition qui additionne deux nombres a et b :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | long addition(long a, long b)  {  **return** a + b;  } |

Il suffit de séparer les différents paramètres par une virgule comme vous le voyez ;)

#### Aucun paramètre

Certaines fonctions, plus rares, ne prennent aucun paramètre en entrée. Ces fonctions feront généralement toujours la même chose. En effet, si elles n'ont pas de nombres sur lesquels travailler, vos fonctions serviront juste à effectuer certaines actions, comme afficher du texte à l'écran (et encore, ce sera toujours le même texte !)  
  
Imaginons une fonction "bonjour" qui affiche juste bonjour à l'écran :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | void bonjour()  {  printf("Bonjour");  } |

Je n'ai rien mis entre parenthèses car la fonction ne prend aucun paramètre.  
De plus, j'ai mis le type "void" dont je vous ai parlé plus haut.  
  
En effet, comme vous le voyez ma fonction n'a pas non plus de "return". Elle ne retourne rien. Une fonction qui ne retourne rien est de type void.  
  
  
Appeler une fonction

On va maintenant tester un code source pour pratiquer un peu avec ce qu'on vient d'apprendre.  
Nous allons utiliser notre fonction "triple" (décidemment je l'aime bien) pour calculer le triple d'un nombre.  
  
Pour le moment, je vous demande d'écrire la fonction "triple" AVANT la fonction main. Si vous la mettez après, ça ne marchera pas. Je vous expliquerai pourquoi par la suite ;)  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | #include <stdio.h>  #include <stdlib.h>  long triple(long nombre)  {  **return** 3 \* nombre;  }  int main(int argc, char \*argv[])  {  long nombreEntre = 0, nombreTriple = 0;    printf("Entrez un nombre... ");  scanf("%ld", &nombreEntre);    nombreTriple = triple(nombreEntre);  printf("Le triple de ce nombre est %ld**\n**", nombreTriple);    system("PAUSE");  **return** 0;  } |

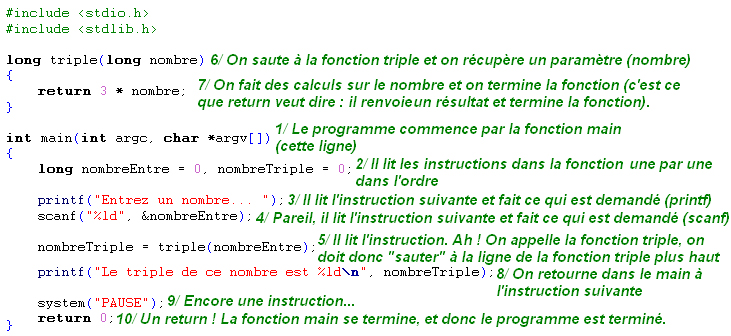
Notre programme commence par la fonction main comme vous le savez.  
On demande à l'utilisateur de rentrer un nombre. On envoie ce nombre qu'il a rentré à la fonction triple, et on récupère le résultat dans la variable nombreTriple. Regardez en particulier cette ligne, c'est la plus intéressante car c'est l'*appel de la fonction*:  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | nombreTriple = triple(nombreEntre); |

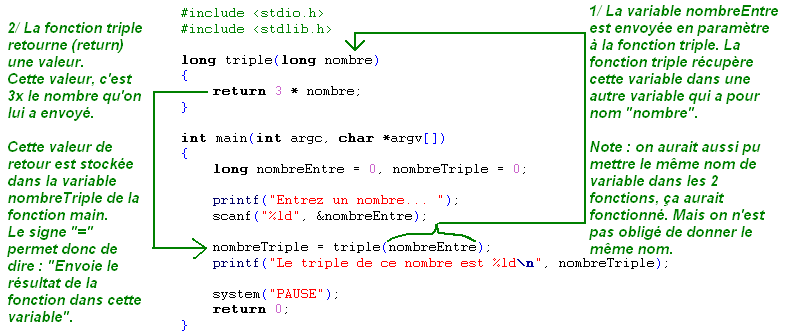
Entre parenthèses, on envoie une variable en **entrée** à la fonction triple, c'est le nombre sur lequel elle va travailler.  
Cette fonction renvoie une valeur, valeur qu'on récupère dans la variable nombreTriple. On demande donc à l'ordinateur dans cette ligne : "Demande à la fonction triple de me calculer le triple de nombreEntre, et stocke le résultat dans la variable nombreTriple".

#### Les mêmes explications sous forme de schéma

Vous avez encore du mal à comprendre comment ça fonctionne concrètement ?  
Pas de panique ! Je suis sûr que vous allez comprendre avec mes schémas :)  
  
Ce premier schéma vous explique dans quel ordre le code est lu. Commencez donc par lire la ligne numérotée "1", puis "2", puis "3" (bon vous avez compris je crois :D )



Si vous avez compris dans quel ordre l'ordinateur lisait les instructions, vous avez déjà compris le principal :)  
  
  
Maintenant, il faut bien comprendre qu'une fonction reçoit des paramètres en entrée et renvoie une valeur en sortie.



**Note**: ce n'est pas tout le temps le cas comme ça pour toutes les fonctions. Parfois, une fonction ne prend aucun paramètre en entrée, ou au contraire elle en prend plusieurs (je vous ai expliqué ça un peu plus haut).  
De même, parfois une fonction renvoie une valeur, parfois elle ne renvoie rien (dans ce cas il n'y a pas de *return*).

#### Testons ce programme

Voici un exemple d'utilisation du programme (y'a rien de bien extraordinaire car c'est une fonction toute bête hein ;) ) :  
  
**Code : Console**

|  |
| --- |
| Entrez un nombre... 10  Le triple de ce nombre est 30 |

Vous n'êtes pas obligés de stocker le résultat d'une fonction dans une variable ! Vous pouvez directement envoyer le résultat de la fonction triple à une autre fonction, comme si triple(nombreEntre) était une variable.

Regardez bien ceci, c'est le même code mais y'a un changement au niveau du dernier printf, et on n'a pas déclaré de variable nombreTriple car on ne s'en sert plus :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | #include <stdio.h>  #include <stdlib.h>  long triple(long nombre)  {  **return** 3 \* nombre;  }  int main(int argc, char \*argv[])  {  long nombreEntre = 0;    printf("Entrez un nombre... ");  scanf("%ld", &nombreEntre);  *// Le résultat de la fonction est directement envoyé au printf et n'est pas stocké dans une variable*  printf("Le triple de ce nombre est %ld**\n**", triple(nombreEntre));    system("PAUSE");  **return** 0;  } |

Comme vous le voyez, triple(nombreEntre) est directement envoyé au printf.  
Que fait l'ordinateur quand il tombe sur cette ligne ?  
  
C'est très simple. Il voit que la ligne commence par printf, il va donc appeler la fonction printf.  
Il envoie à la fonction printf tous les paramètres qu'on lui donne. Le premier paramètre est le texte à afficher, et le second est un nombre.  
Votre ordinateur voit que pour envoyer ce nombre à la fonction printf il doit d'abord appeler la fonction triple. C'est ce qu'il fait : il appelle triple, il effectue les calculs de triple, et une fois qu'il a le résultat il l'envoie directement dans la fonction printf !  
  
C'est un peu une imbrication de fonctions ;)  
  
Et le top, c'est qu'une fonction peut en appeler une autre à son tour !  
Notre fonction triple pourrait appeler une autre fonction, qui elle-même appellerait une autre fonction etc... C'est ça le principe de la programmation en C ! Tout est combiné, comme dans un jeu de Legos ;)  
  
Au final, le plus dur sera d'écrire vos fonctions. Une fois que vous les aurez écrites, vous n'aurez plus qu'à appeler les fonctions sans vous soucier des calculs qu'elles peuvent bien faire à l'intérieur. Ca va permettre de simplifier considérablement l'écriture de nos programmes, et ça croyez-moi on en aura bien besoin ! :)

## Plein d'exemples pour bien comprendre

Vous avez dû vous en rendre compte : je suis un maniaque des exemples.  
La théorie c'est bien, mais si on ne fait que ça on risque de ne pas retenir grand chose, et surtout ne pas comprendre comment s'en servir, ce qui serait un peu dommage :p  
  
Je vais donc maintenant vous montrer plusieurs exemples d'utilisation de fonctions, pour que vous ayez une idée de leur intérêt. Je vais m'efforcer de faire des cas différents à chaque fois, pour que vous puissiez avoir des exemples de tous les types de fonctions qui peuvent exister.  
  
Je ne vous apprendrai pas grand chose de nouveau, mais ça sera l'occasion de voir des exemples pratiques. Si vous avez déjà compris tout ce que j'ai expliqué avant, c'est très bien et normalement aucun des exemples qui vont suivre ne devrait vous surprendre ;)

### Conversion euros / francs

On commence par une fonction très similaire à "triple", qui a quand même un minimum d'intérêt cette fois : une fonction qui convertit les euros en francs.  
Pour ceux d'entre vous qui ne connaîtraient pas ces monnaies (il n'y a pas que des français sur le Site du Zér0 hein ;) ) sachez que :  
  
**1 euro = 6.55957 francs**  
  
On va créer une fonction appelée conversion.  
Cette fonction prend une variable en entrée de type double et retourne une sortie de type double (on va forcément manipuler des chiffres décimaux !).  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | double conversion(double euros)  {  double francs = 0;    francs = 6.55957 \* euros;  **return** francs;  }  int main(int argc, char \*argv[])  {  printf("10 euros = %lfF**\n**", conversion(10));  printf("50 euros = %lfF**\n**", conversion(50));  printf("100 euros = %lfF**\n**", conversion(100));  printf("200 euros = %lfF**\n**", conversion(200));    system("PAUSE");  **return** 0;  } |

**Code : Console**

|  |
| --- |
| 10 euros = 65.595700F  50 euros = 327.978500F  100 euros = 655.957000F  200 euros = 1311.914000F |

Il n'y a pas grand chose de différent par rapport à la fonction triple je vous avais prévenu :p  
D'ailleurs, ma fonction conversion est un peu longue et pourrait être raccourcie en une ligne, je vous laisse le faire je vous ai déjà expliqué comment faire plus haut.  
  
Dans la fonction main, j'ai fait exprès de faire plusieurs printf pour vous montrer l'intérêt d'avoir une fonction. Pour obtenir la valeur de 50 euros, je n'ai qu'à écrireconversion(50). Et si je veux avoir la conversion en francs de 100 euros, j'ai juste besoin de changer le paramètre que j'envoie à la fonction (100 au lieu de 50).  
  
  
**A vous de jouer !** Ecrivez une seconde fonction (toujours avant la fonction main) qui fera elle la conversion inverse : Francs => Euros. Pas bien difficile hein, y'a juste un signe d'opération à changer ;)

### La punition

On va maintenant s'intéresser à une fonction qui ne renvoie rien (pas de sortie).  
C'est une fonction qui affiche le même message à l'écran autant de fois qu'on lui demande. Cette fonction prend un paramètre en entrée : le nombre de fois où il faut afficher la punition.  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | void punition(long nombreDeLignes)  {  long i;    **for** (i = 0 ; i < nombreDeLignes ; i++)  {  printf("Je ne dois pas recopier mon voisin**\n**");  }  }  int main(int argc, char \*argv[])  {  punition(10);    system("PAUSE");  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Je ne dois pas recopier mon voisin  Je ne dois pas recopier mon voisin  Je ne dois pas recopier mon voisin  Je ne dois pas recopier mon voisin  Je ne dois pas recopier mon voisin  Je ne dois pas recopier mon voisin  Je ne dois pas recopier mon voisin  Je ne dois pas recopier mon voisin  Je ne dois pas recopier mon voisin  Je ne dois pas recopier mon voisin |

On a ici affaire à une fonction qui ne renvoie aucune valeur. Cette fonction se contente juste d'effectuer des actions (ici, elle affiche des messages à l'écran).  
Une fonction qui ne renvoie aucune valeur est de type "void", c'est pour cela qu'on a écrit void.  
A part ça, rien de bien différent ;)  
  
Il aurait été bien plus intéressant de créer une fonction "punition" qui s'adapte à n'importe quelle punition. On lui aurait envoyé 2 paramètres : le texte à répéter, et le nombre de fois qu'il doit être répété. Le problème, c'est qu'on ne sait pas encore gérer le texte en C (au cas où vous auriez pas les yeux très ouverts, je vous rappelle qu'on fait que manipuler des variables contenant des nombres depuis le début :p )  
D'ailleurs à ce sujet, je vous annonce que nous ne tarderons pas à apprendre à utiliser des variables qui retiennent du texte. C'est que c'est plus compliqué qu'il n'y paraît, et on ne pouvait pas l'apprendre dès le début du cours ;)

### Aire d'un rectangle

L'aire d'un rectangle est facile à calculer : largeur \* hauteur.  
Notre fonction aireRectangle va prendre 2 paramètres, la largeur et la hauteur. Elle renverra l'aire :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | double aireRectangle(double largeur, double hauteur)  {  **return** largeur \* hauteur;  }  int main(int argc, char \*argv[])  {  printf("Rectangle de largeur 5 et hauteur 10. Aire = %lf**\n**", aireRectangle(5, 10));  printf("Rectangle de largeur 2.5 et hauteur 3.5. Aire = %lf**\n**", aireRectangle(2.5, 3.5));  printf("Rectangle de largeur 4.2 et hauteur 9.7. Aire = %lf**\n**", aireRectangle(4.2, 9.7));    system("PAUSE");  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Rectangle de largeur 5 et hauteur 10. Aire = 50.000000  Rectangle de largeur 2.5 et hauteur 3.5. Aire = 8.750000  Rectangle de largeur 4.2 et hauteur 9.7. Aire = 40.740000 |

Pourrait-on afficher directement la largeur, la hauteur et l'aire dans la fonction ?

Bien sûr !  
Dans ce cas, la fonction ne renverrait plus rien, elle se contenterait de calculer l'aire et de l'afficher immédiatement.  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | void aireRectangle(double largeur, double hauteur)  {  double aire = 0;    aire = largeur \* hauteur;  printf("Rectangle de largeur %lf et hauteur %lf. Aire = %lf**\n**", largeur, hauteur, aire);  }  int main(int argc, char \*argv[])  {  aireRectangle(5, 10);  aireRectangle(2.5, 3.5);  aireRectangle(4.2, 9.7);    system("PAUSE");  **return** 0;  } |

Comme vous le voyez, le printf est *à l'intérieur* de la fonction aireRectangle et fait le même affichage que tout à l'heure. C'est juste une façon différente de procéder ;)

### Un menu

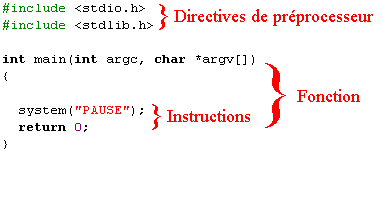
Ce code est plus intéressant et concret. On crée une fonction menu() qui ne prend aucun paramètre en entrée. Cette fonction se contente d'afficher le menu, et demande à l'utilisateur de faire un choix.  
La fonction renvoie le choix de l'utilisateur.  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | long menu()  {  long choix = 0;    **while** (choix < 1 || choix > 4)  {  printf("Menu :**\n**");  printf("1 : Poulet de dinde aux escargots rotis a la sauce bearnaise**\n**");  printf("2 : Concombres sucres a la sauce de myrtilles enrobee de chocolat**\n**");  printf("3 : Escalope de kangourou saignante et sa gelee aux fraises poivree**\n**");  printf("4 : La surprise du Chef (j'en salive d'avance...)**\n**");  printf("Votre choix ? ");  scanf("%ld", &choix);  }    **return** choix;  }  int main(int argc, char \*argv[])  {  **switch** (menu())  {  **case** 1:  printf("Vous avez pris le poulet**\n**");  **break**;  **case** 2:  printf("Vous avez pris les concombres**\n**");  **break**;  **case** 3:  printf("Vous avez pris l'escalope**\n**");  **break**;  **case** 4:  printf("Vous avez pris la surprise du Chef. Vous etes un sacre aventurier dites donc !**\n**");  **break**;  }    system("PAUSE");  **return** 0;  } |

J'en ai profité pour améliorer le menu (par rapport à ce qu'on faisait habituellement) : la fonction menu réaffiche le menu tant que l'utilisateur n'a pas rentré un nombre compris entre 1 et 4. Comme ça, aucun risque que la fonction renvoie un nombre qui ne figure pas au menu !  
  
Dans le main, vous avez vu qu'on fait un switch(menu()). Une fois que la fonction menu() est terminée, elle renvoie le choix de l'utilisateur directement dans le switch. C'est pratique et rapide comme méthode :)  
  
**A vous de jouer !** Le code est encore améliorable : on pourrait afficher un message d'erreur si l'utilisateur rentre un mauvais nombre plutôt que de bêtement réafficher le menu ;)

## Créer et appeler une fonction

Nous avons vu dans les tous premiers chapitres qu'un programme en C commençait par une fonction appelée "main".  
Je vous avais d'ailleurs même fait un schéma récapitulatif, pour vous rappeler quelques mots de vocabulaire. Attendez que je retrouve ce schéma :euh:   
  
*\* va fouiller dans les archives poussièreuses \**  
  
Ah je l'ai :D  
Souvenirs souvenirs :



C'était au tout début hein ;)  
En haut, les directives de préprocesseur (un nom barbare sur lequel on reviendra d'ailleurs). Ces directives sont faciles à identifier : elles commencent par un # et sont généralement mises tout en haut des fichiers sources.  
  
Puis en-dessous, il y avait ce que j'avais déjà appelé "une fonction". Ici, sur mon schéma, vous voyez une fonction "main" (pas trop remplie il faut le reconnaître ^^ )  
Je vous avais dit qu'un programme en langage C **commençait par la fonction main**. Que je vous rassure, c'est toujours vrai :D  
  
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 le verrez : nous sommes toujours restés à l'intérieur des accolades de la fonction main.

Eh bien, c'est mal d'avoir fait ça ?

Non 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, bonjour le bordel :lol:   
  
Nous allons donc maintenant **apprendre à nous organiser**. Nous allons en fait découper nos programmes en petits bouts (souvenez-vous de l'image des legos que je vous ai donnée tout à l'heure).  
Chaque "petit bout de programme" sera ce qu'on appelle une fonction.

Quel est le but d'une fonction ?

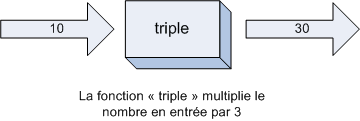
Une fonction exécute des actions et renvoie un résultat. C'est un **morceau de code** qui sert à faire quelque chose de précis.  
On dit qu'une fonction possède une entrée et une sortie. Schématiquement, ça donne quelque chose comme ça :



Lorsqu'on appelle une fonction, il y a 3 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) :



Bien entendu, les fonctions seront en général plus compliquées ;)  
Le but des fonctions est donc de simplifier le code source, pour ne pas avoir à retaper le même code plusieurs fois d'affilée.  
  
Rêvez un peu : plus tard, nous créerons par exemple une fonction "afficherFenetre" qui ouvrira une fenêtre à l'écran. Une fois la fonction écrite (c'est l'étape la plus difficile), on n'aura plus qu'à dire "Hep toi la fonction afficherFenetre, ouvre-moi une fenêtre !" ;)  
On pourra aussi écrire une fonction "deplacerPersonnage" dont le but sera de déplacer le personnage d'un jeu à l'écran etc etc :D

### Schéma d'une fonction

Vous avez déjà eu un aperçu de comment est faite une fonction avec la fonction main.  
Cependant pour bien que vous compreniez il va falloir que je vous montre quand même **comment on construit une fonction**.  
  
Voici le schéma d'une fonction, à retenir par coeur :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | type nomFonction(parametres)  {  *// Insérez vos instructions ici*  } |

Vous reconnaissez là un peu la forme de la fonction main.  
Voici ce qu'il faut savoir sur ce schéma :

* **type *(correspond à la sortie)*** : c'est le type de la fonction. 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 2 sortes de fonctions :
  + Les fonctions qui **renvoient une valeur** : on leur met un des types que l'on connaît (char, int, double...)
  + Les fonctions qui **ne renvoient pas de valeur** : on leur met un type spécial "void" (qui signifie "vide").
* **nomFonction**: 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).
* **parametres *(correspond à l'entrée)*** : entre parenthèses, vous pouvez envoyer des paramètres à la fonction. Ce sont des valeurs avec lesquelles la fonction va travailler.  
  Par exemple, pour une fonction "triple", vous envoyez un nombre en paramètre. La fonction "récupère" ce nombre et en calcule le triple, en le multipliant par 3. Elle renvoie ensuite le résultat de ses calculs.

Vous pouvez envoyer autant de paramètres que vous le voulez.  
Vous pouvez aussi n'envoyer aucun paramètre à la fonction, mais ça se fait plus rarement.

* Ensuite vous avez les **accolades**qui indiquent le début et la fin de la fonction. A l'intérieur de ces accolades vous mettrez les instructions que vous voulez. Pour la fonction triple, il faudra taper des instructions qui multiplient par 3 le nombre reçu en entrée.

Une fonction, c'est donc un mécanisme qui reçoit des valeurs en **entrée**(les paramètres) et qui renvoie un résultat en **sortie**.

### Créer une fonction

Voyons voir un exemple pratique sans plus tarder : la fameuse fonction "triple" dont je vous parle depuis tout à l'heure. On va dire que cette fonction reçoit un nombre entier de type long et qu'elle renvoie un nombre entier aussi de type long. Cette fonction calcule le triple du nombre qu'on lui donne :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | long triple(long nombre)  {  long resultat = 0;  resultat = 3 \* nombre; *// On multiplie le nombre qu'on nous a transmis par 3*  **return** resultat; *// On retourne la variable resultat qui vaut le triple de nombre*  } |

Voilà notre première fonction :D  
Une première chose importante : comme vous le voyez, la fonction est de type long. Elle doit donc renvoyer une valeur de type long.  
  
Entre les parenthèses, vous avez les variables que la fonction **reçoit**. Ici, notre fonction triple reçoit une variable de type long appelée "nombre".  
  
La ligne qui indique de "renvoyer une valeur" est celle qui contient le "return". Cette ligne se trouve généralement à la fin de la fonction, après les calculs.  
*return resultat;* dit à la fonction : arrête-toi là et renvoie le nombre "resultat". Cette variable "resultat" DOIT être de type long, car la fonction renvoie un long comme on l'a dit plus haut ;)  
  
La variable resultat est déclarée (= créée) dans la fonction "triple". Cela signifie qu'elle n'est utilisable que dans cette fonction, et pas dans une autre comme la fonction "main" par exemple. C'est donc une variable propre à la fonction "triple".  
  
  
Mais est-ce la façon la plus courte de faire notre fonction triple ?  
Non, on peut faire ça en une ligne en fait ;)  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | long triple(long nombre)  {  **return** 3 \* nombre;  } |

Cette fonction fait exactement la même chose que la fonction de tout à l'heure, elle est juste plus rapide à écrire :)  
Généralement, vos fonctions contiendront plusieurs variables pour effectuer leurs calculs et leurs opérations, rares seront les fonctions aussi courtes que "triple" ;)

### Plusieurs paramètres, aucun paramètre

#### Plusieurs paramètres

Notre fonction "triple" contient un paramètre, mais il est possible de créer des fonctions prenant plusieurs paramètres.  
Par exemple, une fonction addition qui additionne deux nombres a et b :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | long addition(long a, long b)  {  **return** a + b;  } |

Il suffit de séparer les différents paramètres par une virgule comme vous le voyez ;)

#### Aucun paramètre

Certaines fonctions, plus rares, ne prennent aucun paramètre en entrée. Ces fonctions feront généralement toujours la même chose. En effet, si elles n'ont pas de nombres sur lesquels travailler, vos fonctions serviront juste à effectuer certaines actions, comme afficher du texte à l'écran (et encore, ce sera toujours le même texte !)  
  
Imaginons une fonction "bonjour" qui affiche juste bonjour à l'écran :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | void bonjour()  {  printf("Bonjour");  } |

Je n'ai rien mis entre parenthèses car la fonction ne prend aucun paramètre.  
De plus, j'ai mis le type "void" dont je vous ai parlé plus haut.  
  
En effet, comme vous le voyez ma fonction n'a pas non plus de "return". Elle ne retourne rien. Une fonction qui ne retourne rien est de type void.

### Appeler une fonction

On va maintenant tester un code source pour pratiquer un peu avec ce qu'on vient d'apprendre.  
Nous allons utiliser notre fonction "triple" (décidemment je l'aime bien) pour calculer le triple d'un nombre.  
  
Pour le moment, je vous demande d'écrire la fonction "triple" AVANT la fonction main. Si vous la mettez après, ça ne marchera pas. Je vous expliquerai pourquoi par la suite ;)  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | #include <stdio.h>  #include <stdlib.h>  long triple(long nombre)  {  **return** 3 \* nombre;  }  int main(int argc, char \*argv[])  {  long nombreEntre = 0, nombreTriple = 0;    printf("Entrez un nombre... ");  scanf("%ld", &nombreEntre);    nombreTriple = triple(nombreEntre);  printf("Le triple de ce nombre est %ld**\n**", nombreTriple);    system("PAUSE");  **return** 0;  } |

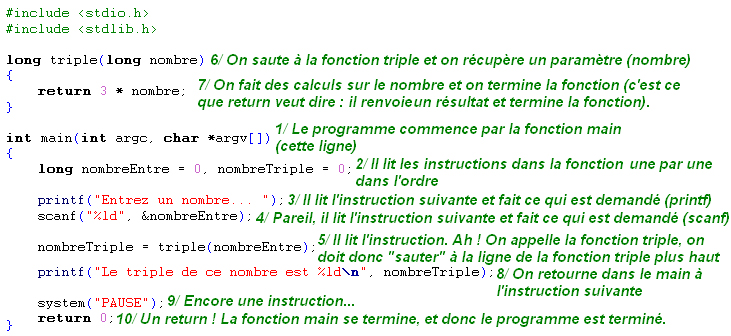
Notre programme commence par la fonction main comme vous le savez.  
On demande à l'utilisateur de rentrer un nombre. On envoie ce nombre qu'il a rentré à la fonction triple, et on récupère le résultat dans la variable nombreTriple. Regardez en particulier cette ligne, c'est la plus intéressante car c'est l'*appel de la fonction*:  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | nombreTriple = triple(nombreEntre); |

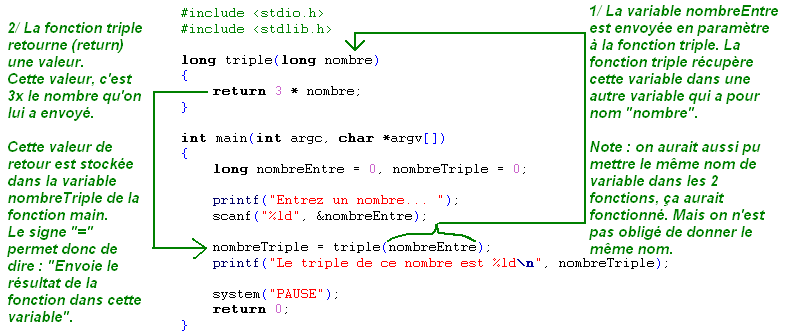
Entre parenthèses, on envoie une variable en **entrée** à la fonction triple, c'est le nombre sur lequel elle va travailler.  
Cette fonction renvoie une valeur, valeur qu'on récupère dans la variable nombreTriple. On demande donc à l'ordinateur dans cette ligne : "Demande à la fonction triple de me calculer le triple de nombreEntre, et stocke le résultat dans la variable nombreTriple".

#### Les mêmes explications sous forme de schéma

Vous avez encore du mal à comprendre comment ça fonctionne concrètement ?  
Pas de panique ! Je suis sûr que vous allez comprendre avec mes schémas :)  
  
Ce premier schéma vous explique dans quel ordre le code est lu. Commencez donc par lire la ligne numérotée "1", puis "2", puis "3" (bon vous avez compris je crois :D )



Si vous avez compris dans quel ordre l'ordinateur lisait les instructions, vous avez déjà compris le principal :)  
  
  
Maintenant, il faut bien comprendre qu'une fonction reçoit des paramètres en entrée et renvoie une valeur en sortie.



**Note**: ce n'est pas tout le temps le cas comme ça pour toutes les fonctions. Parfois, une fonction ne prend aucun paramètre en entrée, ou au contraire elle en prend plusieurs (je vous ai expliqué ça un peu plus haut).  
De même, parfois une fonction renvoie une valeur, parfois elle ne renvoie rien (dans ce cas il n'y a pas de *return*).

#### Testons ce programme

Voici un exemple d'utilisation du programme (y'a rien de bien extraordinaire car c'est une fonction toute bête hein ;) ) :  
  
**Code : Console**

|  |
| --- |
| Entrez un nombre... 10  Le triple de ce nombre est 30 |

Vous n'êtes pas obligés de stocker le résultat d'une fonction dans une variable ! Vous pouvez directement envoyer le résultat de la fonction triple à une autre fonction, comme si triple(nombreEntre) était une variable.

Regardez bien ceci, c'est le même code mais y'a un changement au niveau du dernier printf, et on n'a pas déclaré de variable nombreTriple car on ne s'en sert plus :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | #include <stdio.h>  #include <stdlib.h>  long triple(long nombre)  {  **return** 3 \* nombre;  }  int main(int argc, char \*argv[])  {  long nombreEntre = 0;    printf("Entrez un nombre... ");  scanf("%ld", &nombreEntre);  *// Le résultat de la fonction est directement envoyé au printf et n'est pas stocké dans une variable*  printf("Le triple de ce nombre est %ld**\n**", triple(nombreEntre));    system("PAUSE");  **return** 0;  } |

Comme vous le voyez, triple(nombreEntre) est directement envoyé au printf.  
Que fait l'ordinateur quand il tombe sur cette ligne ?  
  
C'est très simple. Il voit que la ligne commence par printf, il va donc appeler la fonction printf.  
Il envoie à la fonction printf tous les paramètres qu'on lui donne. Le premier paramètre est le texte à afficher, et le second est un nombre.  
Votre ordinateur voit que pour envoyer ce nombre à la fonction printf il doit d'abord appeler la fonction triple. C'est ce qu'il fait : il appelle triple, il effectue les calculs de triple, et une fois qu'il a le résultat il l'envoie directement dans la fonction printf !  
  
C'est un peu une imbrication de fonctions ;)  
  
Et le top, c'est qu'une fonction peut en appeler une autre à son tour !  
Notre fonction triple pourrait appeler une autre fonction, qui elle-même appellerait une autre fonction etc... C'est ça le principe de la programmation en C ! Tout est combiné, comme dans un jeu de Legos ;)  
  
Au final, le plus dur sera d'écrire vos fonctions. Une fois que vous les aurez écrites, vous n'aurez plus qu'à appeler les fonctions sans vous soucier des calculs qu'elles peuvent bien faire à l'intérieur. Ca va permettre de simplifier considérablement l'écriture de nos programmes, et ça croyez-moi on en aura bien besoin ! :)

## Plein d'exemples pour bien comprendre

Vous avez dû vous en rendre compte : je suis un maniaque des exemples.  
La théorie c'est bien, mais si on ne fait que ça on risque de ne pas retenir grand chose, et surtout ne pas comprendre comment s'en servir, ce qui serait un peu dommage :p  
  
Je vais donc maintenant vous montrer plusieurs exemples d'utilisation de fonctions, pour que vous ayez une idée de leur intérêt. Je vais m'efforcer de faire des cas différents à chaque fois, pour que vous puissiez avoir des exemples de tous les types de fonctions qui peuvent exister.  
  
Je ne vous apprendrai pas grand chose de nouveau, mais ça sera l'occasion de voir des exemples pratiques. Si vous avez déjà compris tout ce que j'ai expliqué avant, c'est très bien et normalement aucun des exemples qui vont suivre ne devrait vous surprendre ;)

### Conversion euros / francs

On commence par une fonction très similaire à "triple", qui a quand même un minimum d'intérêt cette fois : une fonction qui convertit les euros en francs.  
Pour ceux d'entre vous qui ne connaîtraient pas ces monnaies (il n'y a pas que des français sur le Site du Zér0 hein ;) ) sachez que :  
  
**1 euro = 6.55957 francs**  
  
On va créer une fonction appelée conversion.  
Cette fonction prend une variable en entrée de type double et retourne une sortie de type double (on va forcément manipuler des chiffres décimaux !).  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | double conversion(double euros)  {  double francs = 0;    francs = 6.55957 \* euros;  **return** francs;  }  int main(int argc, char \*argv[])  {  printf("10 euros = %lfF**\n**", conversion(10));  printf("50 euros = %lfF**\n**", conversion(50));  printf("100 euros = %lfF**\n**", conversion(100));  printf("200 euros = %lfF**\n**", conversion(200));    system("PAUSE");  **return** 0;  } |

**Code : Console**

|  |
| --- |
| 10 euros = 65.595700F  50 euros = 327.978500F  100 euros = 655.957000F  200 euros = 1311.914000F |

Il n'y a pas grand chose de différent par rapport à la fonction triple je vous avais prévenu :p  
D'ailleurs, ma fonction conversion est un peu longue et pourrait être raccourcie en une ligne, je vous laisse le faire je vous ai déjà expliqué comment faire plus haut.  
  
Dans la fonction main, j'ai fait exprès de faire plusieurs printf pour vous montrer l'intérêt d'avoir une fonction. Pour obtenir la valeur de 50 euros, je n'ai qu'à écrireconversion(50). Et si je veux avoir la conversion en francs de 100 euros, j'ai juste besoin de changer le paramètre que j'envoie à la fonction (100 au lieu de 50).  
  
  
**A vous de jouer !** Ecrivez une seconde fonction (toujours avant la fonction main) qui fera elle la conversion inverse : Francs => Euros. Pas bien difficile hein, y'a juste un signe d'opération à changer ;)

### La punition

On va maintenant s'intéresser à une fonction qui ne renvoie rien (pas de sortie).  
C'est une fonction qui affiche le même message à l'écran autant de fois qu'on lui demande. Cette fonction prend un paramètre en entrée : le nombre de fois où il faut afficher la punition.  
  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | void punition(long nombreDeLignes)  {  long i;    **for** (i = 0 ; i < nombreDeLignes ; i++)  {  printf("Je ne dois pas recopier mon voisin**\n**");  }  }  int main(int argc, char \*argv[])  {  punition(10);    system("PAUSE");  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Je ne dois pas recopier mon voisin  Je ne dois pas recopier mon voisin  Je ne dois pas recopier mon voisin  Je ne dois pas recopier mon voisin  Je ne dois pas recopier mon voisin  Je ne dois pas recopier mon voisin  Je ne dois pas recopier mon voisin  Je ne dois pas recopier mon voisin  Je ne dois pas recopier mon voisin  Je ne dois pas recopier mon voisin |

On a ici affaire à une fonction qui ne renvoie aucune valeur. Cette fonction se contente juste d'effectuer des actions (ici, elle affiche des messages à l'écran).  
Une fonction qui ne renvoie aucune valeur est de type "void", c'est pour cela qu'on a écrit void.  
  
A part ça, rien de bien différent ;)  
  
Il aurait été bien plus intéressant de créer une fonction "punition" qui s'adapte à n'importe quelle punition. On lui aurait envoyé 2 paramètres : le texte à répéter, et le nombre de fois qu'il doit être répété. Le problème, c'est qu'on ne sait pas encore gérer le texte en C (au cas où vous auriez pas les yeux très ouverts, je vous rappelle qu'on fait que manipuler des variables contenant des nombres depuis le début :p )  
D'ailleurs à ce sujet, je vous annonce que nous ne tarderons pas à apprendre à utiliser des variables qui retiennent du texte. C'est que c'est plus compliqué qu'il n'y paraît, et on ne pouvait pas l'apprendre dès le début du cours ;)

### Aire d'un rectangle

L'aire d'un rectangle est facile à calculer : largeur \* hauteur.  
Notre fonction aireRectangle va prendre 2 paramètres, la largeur et la hauteur. Elle renverra l'aire :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | double aireRectangle(double largeur, double hauteur)  {  **return** largeur \* hauteur;  }  int main(int argc, char \*argv[])  {  printf("Rectangle de largeur 5 et hauteur 10. Aire = %lf**\n**", aireRectangle(5, 10));  printf("Rectangle de largeur 2.5 et hauteur 3.5. Aire = %lf**\n**", aireRectangle(2.5, 3.5));  printf("Rectangle de largeur 4.2 et hauteur 9.7. Aire = %lf**\n**", aireRectangle(4.2, 9.7));    system("PAUSE");  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Rectangle de largeur 5 et hauteur 10. Aire = 50.000000  Rectangle de largeur 2.5 et hauteur 3.5. Aire = 8.750000  Rectangle de largeur 4.2 et hauteur 9.7. Aire = 40.740000 |

Pourrait-on afficher directement la largeur, la hauteur et l'aire dans la fonction ?

Bien sûr !  
Dans ce cas, la fonction ne renverrait plus rien, elle se contenterait de calculer l'aire et de l'afficher immédiatement.  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | void aireRectangle(double largeur, double hauteur)  {  double aire = 0;    aire = largeur \* hauteur;  printf("Rectangle de largeur %lf et hauteur %lf. Aire = %lf**\n**", largeur, hauteur, aire);  }  int main(int argc, char \*argv[])  {  aireRectangle(5, 10);  aireRectangle(2.5, 3.5);  aireRectangle(4.2, 9.7);    system("PAUSE");  **return** 0;  } |

Comme vous le voyez, le printf est *à l'intérieur* de la fonction aireRectangle et fait le même affichage que tout à l'heure. C'est juste une façon différente de procéder ;)

### Un menu

Ce code est plus intéressant et concret. On crée une fonction menu() qui ne prend aucun paramètre en entrée. Cette fonction se contente d'afficher le menu, et demande à l'utilisateur de faire un choix.  
La fonction renvoie le choix de l'utilisateur.  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | long menu()  {  long choix = 0;    **while** (choix < 1 || choix > 4)  {  printf("Menu :**\n**");  printf("1 : Poulet de dinde aux escargots rotis a la sauce bearnaise**\n**");  printf("2 : Concombres sucres a la sauce de myrtilles enrobee de chocolat**\n**");  printf("3 : Escalope de kangourou saignante et sa gelee aux fraises poivree**\n**");  printf("4 : La surprise du Chef (j'en salive d'avance...)**\n**");  printf("Votre choix ? ");  scanf("%ld", &choix);  }    **return** choix;  }  int main(int argc, char \*argv[])  {  **switch** (menu())  {  **case** 1:  printf("Vous avez pris le poulet**\n**");  **break**;  **case** 2:  printf("Vous avez pris les concombres**\n**");  **break**;  **case** 3:  printf("Vous avez pris l'escalope**\n**");  **break**;  **case** 4:  printf("Vous avez pris la surprise du Chef. Vous etes un sacre aventurier dites donc !**\n**");  **break**;  }    system("PAUSE");  **return** 0;  } |

J'en ai profité pour améliorer le menu (par rapport à ce qu'on faisait habituellement) : la fonction menu réaffiche le menu tant que l'utilisateur n'a pas rentré un nombre compris entre 1 et 4. Comme ça, aucun risque que la fonction renvoie un nombre qui ne figure pas au menu !  
  
Dans le main, vous avez vu qu'on fait un switch(menu()). Une fois que la fonction menu() est terminée, elle renvoie le choix de l'utilisateur directement dans le switch. C'est pratique et rapide comme méthode :)  
  
**A vous de jouer !** Le code est encore améliorable : on pourrait afficher un message d'erreur si l'utilisateur rentre un mauvais nombre plutôt que de bêtement réafficher le menu ;)

## Partie 2 : [Théorie] Techniques avancées

## Les prototypes

Jusqu'ici, je vous ai demandé de placer votre fonction *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.  
Si vous mettez votre fonction après le main, ça ne marchera pas car l'ordinateur ne connaîtra pas encore la fonction. Essayez vous verrez ;)

Mais... C'est un peu nul non ?

Tout à fait d'accord avec vous :D  
Mais rassurez-vous, les programmeurs s'en sont rendu compte avant vous et ont prévu le coup ;)  
  
Grâce à ce que je vais vous apprendre maintenant, vous pourrez mettre vos fonctions dans n'importe quel ordre dans le code source. C'est mieux de ne pas avoir à s'en soucier, croyez-moi ^^

### Le prototype pour annoncer une fonction

Nous allons "annoncer" nos fonctions à l'ordinateur en écrivant ce qu'on appelle des **prototypes**. Ne soyez pas intimidés par ce nom high-tech, ça cache en fait quelque chose de tout bête ;)  
  
Regardez la première ligne de notre fonction aireRectangle :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | double aireRectangle(double largeur, double hauteur)  {  **return** largeur \* hauteur;  } |

Copiez la première ligne (double aireRectangle...) tout en haut de votre fichier source (juste après les #include).  
**Rajoutez un point-virgule** à la fin de cette nouvelle ligne.  
Et voilà ! Maintenant vous pouvez mettre votre fonction aireRectangle après la fonction main si vous le voulez ;)  
  
Vous devriez avoir le code suivant sous les yeux :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | #include <stdio.h>  #include <stdlib.h>  *// La ligne suivante est le prototype de la fonction aireRectangle :*  double aireRectangle(double largeur, double hauteur);  int main(int argc, char \*argv[])  {  printf("Rectangle de largeur 5 et hauteur 10. Aire = %lf**\n**", aireRectangle(5, 10));  printf("Rectangle de largeur 2.5 et hauteur 3.5. Aire = %lf**\n**", aireRectangle(2.5, 3.5));  printf("Rectangle de largeur 4.2 et hauteur 9.7. Aire = %lf**\n**", aireRectangle(4.2, 9.7));    **return** 0;  }  *// Notre fonction aireRectangle peut maintenant être mise n'importe où dans le code source :*  double aireRectangle(double largeur, double hauteur)  {  **return** largeur \* hauteur;  } |

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.  
Ca permet à l'ordinateur de s'organiser.  
  
Grâce à cette ligne, vous pouvez maintenant mettre vos fonctions dans n'importe quel ordre sans vous prendre la tête ;)  
  
**Ecrivez toujours le prototype de vos fonctions**. Vos programmes ne vont pas tarder à se complexifier et à utiliser plein de fonctions : mieux vaut prendre dès maintenant la bonne habitude de mettre un prototype pour chacune de vos fonctions ;)  
  
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 (c'est toujours la même pour tous les programmes, alors il peut bien la connaître à force :lol: )

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 :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | double aireRectangle(double, double); |

Toutefois, l'autre méthode que je vous ai montrée tout à l'heure fonctionne aussi bien. L'avantage avec ma méthode c'est que vous avez juste besoin de copier-coller la première ligne de la fonction et de rajouter un point-virgule. Ca va plus vite ;)

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 :p

## Les headers

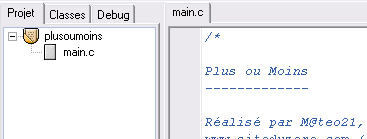
Jusqu'ici, nous n'avions qu'un seul fichier source dans notre projet. Ce fichier source, je vous avais demandé de l'appeler main.c

### Plusieurs fichiers par projet

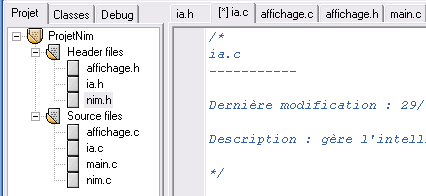
Dans la pratique, vos programmes ne seront pas tous écrits dans ce même fichier main.c. Bien sûr, c'est *possible* de le faire, mais ce n'est jamais très pratique se ballader dans un fichier de 10000 lignes (enfin personnellement je trouve :p ).  
C'est pour cela qu'en général on crée plusieurs fichiers par projet.

Euh c'est quoi un projet ?

Non, vous avez pas déjà oublié ? o_O  
Bon allez je vous le réexplique, parce qu'il est important qu'on soit bien d'accord entre nous là ^^  
  
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 sur cette capture d'écran à gauche, notre projet "plusoumoins" n'était composé que d'un fichier main.c.  
Ca c'était parce que c'était notre premier TP et qu'il était tout bête :p  
  
Laissez-moi maintenant vous montrer un vrai projet que j'avais réalisé il y a quelques temps pour un devoir d'école (un jeu d'allumettes) :



Comme vous le voyez, il y a plusieurs fichiers. Un vrai projet ressemblera à ça : vous verrez plusieurs fichiers dans la colonne de gauche.  
En fait, c'était un "assez petit" projet ça encore. Les gros programmes du commerce ont sûrement beaucoup plus de fichiers que ça, mais c'était pour vous donner une idée ^^  
  
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 (mais ce n'est pas du tout une obligation, chacun s'organise comme il veut !)

Mais pourquoi avoir créé plusieurs fichiers ? Et comment je sais combien de fichiers je dois créer pour mon projet ?

Ca c'est vous qui choisissez ;)  
En général, on regroupe dans un même fichier des fonctions par thème. Ainsi, dans le fichier affichage.c j'ai regroupé toutes les fonctions gérant l'affichage à l'écran, dans le fichier ia.c j'ai regroupé toutes les fonctions gérant l'intelligence artificielle de l'ordinateur etc etc...

Pour la petite histoire, j'ai connu un professeur qui voulait qu'on mette UNE seule fonction par fichier :lol:  
Sans aller jusqu'à de tels extrêmes, sachez que tout est une question de dosage. Essayez de faire des fichiers avec plusieurs fonctions, sans qu'il y en ait trop à la fois (sinon on s'y perd) ou pas assez (sinon vous risquez d'avoir trop de fichiers par projet, et là aussi vous vous y perdrez ;) )

### Fichiers .h et .c

Comme vous le voyez, il y a 2 types de fichiers différents sur la capture d'écran que je vous ai montrée :

* **Les .h** : appelés fichiers headers. Ces fichiers contiennent les prototypes des fonctions.
* **Les .c** : les 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).  
  
Pour chaque fichier .c, il y a son équivalent .h qui contient les prototypes des fonctions.  
Rejetez un oeil à ma capture d'écran :

* Il y a ia.c (le code des fonctions) et ia.h (les prototypes des fonctions)
* Il y a affichage.c et affichage.h
* etc.

Mais comment faire pour que l'ordinateur sache que les prototypes sont dans un autre fichier que le .c ?

Il faut **inclure** le fichier .h grâce à une directive de préprocesseur.  
Attention, préparez-vous à comprendre plein de trucs tout d'un coup :D  
  
Comment inclure un fichier header ?  
Vous savez le faire, vous l'avez déjà fait !  
  
Regardez par exemple le début de mon fichier affichage.c :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #include <stdio.h>  #include <stdlib.h>  #include "affichage.h" *// On inclut affichage.h*  *// Affiche une ligne d'allumettes à l'écran, de la couleur demandée*  void afficherLigneDAllumettes (long numeroDuTas, long nombreTotalDAllumettes, long nombreDAllumettesAColorier, long couleur, int effacerLigne)  {  long ordonneeActuelleDansLAllumette, abscisseAllumetteActuelle, allumetteActuelle,  ordonneeHautAllumette, abscisseGaucheAllumette, abscisseActuelle;    ordonneeHautAllumette = ((numeroDuTas - 1) \* 5) + 2;  abscisseGaucheAllumette = 20 + (60 - (nombreTotalDAllumettes \* 4)) / 2;        **if** (effacerLigne)  {  *// Reste du code...* |

L'inclusion se fait grâce à la directive de préprocesseur #include que vous connaissez bien maintenant :)  
Regardez les premières lignes du code source ci-dessus :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3 | #include <stdio.h>  #include <stdlib.h>  #include "affichage.h" *// On inclut affichage.h* |

On inclut 3 fichiers .h : stdio, stdlib et affichage.  
**Notez une différence** : les fichiers que vous avez créés et placés dans le répertoire de votre projet doivent être inclus avec des guillemets ("affichage.h") tandis que les fichiers correspondants aux librairies (qui sont installés, eux, dans le répertoire de votre IDE généralement) sont inclus entre chevrons (<stdio.h>).  
  
Vous utiliserez donc :

* **Les chevrons < >** pour inclure un fichier se trouvant dans le répertoire "include" de votre IDE
* **Les guillemets " "** pour inclure un fichier se trouvant dans le répertoire de votre projet (à côté des .c généralement ;) )

La commande #include demande d'insérer le contenu du fichier dans le .c. C'est donc une commande qui dit "Insère ici le fichier affichage.h" par exemple.  
  
Et dans le fichier affichage.h que trouve-t-on ?  
On trouve juste les prototypes des fonctions du fichier affichage.c !  
  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | */\**  *affichage.h*  *-----------*  *Par Mathieu et Jonathan*  *Dernière modification : 29/02/04*  *Description : gère l'affichage graphique du jeu*  *\*/*  *// Affiche une ligne d'allumettes à l'écran, de la couleur demandée*  void afficherLigneDAllumettes (long numeroDuTas, long nombreTotalDAllumettes,  long nombreDAllumettesAColorier, long couleur, int effacerLigne);  *// Place le curseur lors de la partie à la position définie*  void afficherCurseurDuJeu(long position);  void afficherInfosJeu(long nombreDAllumettes, long numeroDuTas, int tourDuPremierJoueur, long couleur, int jeuContreLOrdinateur);  *// Affiche le menu d'accueil*  void afficherMenu(); |

Voilà comment fonctionne un vrai projet ;)

Quel 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 mettez 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 lesquelles les fonctions se trouvent dans vos fichiers .c  
Si vous faites un petit programme maintenant contenant 2-3 fonctions, vous vous rendrez compte que les prototypes semblent facultatifs (ça marche sans). 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 plantera lamentablement :p

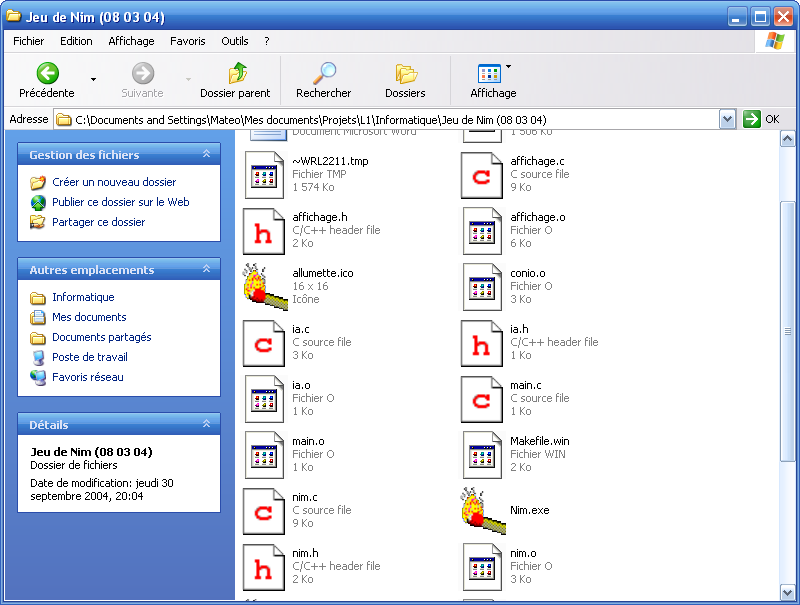
Lorsque vous appelez une fonction située dans fonctions.c depuis le fichier main.c, vous aurez besoin d'inclure les prototypes de fonctions.c dans main.c. Il faudra donc mettre un #include "fonctions.h" en haut de main.c  
Souvenez-vous de ceci : à chaque fois que vous faites appel à une fonction X dans 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.

Comment j'ajoute des fichiers .c et .h à mon projet ?

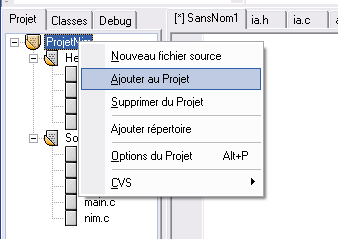
Ca dépend de l'IDE que vous utilisez, mais globalement la procédure est la même : Fichier / Nouveau / Fichier source.  
Cela crée un nouveau fichier vide. Ce fichier n'est pas encore de type .c ou .h, il faut que vous l'enregistriez pour le dire. Enregistrez donc ce nouveau fichier (même s'il est encore vide !). On vous demandera alors quel nom vous voulez donner au fichier. C'est là que vous choisissez si c'est un .c ou un .h :

* Si vous l'appelez fichier.c, ce sera un .c
* Si vous l'appelez fichier.h, ce sera un .h

C'est aussi simple que cela ;)  
Enregistrez votre fichier dans le répertoire où se trouvent les autres fichiers de votre projet (le même dossier que main.c). Généralement, **vous enregistrerez tous vos fichiers dans le même répertoire, les .c comme les .h**.  
  
Le dossier de mon jeu d'allumettes ressemble donc au final à ça :



Vous y voyez des .c et des .h ensemble.  
Vous notez aussi qu'il y a des .o, nous allons voir juste après ce que c'est.  
  
Bref, maintenant votre fichier est enregistré, mais il n'est pas encore vraiment ajouté au projet !  
Pour l'ajouter au projet, faites un clic droit dans la partie à gauche de l'écran (où il y a la liste des fichiers du projet) et choisissez "Ajouter au projet" :



Une fenêtre s'ouvre et vous demande quels fichiers ajouter au projet. Sélectionnez le fichier que vous venez de créer, et c'est fait :)  
Le fichier fait maintenant partie du projet et apparaît dans la liste à gauche !

Certains IDE, comme Code::Blocks, séparent automatiquement les .c et les .h dans la liste à gauche. D'autres IDE, comme Dev-C++, permettent de le faire mais ne les séparent pas automatiquement. Pour faire cela, vous pouvez ajouter un "répertoire" en faisant clic droit / ajouter répertoire (regardez sur ma capture d'écran ci-dessus).  
Cela est surtout utile quand vous avez beaucoup de fichiers source et que vous voulez les organiser pour pas vous perdre dans votre projet. Ca ne change rien au programme final.

### Les includes des librairies standard

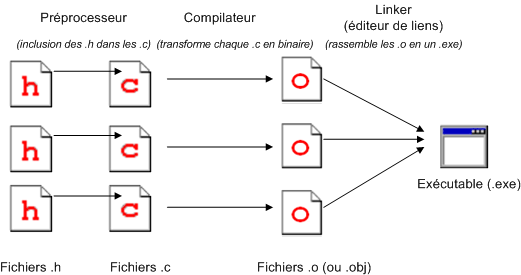
Une question devrait vous trotter dans la tête...  
Si on inclut les fichiers stdio.h et stdlib.h, c'est donc qu'ils existent quelque part et qu'on peut aller les chercher non ?  
  
Oui bien sûr !  
Ils sont installés normalement là où se trouve votre IDE. Dans mon cas sous Dev C++, je les trouve là :  
  
C:\Program Files\Dev-Cpp\include  
  
Il faut généralement chercher un dossier include.  
Là-dedans, vous allez trouver plein plein de fichiers. Ce sont des headers (.h) des librairies standard, c'est-à-dire des librairies disponibles partout (que ce soit sous Windows, Mac, Linux...). Vous y retrouverez donc stdio.h et stdlib.h entre autres.  
  
Vous pouvez les ouvrir si vous voulez, mais prévoyez une bassine à côté on sait jamais :-°  
En effet, c'est un peu compliqué et ça peut donner la nausée (il y a pas mal de choses qu'on n'a pas encore vues, notamment pas mal de directives de préprocesseur). Si vous cherchez bien, vous verrez que ce fichier est rempli de prototypes de fonctions standard, comme printf par exemple.

Ok, je sais maintenant où se trouvent les prototypes des fonctions standard. Mais je pourrai pas aussi voir le code source de ces fonctions ? Où sont les .c ?!

Ils n'existent pas :D  
En fait, les fichiers .c sont déjà compilés (en code binaire, c'est-à-dire en code machine). Il est donc totalement impossible de les lire.  
  
Vous pouvez retrouver les fichiers compilés dans un répertoire appelé "lib" généralement (pour "library"). Chez moi ils se trouvent dans :  
  
C:\Program Files\Dev-Cpp\lib  
  
Les fichiers compilés des librairies ont l'extension .a sous Dev (qui utilise le compilateur appelé mingw), et ont l'extension .lib sous Visual C++ (qui utilise le compilateur Visual).  
N'essayez pas de les lire c'est totalement pas comestible ;)  
  
  
Voilà vous savez maintenant un peu mieux comment ça fonctionne j'espère :)  
Dans vos fichiers .c, vous incluez les .h des librairies standard pour pouvoir utiliser des fonctions standard comme printf. Votre ordinateur a ainsi les prototypes sous les yeux et peut vérifier si vous appelez les fonctions correctement (si vous n'oubliez pas de paramètres par exemple).

## La compilation séparée

Maintenant que vous savez qu'un projet est composé de plusieurs fichiers sources, nous pouvons rentrer plus en détail dans le fonctionnement de la compilation. Jusqu'ici, nous avions vu un schéma très simplifié.  
  
Voici un schéma plus précis de la compilation. Croyez-moi, celui-là il vaut mieux le connaître par coeur ! :D



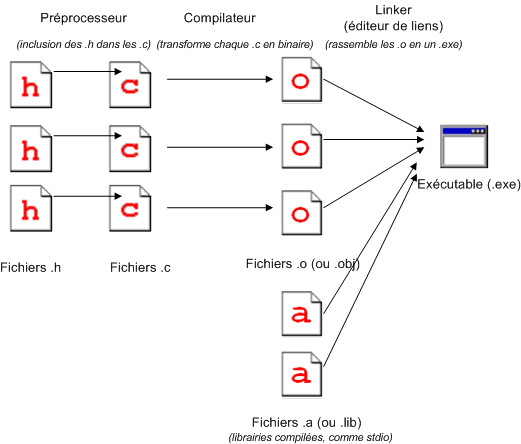
Ca c'est un vrai schéma de ce qu'il se passe à la compilation.  
Allez, je vous détaille ça dans l'ordre ^^

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 #.  
   Pour l'instant, la seule directive de préprocesseur que l'on connaît est #include, qui permet d'inclure un fichier dans un autre. Le préprocesseur sait faire d'autres choses, mais ça nous le verrons plus tard. Le #include est quand même ce qu'il y a de plus important ;)  
   Le préprocesseur "remplace" donc les lignes #include par le fichier indiqué. Il met à l'intérieur de chaque fichier .c les fichiers .h qu'on a demandé d'inclure.  
   A ce moment-ci de la compilation, votre fichier .c est complet et contient tous les prototypes des fonctions que vous utilisez (votre fichier .c est donc un peu plus gros que la normale).
2. **Compilation** : cette étape très importante consiste à transformer vos fichiers sources 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 (ils doivent tous apparaître dans la fameuse liste à gauche ;) )  
   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 que vous mettez vous pouvez choisir de les garder (mais ça sert à rien :lol: )
3. **Edition de liens** : le linker (ou "éditeur de liens" en français) est un programme dont le rôle est d'assembler les fichiers binaires .o. 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 :)

Et voilà, maintenant vous savez comment ça se passe à l'intérieur :D  
Je le dis et je le répète, ce schéma est super important. Il fait la différence entre un programmeur du dimanche qui copie à l'arrache des codes sources et un programmeur qui sait et comprend ce qu'il fait ;)  
  
  
La plupart des erreurs surviennent à la compilation, mais il m'est arrivé aussi d'avoir des erreurs de linker. Cela signifie que le linker n'est pas arrivé à assembler tous les .o (il en manquait peut-être).

### Lorsque vous utilisez des librairies

Notre schéma est par contre encore un peu incomplet. En effet, les librairies n'apparaissent pas dedans !  
Comment cela se passe-t-il quand on utilise des librairies ?  
  
En fait le début du schéma reste le même, c'est seulement le linker qui va avoir un peu plus de travail. Il va assembler vos .o (temporaires) avec les librairies compilées dont vous avez besoin (.a ou .lib selon le compilateur) :



Nous y sommes, le schéma est cette fois complet :)  
Vos fichiers de librairies .a (ou .lib) sont rassemblés dans l'exécutable avec vos .o  
  
C'est comme cela qu'on peut obtenir au final un programme 100% complet, qui contient toutes les instructions nécessaires à l'ordinateur, même celles *qui lui expliquent comment afficher du texte* !  
Par exemple la fonction printf se trouve dans un .a, et donc sera rassemblée avec votre code source dans l'exécutable.  
  
Dans quelques temps, nous apprendrons à utiliser des librairies graphiques. Celles-ci seront là aussi dans des .a et contiendront des instructions pour indiquer à l'ordinateur comment ouvrir une fenêtre à l'écran par exemple. Mais, patience, car tout vient à point à qui sait attendre c'est bien connu ;)

## La portée des fonctions et variables

Pour terminer ce chapitre, je vais vous parler de ce qu'on appelle **la 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.

### Les variables propres aux fonctions

Lorsque vous déclarez une variable dans une fonction, celle-ci est supprimée de la mémoire à la fin de la fonction :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | long triple(long nombre)  {  long resultat = 0; *// La variable resultat est créée en mémoire*    resultat = 3 \* nombre;  **return** resultat;  } *// La fonction est terminée, la variable resultat est supprimée de la mémoire* |

Une variable déclarée dans une fonction n'existe donc que pendant que la fonction est exécutée.  
Qu'est-ce que ça veut dire concrètement ? Que vous ne pouvez pas y accéder depuis une autre fonction !  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | long triple(long nombre);    int main(int argc, char \*argv[])  {  printf("Le triple de 15 est %ld**\n**", triple(15));    printf("Le triple de 15 est %ld", resultat); *// Cette ligne plantera à la compilation*    **return** 0;  }    long triple(long nombre)  {  long resultat = 0;    resultat = 3 \* nombre;  **return** resultat;  } |

Dans le main, j'essaie d'accéder à la variable résultat. Or, comme cette variable résultat a été créée dans la fonction triple, elle n'est pas accessible dans la fonction main !  
  
**Retenez** : une variable déclarée dans une fonction n'est accessible qu'à l'intérieur de cette fonction.  
On dit que c'est une variable locale.

### Les variables globales : à éviter

#### Variable globale accessible dans tous les fichiers

Il est possible de déclarer des variables qui sont 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. Ca aura l'air de simplifier votre code au début, mais après vous risquez de vous retrouver avec plein de variables accessibles partout, ce qui risquera de vous poser des soucis.  
  
Pour déclarer une variable "globale" accessible partout, vous devez faire la déclaration de la variable **en-dehors** des fonctions. Vous ferez la déclaration tout en haut du fichier, après les #include généralement.  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #include <stdio.h>  #include <stdlib.h>    long resultat = 0; *// Déclaration de variable globale*    void triple(long nombre); *// Prototype de fonction*    int main(int argc, char \*argv[])  {  triple(15); *// On appelle la fonction triple, qui modifie la variable globale resultat*  printf("Le triple de 15 est %ld**\n**", resultat); *// On a accès à resultat*    **return** 0;  }    void triple(long nombre)  {  resultat = 3 \* nombre;  } |

Sur cet exemple, ma fonction triple ne renvoie plus rien (void). Elle se contente de modifier la variable globale resultat que la fonction main peut récupérer.  
  
Ma variable resultat sera accessible dans tous les fichiers du projet, donc on pourra faire appel à elle dans TOUTES les fonctions du programme.

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.

#### Variable globale accessible uniquement dans un fichier

La variable globale de tout à l'heure était accessible dans tous les fichiers du projet.  
Il est possible de la rendre accessible uniquement dans le fichier où elle se trouve. Ca 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 juste le mot-clé static devant :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | **static** long resultat = 0; |

### Variable statique à une fonction

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**.  
  
Par exemple :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | long triple(long nombre)  {  **static** long resultat = 0; *// La variable resultat est créée la première fois que la fonction est appelée*    resultat = 3 \* nombre;  **return** resultat;  } *// La variable resultat n'est PAS supprimée lorsque la fonction est terminée.* |

Qu'est-ce que ça signifie concrètement ?  
Qu'on pourra rappeler la fonction plus tard et la variable resultat contiendra toujours la valeur de la dernière fois.  
  
Voici un petit exemple pour bien comprendre :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | long incremente();    int main(int argc, char \*argv[])  {  printf("%ld**\n**", incremente());  printf("%ld**\n**", incremente());  printf("%ld**\n**", incremente());  printf("%ld**\n**", incremente());    **return** 0;  }    long incremente()  {  **static** long nombre = 0;    nombre++;  **return** nombre;  } |

**Code : Console**

|  |
| --- |
| 1  2  3  4 |

Ici, la première fois qu'on appelle la fonction incremente, 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...  
  
Ce type de variable est assez rarement utilisé, mais ça peut vous servir à l'occasion donc je tenais à vous le présenter ;)

### 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 où se trouve la fonction.  
  
Pour faire cela, rajoutez le mot-clé static (encore lui) devant la fonction :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | **static** long triple(long nombre)  {  *// Instructions*  } |

Pensez à mettre à jour le prototype aussi :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | **static** long triple(long nombre); |

Et voilà ! Votre fonction "static" triple ne peut être appelée que depuis une autre fonction du même fichier (par exemple main.c).  
Si vous essayez d'appeler la fonction triple depuis une fonction d'un autre fichier (par exemple affichage.c), ça ne marchera pas car "triple" n'y sera pas accessible ;)

### On résume !

#### Portée des variables

* Une variable déclarée dans une fonction est supprimée à la fin de la fonction, elle n'est **accessible que dans cette fonction**.
* Une variable déclarée dans une fonction avec le mot-clé static devant n'est pas supprimée à la fin de la fonction, **elle conserve sa valeur au fur et à mesure de l'exécution du programme**
* Une variable déclarée en-dehors des fonctions est une variable globale, **accessible depuis toutes les fonctions de tous les fichiers source du projet**
* Une variable globale avec le mot-clé static devant **est globale uniquement dans le fichier où elle se trouve**, elle n'est pas accessible depuis les fonctions des autres fichiers.

#### Portée des fonctions

* **Une fonction est par défaut accessible depuis tous les fichiers du projet**, on peut donc l'appeler depuis n'importe quel autre fichier.
* Si on veut qu'une fonction ne soit **accessible que dans le fichier où elle se trouve**, il faut rajouter le mot-clé static devant.

A l'assaut des pointeurs

**Un problème bien ennuyeux**

Un des plus gros problèmes avec les pointeurs, en plus d'être assez difficiles à assimiler pour un débutant, c'est qu'on a du mal à comprendre à quoi ça peut bien servir.  
Alors bien sûr, je pourrais vous dire : "Les pointeurs c'est totalement indispensable on s'en sert tout le temps, croyez-moi c'est super utile !", mais je vois de là vos mines sceptiques :p  
  
Alors, afin d'éviter cela, je vais vous poser un problème que vous ne pourrez pas résoudre sans utiliser de pointeurs. Ce sera en quelque sorte le fil rouge du chapitre. Nous en reparlerons à la fin de ce chapitre et verrons quelle est la solution en utilisant ce que vous aurez appris.  
  
Voici le problème : je veux écrire une fonction qui renvoie 2 valeurs.  
"Impossible" me direz-vous !  
  
En effet, on ne peut renvoyer qu'une valeur par fonction :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | long fonction()  {  **return** machin;  } |

Si on indique long, on renverra un nombre de type long (grâce à l'instruction return).  
  
On peut aussi écrire une fonction qui ne renvoie aucune valeur avec le mot-clé void :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | void fonction()  {  } |

Mais renvoyer 2 valeurs à la fois... c'est pas possible. On ne peut pas faire 2 "return" ni rien.  
  
Alors supposons que je veuille écrire une fonction à laquelle on envoie un nombre de minutes, et celle-ci renverrait le nombre d'heures et minutes correspondantes :  
  
Si on envoie 45, la fonction renvoie 0 heures et 45 minutes.  
Si on envoie 60, la fonction renvoie 1 heure et 0 minutes.  
Si on envoie 90, la fonction renvoie 1 heure et 30 minutes.  
  
Allez, soyons fous, tentons le coup :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | #include <stdio.h>  #include <stdlib.h>  */\* Je mets le prototype en haut. Comme c'est un tout*  *petit programme je ne le mets pas dans un .h, mais*  *en temps normal (dans un vrai programme) j'aurais placé*  *le prototype dans un fichier .h bien entendu ;o) \*/*  void decoupeMinutes(long heures, long minutes);  int main(int argc, char \*argv[])  {  long heures = 0, minutes = 90;  */\* On a une variable minutes qui vaut 90.*  *Après appel de la fonction, je veux que ma variable*  *"heures" vaille 1 et que ma variable "minutes" vaille 30 \*/*  decoupeMinutes(heures, minutes);  printf("%ld heures et %ld minutes", heures, minutes);  **return** 0;  }  void decoupeMinutes(long heures, long minutes)  {  heures = minutes / 60; *// 90 / 60 = 1*  minutes = minutes % 60; *// 90 % 60 = 30 (rappelez-vous : modulo = reste de la division, "90 divisés par 60 font 1, et il reste 30")*  } |

Résultat :  
  
**Code : Console**

|  |
| --- |
| 0 heures et 90 minutes |

Rhaa, zut zut zut et rezut, ça n'a pas marché. Remarquez, je n'avais guère d'espoir :p  
  
Que s'est-il passé ?  
En fait, quand vous "envoyez" une variable à une fonction, une copie de la variable est réalisée. Ainsi, la variable heures dans la fonction "decoupeMinutes" n'est pas la même que celle de la fonction main ! C'est juste **une copie** !  
  
Votre fonction decoupeMinutes fait son job (d'ailleurs j'ose espérer que vous auriez su l'écrire cette fonction, y'a un bon exemple d'utilisation de la division et du modulo ;) ). A l'intérieur de la fonction decoupeMinutes, la variable heures et la variable minutes valent les bonnes valeurs : 1 et 30.  
Mais ensuite, la fonction s'arrête lorsqu'on arrive à l'accolade fermante. Comme on l'a appris dans les chapitres précédents, toutes les variables créées dans une fonction sont détruites à la fin de cette fonction. Votre copie de heures et votre copie de minutes sont donc supprimées.  
On retourne ensuite à la fonction main, dans laquelle vos variables heures et minutes valent toujours 0 et 90. Echec !

I : notez que, comme une fonction fait une copie des variables qu'on lui envoie, vous n'êtes pas du tout obligés d'appeler vos variables de la même façon que dans le main. Ainsi, vous pourriez très bien écrire :  
**Code : C**

|  |  |
| --- | --- |
| 1 | void decoupeMinutes(long h, long m) |

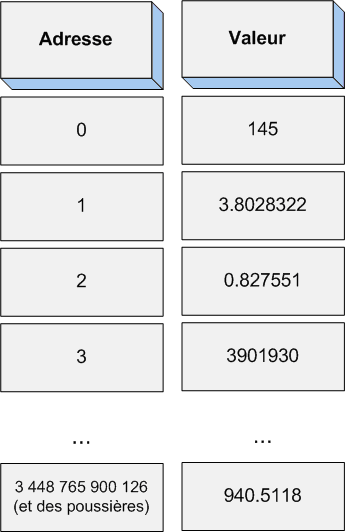
h pour heures et m pour minutes.  
Si vos variables ne s'appellent pas de la même façon que dans le main, ça ne pose donc aucun problème !

Bref, vous aurez beau retourner le problème dans tous les sens... Vous pouvez essayer de renvoyer une valeur avec la fonction (en utilisant un return et en mettant le type long à la fonction), mais vous n'arriveriez à renvoyer qu'une des 2 valeurs. Vous ne pouvez pas renvoyer les 2 valeurs à la fois.  
  
Voilà le problème est posé ;)  
Ce n'est qu'un exemple parmi tant d'autres qui va vous montrer l'utilité des pointeurs. J'ai choisi celui-ci parce qu'il me paraissait intéressant.  
  
Allez, maintenant on peut attaquer le chapitre !

**La mémoire, une question d'adresse**

**Rappel des faits**

Petit flash-back.  
Vous souvenez-vous du chapitre sur les variables ?  
  
Quelle que soit la réponse, je vous recommande **très vivement** d'aller relire la première partie de ce chapitre, intitulée "Une affaire de mémoire". Bien entendu, je ne peux pas vous obliger à le faire, mais ne venez pas pleurnicher ensuite en me disant que vous ne comprenez rien ;)  
  
Il y avait un schéma très important dans ce chapitre, je vous le ressors pour l'occasion. C'était le schéma de la mémoire :



C'est un peu comme ça qu'on peut représenter la mémoire vive (RAM) de votre ordinateur.  
Il faut lire ce schéma ligne par ligne.  
  
La première ligne représente la "cellule" du tout début de la mémoire vive. Chaque cellule a un numéro, c'est **son adresse** (hyper important le vocabulaire là !). La mémoire comporte un grand nombre d'adresses, commençant à l'adresse numéro 0 et se terminant à l'adresse numéro *(insérez un très grand nombre ici)*.  
  
A chaque adresse, on peut stocker un nombre. Un et UN SEUL nombre.  
  
On ne peut donc pas stocker 2 nombres par adresse.  
  
Votre mémoire n'est faite que pour stocker des nombres. Elle ne peut pas stocker de lettres, de phrases. Pour contourner ce problème, on a inventé une table qui fait la liaison entre les nombres et les lettres. Cette table dit par exemple : "*le nombre 89 représente la lettre Y*".  
Mais bon, la gestion de texte en C n'est pas encore pour tout de suite. Nous en parlerons dans quelques chapitres. Avant de pouvoir comprendre ça, il faut d'abord comprendre ce que sont les pointeurs.

**Adresse et valeur**

Revenons-y justement car c'est le sujet.  
Quand vous créez une variable "age" de type long par exemple, en tapant ça :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | long age = 10; |

... votre programme demande au système d'exploitation (Windows par exemple) la permission d'utiliser un peu de mémoire. Le système d'exploitation répond en indiquant à quelle adresse en mémoire il vous laisse le droit d'inscrire votre nombre.  
C'est d'ailleurs justement un des rôles principaux d'un système d'exploitation : **on dit qu'il alloue de la mémoire aux programmes**. C'est un peu lui le chef, il contrôle chaque programme et vérifie qu'il se sert de la mémoire à l'endroit qu'il lui a autorisé.

C'est d'ailleurs là la cause n°1 des plantages de programmes : si votre programme essaie d'accéder à une zone de la mémoire qui ne lui appartient pas, le système d'exploitation (abrégez "OS") refuse cela et coupe brutalement le programme en guise de punition (*"C'est qui le chef ici ?"*).  
L'utilisateur, lui, voit une jolie boîte de dialogue *"Ce programme va être arrêté parce qu'il a effectué une opération non conforme"* (quand c'est pas trop trop grave), ou, pire : un terrible écran-bleu-de-la-mort o_O  
Mais généralement, si l'OS est bien codé votre ordinateur ne devrait pas complètement se bloquer à cause d'un simple "dépassement de mémoire". Enfin, moi j'dis ça, j'dis rien :p

Je m'égare. Où en étions-nous déjà ?  
Ah oui, notre variable age. La valeur 10 a été inscrite quelque part en mémoire, disons par exemple à l'adresse n°4655.  
Ce qu'il se passe (et c'est le rôle du compilateur) c'est que le mot "age" dans votre programme est remplacé par l'adresse 4655 à l'exécution. Cela fait que, à chaque fois que vous avez tapé le mot age dans votre code source, cela est remplacé par 4655, et votre ordinateur voit ainsi à quelle adresse il doit aller chercher en mémoire ! Du coup, l'ordinateur se rend en mémoire à l'adresse 4655 et répond fièrement : "Ca vaut 10 !".  
  
On sait donc comment récupérer la valeur de la variable : il suffit tout bêtement de taper "age" dans son code source. Si on veut afficher l'âge, on peut utiliser la fonction*printf*:  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | printf("La variable age vaut : %ld", age); |

Résultat à l'écran :  
  
**Code : Console**

|  |
| --- |
| La variable age vaut : 10 |

Rien de bien nouveau jusque-là.

**Le scoop du jour**

On sait afficher la valeur de la variable, mais saviez-vous que l'on peut aussi afficher l'adresse correspondante ? :D  
... Ah oui non c'est vrai vous ne saviez pas ;)  
  
Pour afficher l'adresse de la variable, on doit utiliser le symbole %p (le p du mot "pointeur") dans le printf. En outre, on doit envoyer à la fonction printf non pas la variable age, mais son adresse... Et pour faire cela, vous devez mettre le symbole & devant la variable age, comme je vous avais demandé de le faire pour les scanf il y a quelques temps sans vous expliquer pourquoi ;)  
  
Tapez donc :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | printf("L'adresse de la variable age est : %p", &age); |

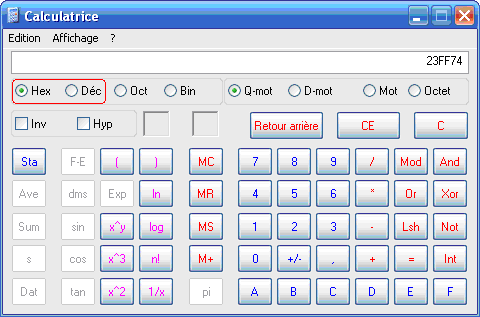
Résultat :  
  
**Code : Console**

|  |
| --- |
| L'adresse de la variable age est : 0023FF74 |

Ce que vous voyez là est l'adresse de la variable age au moment où j'ai lancé le programme sur mon ordinateur. Oui oui, c'est un nombre.  
0023FF74 est un nombre, il est simplement écrit dans le système hexadécimal, au lieu du système décimal auquel nous avons l'habitude.

Si vous remplacez le %p par un %ld, vous devriez obtenir le nombre en système décimal (plus compréhensible pour nous pauvres humains). Toutefois, le %p a été fait spécialement pour afficher des adresses, donc je préfère en général l'utiliser à la place de %ld.

Sans rentrer dans les détails, juste pour que vous soyez pas trop perturbés, sachez que le fameux système décimal représente tous les nombres avec 10 chiffres : 0 1 2 3 4 5 6 7 8 9  
En hexadécimal (un mode dans lequel l'ordinateur travaille souvent), les nombres sont représentés avec 16 chiffres : 0 1 2 3 4 5 6 7 8 9 A B C D E F (les lettres sont en fait des chiffres supplémentaires pour représenter les nombres).  
  
Tout nombre en hexadécimal peut se convertir en décimal et inversement. Ainsi, A vaut 10, B vaut 11, C vaut 12... F vaut 15, 10 vaut 16, 11 vaut 17, 12 vaut 18 et ainsi de suite.  
Si vous avez une calculatrice (au hasard la calculatrice de Windows en mode scientifique), vous pouvez convertir les nombres.

  
*La calculatrice de Windows peut convertir les hexadécimaux*

Vous devez d'abord vous assurer que vous êtes dans le mode scientifique : Affichage / Scientifique.  
Ensuite, cliquez sur Hex (j'ai entouré en rouge sur ma capture d'écran). Tapez le nombre en hexadécimal que vous avez. Puis, cliquez sur Déc juste à côté pour transformer en décimal. Et voilà le travail ;)  
Ca marche aussi en sens inverse bien sûr :)  
  
Ainsi, j'ai pu voir que 0023FF74 correspondait en fait au nombre 2359156. Bon, on s'en fout un peu, ça ne changera pas notre vie, mais ça fait du bien de savoir comment ça marche non ? ^^

Si vous exécutez ce programme sur votre ordinateur, l'adresse sera très certainement différente. Tout dépend de la place que vous avez en mémoire, des programmes que vous avez lancés etc... Il est totalement impossible de prédire à quelle adresse la variable sera stockée chez vous ;)  
Si vous lancez votre programme plusieurs fois d'affilée, il se peut que l'adresse soit identique, la mémoire n'ayant pas beaucoup changé entre temps. Si vous redémarrez votre ordinateur par contre, vous aurez sûrement une valeur différente.

Où je veux en venir avec tout ça ?  
Eh bien en fait, je veux vous faire retenir la chose suivante toute bête :

* age : affiche **la VALEUR** de la variable.
* &age : affiche **l'ADRESSE** de la variable.

Avec "age", l'ordinateur va lire la valeur de la variable en mémoire et vous renvoie cette valeur. Avec "&age", votre ordinateur vous dit en revanche à quelle adresse se trouve la variable.

**Utiliser des pointeurs**

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.

Mais... Les adresses sont des nombres aussi non ? Ca revient à stocker des nombres encore et toujours !

C'est exact. Mais ces nombres auront une signification particulière : ils indiqueront l'adresse d'une autre variable en mémoire.

**Créer un pointeur**

Pour créer une variable de type pointeur, on doit rajouter le symbole \* devant le nom de la variable.  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | long \*monPointeur; |

Notez qu'on peut aussi écrire...  
**Code : C**

|  |  |
| --- | --- |
| 1 | long\* pointeurSurAge; |

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 (première méthode) :  
**Code : C**

|  |  |
| --- | --- |
| 1 | long \*pointeur1, \*pointeur2, \*pointeur3; |

Comme je vous l'ai appris, il est important d'initialiser dès le début ses variables (en leur donnant la valeur 0 par exemple). C'est encore plus important de le faire avec les pointeurs !  
Pour initialiser un pointeur (lui donner une valeur par défaut), on n'utilise généralement pas le nombre 0 mais le mot-clé NULL (les majuscules sont importantes attention) :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | long \*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.  
  
Pourquoi pas l'adresse de la variable age ? Vous savez maintenant comment indiquer l'adresse d'une variable au lieu de sa valeur (en utilisant le symbole &) alors zou ! Ca nous donne :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2 | long age = 10;  long \*pointeurSurAge = &age; |

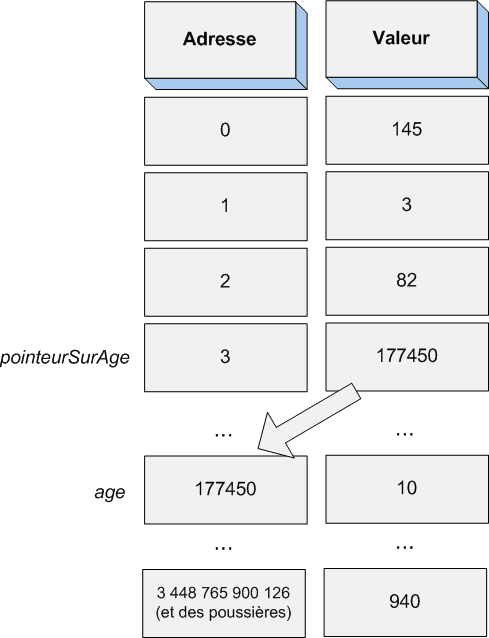
La première ligne signifie : *"Créer une variable de type long dont la valeur vaut 10"*  
La deuxième ligne signifie : *"Créer une variable de type pointeur dont la valeur vaut l'adresse de la variable age"*.  
  
La seconde ligne fait donc 2 choses à la fois. Si vous le souhaitez, pour ne pas tout mélanger, sachez qu'on peut la découper en 2 temps :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3 | long age = 10;  long \*pointeurSurAge; *// 1) Signifie "Je crée un pointeur"*  pointeurSurAge = &age; *// 2) Signifie "pointeurSurAge contient l'adresse de la variable age"* |

Vous avez remarqué qu'il n'y a pas de type "pointeur" comme il y a un type "int", un type "double" ou encore un type "long".  
On n'écrit donc pas :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | pointeur pointeurSurAge; |

Au lieu de ça, on utilise le symbole \*, mais on continue à écrire "long". Qu'est-ce que ça signifie ?  
En fait, (accrochez-vous), on doit indiquer quel est le type de la variable dont le pointeur va contenir l'adresse. Comme notre pointeur pointeurSurAge va contenir l'adresse de la variable age (qui est de type long) alors mon pointeur doit être de type "long\*" ! Si ma variable age avait été de type int, alors j'aurais dû écrire "int \*monPointeur".  
  
**Vocabulaire**: on dit que le pointeur pointeurSurAge pointe sur la variable age.  
  
Un petit schéma de ce qu'il se passe en mémoire :



Dans ce schéma, la variable age a été placée à l'adresse 177450 (vous voyez d'ailleurs que sa valeur est 10), et le pointeur pointeurSurAge a été placé à l'adresse 3 (c'est tout à fait le fruit du hasard hein, j'invente :p ).  
  
Lorsque mon pointeur est créé, le système d'exploitation réserve une case en mémoire comme il l'a fait pour age. La différence ici, c'est que la valeur de pointeurSurAge est un peu particulière. Regardez bien le schéma : c'est l'adresse de la variable age !  
  
Ceci, mesdames et messieurs, est le secret absolu de tout programme écrit en langage C (et donc aussi en langage C++ ;) ). On y est, nous venons de rentrer dans le monde merveilleux des pointeurs !

Ouah, super. Et ça fait quoi ton truc ?

Ca ne transforme pas encore votre ordinateur en machine à café, certes.  
  
Seulement maintenant, on a un pointeurSurAge qui contient l'adresse de la variable age.  
Essayons de voir ce que contient le pointeur en faisant un printf dessus :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | long age = 10;  long \*pointeurSurAge = &age;  printf("%ld", pointeurSurAge); |

**Code : Console**

|  |
| --- |
| 177450 |

Hum. En fait, cela n'est pas très étonnant. On demande la valeur de pointeurSurAge, et sa valeur c'est l'adresse de la variable age (177450).  
Comment faire pour demander à avoir la valeur de la variable se trouvant à l'adresse indiquée dans pointeurSurAge ? Il faut mettre le symbole \* devant le nom du pointeur :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | long age = 10;  long \*pointeurSurAge = &age;  printf("%ld", \*pointeurSurAge); |

**Code : Console**

|  |
| --- |
| 10 |

Hourra ! :D  
On y est arrivés ! En mettant le symbole \* devant le nom du pointeur, on accède à la valeur de la variable age :)  
  
Si au contraire on avait mis le symbole & devant le nom du pointeur, on aurait eu l'adresse où se trouve le pointeur (ici, c'est 3)

Je ne vois pas ce qu'on y gagne. Après tout, sans pointeur on peut très bien afficher la valeur de la variable age !

Cette question (que vous devez inévitablement vous poser) me fait sourire pour 2 raisons :

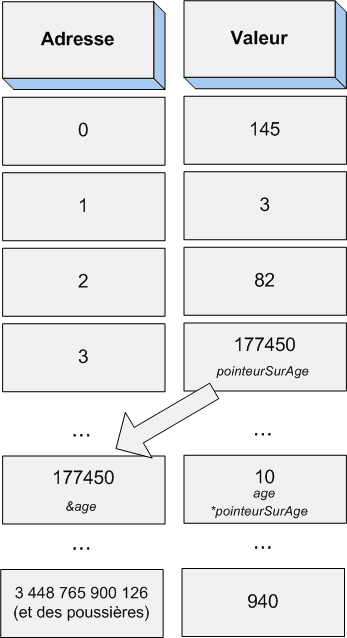
* La première, c'est que j'étais sûr que vous me diriez ça. Et après tout, qui pourrait vous en vouloir ? Actuellement l'intérêt n'est pas évident, mais petit à petit, au fur et à mesure des chapitres suivants, vous comprendrez que tout ce tintouin n'a pas été inventé par pur plaisir de compliquer les choses :D
* La seconde, elle est toute bête : c'est que je me souviens exactement que j'étais comme vous à ce moment-là lorsque j'apprenais le C ^^ Bref, je comprends exactement la frustration que vous devez ressentir. Mais c'est une bien maigre consolation pour vous j'en conviens :p

**A retenir absolument**

Avant d'aller plus loin, s'il y avait une chose à retenir pour le moment ce serait cela en 4 points :

* Sur une variable, comme la variable age :
  + "age" signifie : "Je veux la valeur de la variable age".
  + "&age" signifie : "Je veux l'adresse où se trouve la variable age".
* Sur un pointeur, comme pointeurSurAge :
  + "pointeurSurAge" signifie : "Je veux la valeur de pointeurSurAge" (cette valeur étant une adresse).
  + "\*pointeurSurAge" signifie : "Je veux la valeur de la variable qui se trouve à l'adresse contenue dans pointeurSurAge"

Contentez-vous de bien retenir ces 4 points. Faites des tests et vérifiez que ça marche.  
Voici un schéma qui va vous permettre de bien situer qui désigne quoi :



Attention à 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 :  
**Code : C**

|  |  |
| --- | --- |
| 1 | long \*pointeurSurAge; |

En revanche, lorsque vous utilisez votre pointeur ensuite en écrivant :  
**Code : C**

|  |  |
| --- | --- |
| 1 | printf("%ld", \*pointeurSurAge); |

... cela ne signifie pas "Je veux créer un pointeur" mais : "Je veux la valeur de la variable sur laquelle pointe mon pointeurSurAge".

Tout cela est fon-da-men-tal. Il faut savoir cela par coeur, et surtout le comprendre. Même pas la peine de continuer ce chapitre si vous n'avez pas compris cela, je préfère être franc ;)  
  
N'hésitez pas à lire et relire ce qu'on vient d'apprendre. Je ne peux pas vous en vouloir si vous n'avez pas compris du premier coup, et ce n'est pas une honte non plus d'ailleurs.  
Pour info, avant que j'arrive à comprendre cela il a dû se passer une petite semaine (bon pas à temps plein je l'avoue :p ).  
Et pour comprendre la plupart des subtilités des pointeurs, je crois qu'il m'a fallu bien 2 ou 3 mois au moins (pas à temps plein là non plus, rassurez-vous ;) )  
  
Bref, si vous vous sentez un peu perdus, pensez à ces gens qui sont aujourd'hui des grands gourous de la programmation : aucun d'entre eux n'a compris tout le fonctionnement des pointeurs du premier coup.  
Et si jamais cette personne existe, croyez-moi j'aimerais la rencontrer :D

**Envoyer un pointeur à une fonction**

Le gros intérêt des pointeurs (mais ce n'est pas le seul), c'est de 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 :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | void triplePointeur(long \*pointeurSurNombre);  int main(int argc, char \*argv[])  {  long nombre = 5;  triplePointeur(&nombre); *// On envoie l'adresse de nombre à la fonction*  printf("%ld", nombre); *// On affiche la variable nombre. La fonction a directement modifié la valeur de la variable car elle connaissait son adresse*  **return** 0;  }  void triplePointeur(long \*pointeurSurNombre)  {  \*pointeurSurNombre \*= 3; *// On multiplie par 3 la valeur de la variable nombre*  } |

**Code : Console**

|  |
| --- |
| 15 |

La fonction triplePointeur prend un paramètre de type long\* (c'est-à-dire un pointeur sur long). 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. Ca, 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. A 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"). On n'est plus limités à une seule valeur !

Quel est l'intérêt du coup d'utiliser un return dans une fonction si on peut se servir des pointeurs pour modifier des valeurs ?

Ca dépendra de vous et de votre programme. C'est à vous de décider. Il faut savoir que les return sont bel et bien toujours utilisés en C. Le plus souvent, on s'en sert pour renvoyer ce qu'on appelle *un code d'erreur* : la fonction renvoie 1 (vrai) si tout s'est bien passé, et 0 (faux) s'il y a eu une erreur pendant le déroulement de la fonction.  
Mais bon, on aura le temps de voir comment gérer les erreurs en C plus tard ;)

**Une autre façon d'envoyer un pointeur à une fonction**

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 tripleNombre (de type long\*).  
  
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 :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | void triplePointeur(long \*pointeurSurNombre);  int main(int argc, char \*argv[])  {  long nombre = 5;  long \*pointeur = &nombre; *// pointeur prend l'adresse de nombre*  triplePointeur(pointeur); *// On envoie pointeur (l'adresse de nombre) à la fonction*  printf("%ld", \*pointeur); *// On affiche la valeur de nombre, en tapant \*pointeur*  **return** 0;  }  void triplePointeur(long \*pointeurSurNombre)  {  \*pointeurSurNombre \*= 3; *// On multiplie par 3 la valeur de la variable nombre*  } |

Pour que vous ayez les 2 codes sources côte à côte, je vous mets celui de tout à l'heure ci-dessous. Comparez-les bien, il y a de subtiles différences et vous devez arriver à comprendre pourquoi il y a ces différences entre ces 2 codes sources.  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | void triplePointeur(long \*pointeurSurNombre);  int main(int argc, char \*argv[])  {  long nombre = 5;  triplePointeur(&nombre); *// On envoie l'adresse de nombre à la fonction*  printf("%ld", nombre); *// On affiche la variable nombre. La fonction a directement modifié la valeur de la variable car elle connaissait son adresse*  **return** 0;  }  void triplePointeur(long \*pointeurSurNombre)  {  \*pointeurSurNombre \*= 3; *// On multiplie par 3 la valeur de la variable nombre*  } |

Et le résultat dans les deux cas est le même :  
  
**Code : Console**

|  |
| --- |
| 15 |

Ce qui compte, c'est d'envoyer l'adresse de la variable nombre à la fonction. Or, pointeur vaut l'adresse de la variable nombre, donc c'est bon de ce côté ! On le fait juste d'une manière différente en créant un pointeur dans la fonction main.  
Dans le printf (et c'est juste pour l'exercice), j'affiche le contenu de la variable nombre en tapant \*pointeur. Notez que j'aurais pu à la place taper "nombre" : ça aurait été pareil car cela désigne la même chose dans la mémoire.  
  
J'ai mis des semaines avant de comprendre que ces 2 codes faisaient effectivement la même chose, mais d'une manière différente. Si vous arrivez à comprendre ça, alors bravo, respect, bien joué, vous avez compris tout ce que je voulais vous enseigner sur les pointeurs :)

Comme je vous le disais tout à l'heure, dans le programme "Plus ou Moins" nous avons utilisé des pointeurs sans vraiment savoir. C'était en fait en appelant la fonction scanf. En effet, cette fonction a pour rôle de lire ce que l'utilisateur a rentré au clavier et de renvoyer cela.  
Pour que la fonction puisse modifier *directement*le contenu de votre variable afin d'y mettre la valeur tapée au clavier, elle a besoin de l'adresse de la variable :  
**Code : C**

|  |  |
| --- | --- |
| 1  2 | long nombre = 0;  scanf("%ld", &nombre); |

La fonction travaille avec un pointeur sur la variable nombre, et peut ainsi modifier directement le contenu de nombre.  
Comme on vient de le voir, on pourrait créer un pointeur qu'on enverrait à la fonction scanf :  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3 | long nombre = 0;  long \*pointeur = &nombre;  scanf("%ld", pointeur); |

Attention à ne pas mettre le symbole & devant pointeur dans la fonction scanf ! Ici, pointeur contient lui-même l'adresse de la variable nombre, pas besoin de mettre un & ! Si vous faisiez ça, vous enverriez l'adresse où se trouve le pointeur, et ça, excusez mon langage, mais on s'en fout complètement :D

**Qui a dit : "Un problème bien ennuyeux" ?**

Le chapitre est sur le point de s'achever, il est temps de retrouver notre fil rouge :)  
Si vous avez compris ce chapitre, vous devriez être capables de résoudre le problème maintenant.  
  
...  
  
Quoi qu'est-ce que vous attendez ? Allez au boulot tas d'feignasses ! :p  
  
...  
...  
  
Vous voulez la solution pour comparer ? La voici ! :D  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | void decoupeMinutes(long\* pointeurHeures, long\* pointeurMinutes);  int main(int argc, char \*argv[])  {  long heures = 0, minutes = 90;  *// On envoie l'adresse de heures et minutes*  decoupeMinutes(&heures, &minutes);  *// Cette fois, les valeurs ont été modifiées !*  printf("%ld heures et %ld minutes", heures, minutes);  **return** 0;  }  void decoupeMinutes(long\* pointeurHeures, long\* pointeurMinutes)  {  */\* Attention à ne pas oublier de mettre une étoile devant le nom*  *des pointeurs ! Comme ça, vous pouvez modifier la valeur des variables,*  *et non leur adresse ! Vous ne voudriez pas diviser des adresses*  *n'est-ce pas ? ;o) \*/*  \*pointeurHeures = \*pointeurMinutes / 60;  \*pointeurMinutes = \*pointeurMinutes % 60;  } |

Résultat :  
  
**Code : Console**

|  |
| --- |
| 1 heures et 30 minutes |

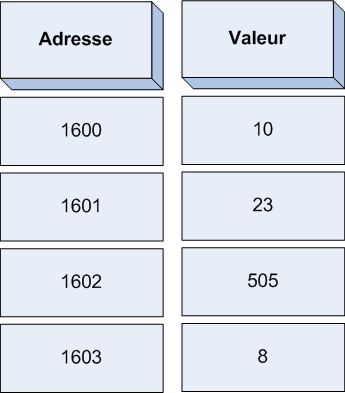
Alors, c'est qui le plus fort ? :D  
  
Est-ce que j'ai besoin de vous expliquer encore une fois comment ça marche ? En théorie, mes explications précédentes devraient suffire, je ne peux rien vous apprendre de nouveau.  
Mais bon allez, pour la forme, et parce que c'est un chapitre important, je vais me répéter encore une fois. Il paraît qu'en rabâchant les mêmes choses ça finit par rentrer, si c'est le cas tant mieux pour vous ;)  
  
*Explications :*

1. Les variables heures et minutes sont créées dans le main.
2. On envoie à la fonction decoupeMinutes l'adresse de heures et minutes.
3. La fonction decoupeMinutes récupère ces adresses dans des pointeurs appelés pointeurHeures et pointeurMinutes. Notez que, là encore, le nom importe peu. J'aurais pu les appeler h et m, ou même encore heures et minutes (mais je ne veux pas que vous risquiez de confondre avec les variables heures et minutes du main, qui ne sont pas les mêmes ;) )
4. La fonction decoupeMinutes modifie directement les valeurs des variables heures et minutes en mémoire car elle possède leurs adresses dans des pointeurs. La seule contrainte, un peu gênante je dois le reconnaître, c'est qu'il faut impérativement mettre une étoile devant le nom des pointeurs si on veut modifier la valeur de heures et minutes. Si on n'avait pas fait ça, on aurait modifié l'adresse contenue dans les pointeurs, ce qui aurait servi... à rien :p

De nombreux lecteurs m'ont fait remarquer qu'il était possible de résoudre le "problème" sans utiliser de pointeurs. Oui, bien sûr que je sais que c'est possible, mais il faut contourner les règles que nous nous sommes fixées : on peut utiliser des variables globales (bêrk), ou encore faire un printf dans la fonction (alors que c'est dans le main qu'on veut faire le printf !)  
Bref, si vous aussi vous trouvez un moyen de résoudre le problème *autrement*, vous emballez pas. Ce n'était qu'un exemple un peu "théorique" pour vous montrer l'intérêt des pointeurs. Dans les prochains chapitres cet intérêt vous paraîtra de plus en plus évident ;)

**Les tableaux dans la mémoire**

*"Les tableaux sont une suite de variables de même type, situées dans un espace contigu en mémoire"*  
  
Bon, je reconnais que ça fait un peu définition de dictionnaire tout ça ^^  
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 cases, 3 cases, 10 cases, 150 cases, 2500 cases, c'est vous qui décidez.  
Ci-dessous, voici un schéma d'un tableau de 4 cases en mémoire qui commence à l'adresse 1600 :



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 long, alors chaque case du tableau contiendra un long. On ne peut pas faire de tableau contenant à la fois des long 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.

**Définir un tableau**

Pour commencer, nous allons voir comment définir un tableau de 4 long :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | long tableau[4]; |

Voilà c'est tout :)  
Il suffit donc de rajouter entre crochets le nombre de cases que vous voulez mettre dans votre tableau. Il n'y a pas de limite (à part peut-être la taille de votre mémoire :D ).  
  
Maintenant, comment accéder à chaque case du tableau ?  
Il faut écrire tableau[numeroDeLaCase].

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

Si je veux mettre dans mon tableau les mêmes valeurs que celles indiquées dans mon schéma, je devrai donc faire :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | long tableau[4];  tableau[0] = 10;  tableau[1] = 23;  tableau[2] = 505;  tableau[3] = 8; |

Tu as dit qu'il y avait des pointeurs avec les tableaux. Où ça, je n'en vois pas ?

En fait, si vous écrivez juste "tableau", vous avez un pointeur. C'est un pointeur sur la première case du tableau.  
Faites le test :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3 | long tableau[4];  printf("%ld", tableau); |

Résultat, on voit l'adresse où se trouve tableau :  
  
**Code : Console**

|  |
| --- |
| 1600 |

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

|  |  |
| --- | --- |
| 1  2  3 | long tableau[4];  printf("%ld", tableau[0]); |

**Code : Console**

|  |
| --- |
| 10 |

De même pour les autres indices.  
  
  
Notez que, comme tableau est un pointeur, on peut utiliser le symbole \* pour connaître la première valeur :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3 | long tableau[4];  printf("%ld", \*tableau); |

**Code : Console**

|  |
| --- |
| 10 |

Il est aussi possible d'avoir la valeur de la seconde case en tapant \*(tableau + 1) (adresse de tableau + 1).  
Les 2 lignes suivantes sont donc identiques :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2 | tableau[1] *// Renvoie la valeur contenue dans la seconde case (la première case étant 0)*  \*(tableau + 1) *// Identique : renvoie la valeur contenue dans la seconde case* |

Donc, quand vous écrivez tableau[0], vous demandez la valeur qui se trouve à l'adresse tableau + 0 cases (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 :)

**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 :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2 | long taille = 5;  long tableau[taille]; |

Or, cela n'est pas forcément reconnu par tous les compilateurs, certains planteront sur la ligne n°2.  
Le langage C que je vous enseigne depuis le début (appelé le C89) n'autorise pas ce genre de choses. Nous considèrerons donc que faire cela est interdit.  
  
Nous allons nous mettre d'accord sur la chose suivante : vous n'avez pas le droit de mettre une variable entre crochets pour la définition de la taille du tableau, même si cette variable est une constante ! const long taille = 5; ne marchera donc pas mieux.  
Le tableau doit avoir une dimension fixe, c'est-à-dire que vous devez écrire **noir sur blanc** le nombre correspondant à la taille :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | long tableau[5]; |

Mais... Alors il est interdit de créer un tableau en fonction de la taille d'une variable ?

Non rassurez-vous, c'est possible ! (même en C89 :p )  
Mais pour faire cela nous utiliserons une autre technique (plus sûre et qui marche partout) appelée **l'allocation dynamique**. Nous verrons cela bien plus tard dans la partie II de ce cours.

**Parcourir un tableau**

Supposons que je veuille maintenant afficher les valeurs de chaque case du tableau.  
Je pourrais faire autant de *printf*qu'il n'y a de cases. Mais bon, c'est répétitif, un peu lourd, et imaginez le bazar si le tableau contenait 8000 nombres :p  
  
Le mieux c'est de se servir d'une boucle. Pourquoi pas d'une boucle for ? Les boucles for sont très pratiques pour parcourir un tableau :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | int main(int argc, char \*argv[])  {  long tableau[4], i = 0;  tableau[0] = 10;  tableau[1] = 23;  tableau[2] = 505;  tableau[3] = 8;  **for** (i = 0 ; i < 4 ; i++)  {  printf("%ld**\n**", tableau[i]);  }  **return** 0;  } |

**Code : Console**

|  |
| --- |
| 10  23  505  8 |

Notre boucle parcourt le tableau à l'aide d'une variable appelée i (c'est le nom super original que les programmeurs donnent en général à la variable qui leur permet de parcourir le tableau :p )  
  
Ce qui est particulièrement pratique, c'est qu'on peut mettre une variable entre crochets. En effet, la variable était interdite pour la création du tableau (pour définir sa taille), mais elle est heureusement autorisée pour "parcourir" le tableau, c'est-à-dire afficher ses valeurs !  
Ici, on a mis la variable i, qui vaut successivement 0, 1, 2, 3. Ainsi, on va donc afficher la valeur de tableau[0], tableau[1], tableau[2] et tableau[3] ! :)

Attention à ne pas tenter 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.

Voilà la technique ^^  
Ce n'est pas bien bien compliqué vous voyez.

**Initialiser un tableau**

Maintenant que l'on sait parcourir un tableau, on est capables d'initialiser toutes ses valeurs à 0 en faisant une boucle !  
  
Bon, parcourir le tableau pour mettre 0 à chaque case, c'est pas trop dur vous devriez arriver à le faire :)  
Voici le code :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | int main(int argc, char \*argv[])  {  long tableau[4], i = 0;  *// Initialisation du tableau*  **for** (i = 0 ; i < 4 ; i++)  {  tableau[i] = 0;  }  *// Affichage de ses valeurs pour vérifier*  **for** (i = 0 ; i < 4 ; i++)  {  printf("%ld**\n**", tableau[i]);  }  **return** 0;  } |

**Code : Console**

|  |
| --- |
| 0  0  0  0 |

**Une autre façon d'initialiser**

Il faut savoir qu'il existe une autre façon d'initialiser un tableau un peu plus automatisée en C.  
Elle consiste à écrire tableau[4] = {valeur1, valeur2, valeur3, valeur4}  
En clair, vous mettez les valeurs une à une entre accolades, séparées par des virgules :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | int main(int argc, char \*argv[])  {  long tableau[4] = {0, 0, 0, 0}, i = 0;  **for** (i = 0 ; i < 4 ; i++)  {  printf("%ld**\n**", tableau[i]);  }  **return** 0;  } |

**Code : Console**

|  |
| --- |
| 0  0  0  0 |

Mais en fait, c'est même mieux que ça : vous pouvez définir les valeurs des premières cases du tableau, toutes celles que vous n'aurez pas renseignées seront automatiquement mises à 0.  
  
Ainsi, si je fais :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | long tableau[4] = {10, 23}; *// Valeurs insérées : 10, 23, 0, 0* |

La case n°0 prendra la valeur 10, la n°1 prendra 23, et toutes les autres prendront la valeur 0 (par défaut).  
  
Comment initialiser tout le tableau à 0 en sachant ça ?  
Eh bien, il vous suffit d'initialiser au moins la première valeur à 0, et toutes les autres valeurs non indiquées prendront la valeur 0 :)  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | long tableau[4] = {0}; *// Toutes les cases du tableau seront initialisées à 0* |

Cette technique a l'avantage de fonctionner avec un tableau de n'importe quelle taille (là ça marche pour 4 cases, mais s'il en avait eu 100 ça aurait été bon aussi ;) )

Attention, contrairement à ce que beaucoup d'entre vous semblent croire :  
**Code : C**

|  |  |
| --- | --- |
| 1 | long tableau[4] = {1}; *// Valeurs insérées : 1, 0, 0, 0* |

On n'initialise pas toutes les cases à 1 en faisant cela : seule la première case sera à 1, les autres seront à 0. On ne peut donc pas initialiser toutes les cases à 1 automatiquement, à moins de faire une boucle.

**Passage de tableaux à une fonction**

Vous aurez sûrement souvent besoin d'afficher tout le contenu de votre tableau.  
Pourquoi ne pas écrire une fonction qui fait ça ? Ca va nous permettre de voir comment on envoie un tableau à une fonction en plus, donc ça m'arrange :-°  
  
Il va falloir envoyer 2 informations à la fonction : le tableau (enfin, 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 :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | *// Prototype de la fonction d'affichage*  void affiche(long \*tableau, long tailleTableau);    int main(int argc, char \*argv[])  {  long tableau[4] = {10, 15, 3};    *// On affiche le contenu du tableau*  affiche(tableau, 4);    **return** 0;  }    void affiche(long \*tableau, long tailleTableau)  {  long i;    **for** (i = 0 ; i < tailleTableau ; i++)  {  printf("%ld**\n**", tableau[i]);  }  } |

**Code : Console**

|  |
| --- |
| 10  15  3  0 |

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 long (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.  
  
**Important**: il existe une autre façon d'indiquer que la fonction reçoit un tableau. Plutôt que d'indiquer que la fonction attend un long \*tableau, mettez ceci :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | void affiche(long tableau[], long 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. Ca 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.

**Quelques exercices !**

J'ai plein d'idées d'exercices pour vous entraîner ! :D  
Je vous propose de réaliser des fonctions travaillant sur des tableaux.  
  
Je donne juste les énoncés des exercices ici pour vous forcer à réfléchir à vos fonctions. Si vous avez du mal à réaliser ces fonctions, rendez-vous sur les forums pour poser vos questions ;)

* **Exercice 1** : créer une fonction *sommeTableau* qui renvoie la somme des valeurs contenues dans le tableau (utilisez un return pour renvoyer la valeur).  
  Pour vous aider, voici le prototype de la fonction à créer :  
    
  **Code : C**

|  |  |
| --- | --- |
| 1 | long sommeTableau(long tableau[], long tailleTableau); |

* **Exercice 2** : créer une fonction *moyenneTableau*qui calcule et renvoie la moyenne des valeurs.  
  Prototype :  
    
  **Code : C**

|  |  |
| --- | --- |
| 1 | double moyenneTableau(long tableau[], long tailleTableau); |

* La fonction renvoie un double car une moyenne est parfois un nombre décimal (souvent même :p )
* **Exercice 3** : créer une fonction *copierTableau* qui prend en paramètre 2 tableaux. Le contenu du premier tableau devra être copié dans le second tableau.  
  Prototype :  
    
  **Code : C**

|  |  |
| --- | --- |
| 1 | void copie(long tableauOriginal[], long tableauCopie[], long tailleTableau); |

* **Exercice 4** : créer une fonction *maximumTableau* qui aura pour rôle de remettre à 0 toutes les cases du tableau ayant une valeur supérieure à un maximum. Cette fonction prendra en paramètre le tableau ainsi que le nombre maximum autorisé (*valeurMax*). Toutes les cases qui contiennent un nombre supérieur à *valeurMax*doivent être mises à 0.  
  Prototype :  
    
  **Code : C**

|  |  |
| --- | --- |
| 1 | void maximumTableau(long tableau[], long tailleTableau, long valeurMax); |

* **Exercice 5** (plus difficile) : créer une fonction *ordonnerTableau* qui classe les valeurs d'un tableau dans l'ordre croissant. Ainsi, un tableau qui vaut {15, 81, 22, 13} doit à la fin de la fonction valoir {13, 15, 22, 81} !  
  Cet exercice est un peu plus difficile que les autres, mais est tout à fait réalisable. Ca va vous occuper un petit moment :p  
    
  Prototype :  
    
  **Code : C**

|  |  |
| --- | --- |
| 1 | void ordonnerTableau(long tableau[], long tailleTableau); |

Faites-vous un petit fichier de fonctions appelé tableaux.c (avec son homologue tableaux.h qui contiendra les prototypes bien sûr !) contenant toutes les fonctions de votre cru réalisant des opérations sur des tableaux :)  
Vous entraîner comme ça, c'est le meilleur moyen de vous former :)

**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 long ou un int. Certes, ça prend un peu plus de place en mémoire, mais aujourd'hui la mémoire c'est vraiment pas ce qui manque sur un ordinateur ;) Vous ne tuerez pas votre ordi parce que vous utilisez des int ou des long :p

**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 de mettre cette lettre entre apostrophes, comme ceci : 'A'. A la compilation, 'A' sera remplacé par la valeur correspondante.  
  
Testons :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int main(int argc, char \*argv[])  {  char lettre = 'A';  printf("%ld**\n**", lettre);  **return** 0;  } |

**Code : Console**

|  |
| --- |
| 65 |

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.  
  
La plupart des caractères "de base" sont codés entre les nombres 0 et 127.

**Afficher un caractère**

La fonction printf, qui n'a décidemment pas fini de nous étonner, peut aussi afficher un caractère. Pour cela, on doit utiliser le symbole %c (c comme caractère) :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int main(int argc, char \*argv[])  {  char lettre = 'A';  printf("%c**\n**", lettre);  **return** 0;  } |

**Code : Console**

|  |
| --- |
| A |

Hourra !  
Nous savons afficher une lettre :D  
  
On peut aussi demander à l'utilisateur de rentrer une lettre en utilisant le %c dans un scanf :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | int main(int argc, char \*argv[])  {  char lettre = 0;  scanf("%c", &lettre);  printf("%c**\n**", lettre);  **return** 0;  } |

Si je tape la lettre B, je verrai :  
  
**Code : Console**

|  |
| --- |
| B  B |

*(le premier des 2 B étant celui que j'ai tapé au clavier, et le second étant affiché par le printf)*  
  
Voici à peu près tout ce qu'il faut savoir sur le type char. Il cachait décidemment bien son jeu celui-là ;)  
  
**Retenez bien :**

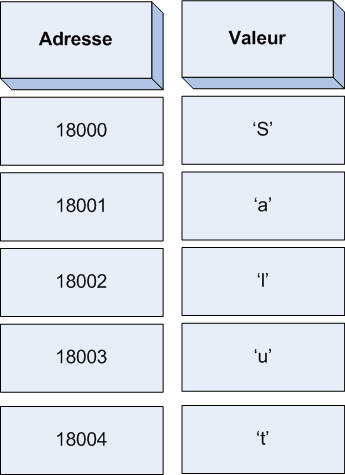
* Le type char permet de stocker des nombres allant de -128 à 127, unsigned char des nombres de 0 à 255.
* Il y a une table que votre ordinateur utilise pour convertir les lettres en nombres et inversement.
* On peut donc utiliser le type char pour stocker UNE lettre.
* 'A' est remplacé à la compilation par la valeur correspondante (65 en l'occurrence). On utilise donc les apostrophes pour obtenir la valeur d'une lettre.

**Les chaînes sont des tableaux de char !**

Arf, j'ai tout dit dans le titre, qu'est-ce que je vais bien pouvoir raconter maintenant :lol:  
Ben oui, tout y est : une chaîne de caractères n'est rien d'autre qu'un tableau de type char. Un bête tableau de rien du tout ;)  
  
Si on crée un tableau :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | char chaine[5]; |

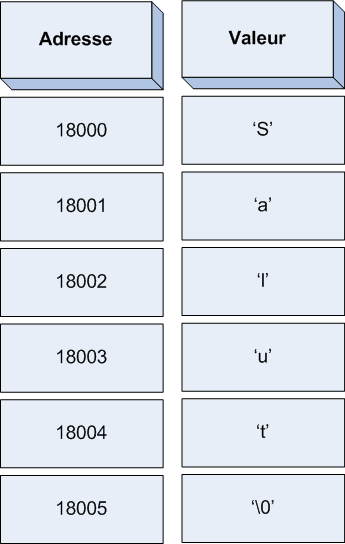
... et qu'on met dans chaine[0] la lettre 'S', dans chaine[1] la lettre 'a', etc... On peut ainsi former une chaîne de caractères, c'est-à-dire du texte :)  
  
Voici un schéma de la façon dont ça pourrait être stocké en mémoire (attention je vous préviens de suite, c'est un peu plus compliqué que ça en réalité, je vous explique après pourquoi) :



Comme on peut le voir, c'est un tableau qui prend 5 cases en mémoire pour représenter le mot "Salut". Pour la valeur, j'ai mis exprès les lettres entre apostrophes, pour indiquer que c'est un nombre qui est stocké et non une lettre. En réalité, dans la mémoire ce sont bel et bien les valeurs de ces lettres qui sont stockées ;)  
  
Oui mais attention, une chaîne de caractères ne contient pas que des lettres ! Le schéma que vous voyez ci-dessus est en fait incomplet.  
Une chaîne de caractère **doit impérativement contenir un caractère spécial à la fin de la chaîne**, appelé "Caractère de fin de chaîne". Ce caractère s'écrit '\0'.

Pourquoi devoir terminer une chaîne de caractères par un \0 ?

Tout simplement pour que votre ordinateur sache quand s'arrête la chaîne ! Le caractère \0 permet de dire : "Stop, c'est fini, y'a plus rien à lire après circulez !"  
  
Par conséquent, pour stocker le mot "Salut" (qui comprend 5 lettres) en mémoire, **il ne faut pas un tableau de 5 char, il faut un tableau de 6 char** !  
A chaque fois que vous créez une chaîne de caractères, vous allez donc devoir penser à prévoir de la place pour le caractère de fin de chaîne. Il faut toujours toujours toujours rajouter un bloc de plus dans le tableau pour stocker ce caractère \0, c'est impératif !  
  
Oublier le caractère de fin \0, c'est une source d'erreurs impitoyable du langage C. Je le sais, j'en ai fait les frais et pas qu'une fois :D  
  
  
En clair, le **bon**schéma de la chaîne de caractères "Salut" en mémoire est le suivant :



Comme vous le voyez, la chaîne prend 6 caractères et non pas 5, il va falloir s'y faire ;)  
La chaîne se termine par '\0', le caractère de fin de chaîne qui permet d'indiquer à l'ordinateur que la chaîne se termine là.

Voyez le caractère \0 comme un avantage. Grâce à lui, vous n'aurez pas à retenir la taille de votre tableau car il indique que le tableau s'arrête là :) Vous pourrez passer votre tableau de char à une fonction sans avoir à ajouter à côté une variable indiquant la taille du tableau.  
Cela n'est valable que pour les chaînes de caractères (c'est-à-dire le type char\*, qu'on peut aussi écrire char[] ). Pour les autres types de tableaux, vous êtes toujours obligés de retenir la taille du tableau quelque part.

**Création et initialisation de la chaîne**

Si on veut initialiser notre tableau chaine avec le texte "Salut", on peut utiliser la méthode old-school un peu bourrin :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | char chaine[6]; *// Tableau de 6 char pour stocker S-a-l-u-t + le \0*    chaine[0] = 'S';  chaine[1] = 'a';  chaine[2] = 'l';  chaine[3] = 'u';  chaine[4] = 't';  chaine[5] = '\0'; |

Cette méthode marche.  
On peut le vérifier en faisant un printf.  
  
Ah oui, j'allais oublier le printf : y'a encore un nouveau symbole à retenir ^^  
C'est le %s (s comme string, qui signifie "chaîne" en anglais).  
Voici le code complet qui crée une chaîne "Salut" en mémoire et qui l'affiche :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | #include <stdio.h>  #include <stdlib.h>  int main(int argc, char \*argv[])  {  char chaine[6]; *// Tableau de 6 char pour stocker S-a-l-u-t + le \0*  *// Initialisation de la chaîne (on écrit les caractères 1 à 1 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;  } |

Résultat :  
  
**Code : Console**

|  |
| --- |
| Salut |

Pfiou ^^  
Tout ça pour stocker "Salut" en mémoire et l'afficher quand même :D  
  
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 :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | int main(int argc, char \*argv[])  {  char chaine[] = "Salut"; *// La taille du tableau chaine est automatiquement calculée*  printf("%s", chaine);  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Salut |

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 rajoute 1 pour placer le caractère \0. Il écrit ensuite une à une les lettres du mot "Salut" en mémoire et rajoute l'\0 comme on l'a fait nous-mêmes manuellement quelques instants plus tôt.  
Bref, c'est bien pratique :)  
  
**Défaut** : ça ne marche que pour l'initialisation ! Vous ne pouvez pas écrire plus loin dans le code :  
**Code : C**

|  |  |
| --- | --- |
| 1 | chaine = "Salut"; |

Cette technique est donc à réserver à l'initialisation. Après cela, il faudra écrire les caractères manuellement un à un en mémoire ^^

**Récupération d'une chaîne via un scanf**

Vous pouvez enregistrer une chaîne rentrée par l'utilisateur via un scanf, en utilisant là encore le symbole %s.  
Seul problème : vous ne savez pas combien de caractères l'utilisateur va rentrer. 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-Edouard (beaucoup plus de caractères) ?  
  
Pour ça, 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] pour stocker le prénom. Ca donne l'impression de gâcher de la mémoire, mais souvenez-vous encore une fois que de la place en mémoire c'est pas ce qui manque (et y'a des programmes qui gâchent de la mémoire de manière bien pire que ça, si vous saviez :D )  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | int main(int argc, char \*argv[])  {  char prenom[100];  printf("Comment t'appelles-tu petit Zer0 ? ");  scanf("%s", prenom);  printf("Salut %s, je suis heureux de te rencontrer !", prenom);  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Comment t'appelles-tu petit Zer0 ? Mateo21  Salut Mateo21, je suis heureux de te rencontrer ! |

Voilà en gros comment ça se passe pour demander d'entrer du texte à l'utilisateur :)

**Fonctions de manipulation des chaînes**

Les chaînes de caractères sont, vous vous en doutez, fréquemment utilisées. Tous les mots, tous les textes que vous voyez à votre écran sont en fait des tableaux de char en mémoire qui fonctionnent de la manière que je viens de vous expliquer (ah on ne voit plus son ordinateur de la même façon du coup hein ? ^^ )  
Afin de nous aider un peu à manipuler les chaînes, on nous fournit dans la librairie string.h une pléthore de fonctions dédiées aux calculs sur des chaînes.  
  
Je ne peux pas vraiment toutes vous les présenter ici, ce serait un peu long et elles ne sont pas toutes forcément indispensables.  
Je vais me contenter de vous parler des principales dont vous aurez très certainement besoin dans peu de temps, ce qui fait déjà pas mal ;)

**Pensez à inclure string.h**

Même si cela devrait vous paraître évident, je préfère vous le préciser encore au cas où : comme on va utiliser une nouvelle librairie appelée string.h, vous devez l'inclure en haut des fichiers .c où vous en avez besoin :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | #include <string.h> |

Si vous ne le faites pas, l'ordinateur ne connaîtra pas les fonctions que je vais vous présenter car il n'aura pas les prototypes, et la compilation plantera.  
Bref, n'oubliez pas d'inclure cette librairie à chaque fois que vous utilisez des fonctions de manipulation de chaînes :)

**strlen : calculer la longueur d'une chaîne**

strlen est une fonction qui calcule la longueur d'une chaîne de caractères (sans compter le caractère \0 ).  
Vous devez lui envoyer un seul paramètre : votre chaîne de caractères ! Cette fonction vous retourne la longueur de la chaîne.  
  
Maintenant que vous savez ce qu'est un prototype, je vais vous donner le prototype des fonctions dont je vous parle. Les programmeurs s'en servent comme "mode d'emploi" de la fonction (même si quelques explications à côté ne sont jamais superflues :p ) :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | size\_t strlen(**const** char\* chaine); |

*size\_t* est un type spécial qui signifie que la fonction renvoie un nombre correspondant à une taille. Ce n'est pas un type de base comme int, long ou char, c'est un type "inventé". Nous apprendrons nous aussi à créer nos propres types de variables quelques chapitres plus loin.  
Pour le moment, on va se contenter de stocker la valeur renvoyée par strlen dans une variable de type *long*(l'ordinateur convertira de *size\_t* en *long*automatiquement). En toute rigueur, il faudrait plutôt stocker le résultat dans une variable de type *size\_t*, mais en pratique un *long* est suffisant pour cela.

La fonction prend un paramètre de type const char\*. Le const (qui signifie constante, rappelez-vous) fait que la fonction strlen "s'interdit" en quelque sorte de modifier votre chaîne. Quand vous voyez un const, vous savez que la variable n'est pas modifiée par la fonction, elle est juste lue.  
  
Testons la fonction strlen :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | int main(int argc, char \*argv[])  {  char chaine[] = "Salut";  long longueurChaine = 0;  *// On récupère la longueur de la chaîne dans longueurChaine*  longueurChaine = strlen(chaine);  *// On affiche la longueur de la chaîne*  printf("La chaine %s fait %ld caracteres de long", chaine, longueurChaine);  **return** 0;  } |

**Code : Console**

|  |
| --- |
| La chaine Salut fait 5 caracteres de long |

Cette fonction strlen est d'ailleurs facile à écrire. Il suffit de faire une boucle sur le tableau de char qui s'arrête quand on tombe sur le caractère \0. Un compteur s'incrémente à chaque tour de boucle, et c'est ce compteur que la fonction retourne.  
  
Allez, ça m'a donné envie d'écrire moi-même une fonction similaire à strlen :)  
Ca vous permettra en plus de bien comprendre comment la fonction marche :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | long longueurChaine(**const** char\* chaine);  int main(int argc, char \*argv[])  {  char chaine[] = "Salut";  long longueur = 0;  longueur = longueurChaine(chaine);  printf("La chaine %s fait %ld caracteres de long", chaine, longueur);  **return** 0;  }  long longueurChaine(**const** char\* chaine)  {  long nombreDeCaracteres = 0;  char caractereActuel = 0;  **do**  {  caractereActuel = chaine[nombreDeCaracteres];  nombreDeCaracteres++;  }  **while**(caractereActuel != '\0'); *// On boucle tant qu'on n'est pas arrivé à l'\0*  nombreDeCaracteres--; *// On retire 1 caractère de long pour ne pas compter l'\0*  **return** nombreDeCaracteres;  } |

La fonction longueurChaine fait une boucle sur le tableau chaine. Elle stocke les caractères un par un dans caractereActuel. Dès que caractèreActuel vaut '\0', la boucle s'arrête.  
A chaque passage dans la boucle, on ajoute 1 au nombre de caractères qu'on a analysé.  
  
A la fin de la boucle, on retire 1 caractère au nombre total de caractères qu'on a comptés. Cela permet de ne pas compter le caractère \0 dans le lot.  
Enfin, on retourne nombreDeCaracteres, et le tour est joué ;)

**strcpy : copier une chaîne dans une autre**

La fonction strcpy (comme "string copy") permet de copier une chaîne à l'intérieur d'une autre.  
Son prototype est :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | char\* strcpy(char\* copieDeLaChaine, **const** char\* chaineACopier); |

Cette fonction prend 2 paramètres :

* copieDeLaChaine : c'est un pointeur vers un char\* (tableau de char). C'est dans ce tableau que la chaîne sera copiée.
* chaineACopier : c'est un pointeur vers un autre tableau de char. Cette chaîne sera copiée dans copieDeLaChaine.

La fonction renvoie un pointeur sur copieDeLaChaine, ce qui n'est pas très utile. En général, on ne récupère pas ce que cette fonction renvoie.  
  
Allons tester ça :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | int main(int argc, char \*argv[])  {  */\* On crée une chaine "chaine" qui contient un peu de texte*  *et une copie (vide) de taille 100 pour être sûr d'avoir la place*  *pour la copie \*/*  char chaine[] = "Texte", copie[100] = {0};  strcpy(copie, chaine); *// On copie "chaine" dans "copie"*  *// Si tout s'est bien passé, la copie devrait être identique à chaine*  printf("chaine vaut : %s**\n**", chaine);  printf("copie vaut : %s**\n**", copie);  **return** 0;  } |

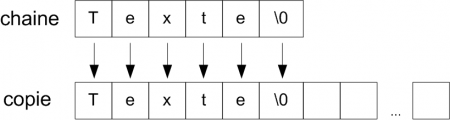
**Code : Console**

|  |
| --- |
| chaine vaut : Texte  copie vaut : Texte |

On voit que chaine vaut "Texte". Bon ça c'est normal ^^  
Par contre, on voit aussi que la variable copie, qui était vide au départ, a été remplie par le contenu de chaine. La chaine a donc bien été copiée dans "copie" :)

Vérifiez que la chaîne "copie" est assez grande pour accueillir le contenu de "chaine". Si, dans mon exemple, j'avais défini copie[5] (ce qui n'est pas suffisant car il n'y aurait pas eu de place pour l'\0), la fonction strcpy aurait "débordé en mémoire" et probablement fait planter votre programme. A éviter à tout prix, sauf si vous aimez faire crasher votre ordinateur bien sûr :)

Schématiquement, il s'est passé ça en mémoire :



Chaque caractère de chaine a été placé dans copie.  
La chaîne copie contient de nombreux caractères inutilisés, vous l'aurez remarqué. Je lui ai donné la taille 100 par sécurité, mais en toute rigueur la taille 6 aurait suffit. L'avantage de créer un tableau un peu plus grand, c'est que de cette façon la chaîne copie sera capable de recevoir d'autres chaînes peut-être plus grandes dans la suite du programme.

**strcat : concaténer 2 chaînes**

Cette fonction ajoute une chaîne à la suite d'une autre. On appelle cela la concaténation.  
Si j'ai :

* chaine1 = "Salut "
* chaine2 = "Mateo21"

Si je concatène chaine2 dans chaine1, alors chaine1 vaudra "Salut Mateo21".  
chaine2, elle, n'aura pas changé et vaudra donc toujours "Mateo21". Seule chaine1 est modifiée.  
  
C'est exactement ce que fait strcat, dont voici le prototype :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | char\* strcat(char\* chaine1, **const** char\* chaine2); |

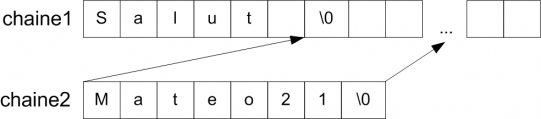
Comme vous pouvez le voir, chaine2 ne peut pas être modifiée car elle est définie comme constante dans le prototype de la fonction.  
La fonction retourne un pointeur vers chaine1 ce qui, comme pour strcpy, ne sert pas à grand-chose dans le cas présent, donc on peut ignorer ce que la fonction nous renvoie.  
  
La fonction ajoute à chaine1 le contenu de chaine2. Regardons-y de plus près :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | int main(int argc, char \*argv[])  {  */\* On crée 2 chaînes. chaine1 doit être assez grande pour accueillir*  *le contenu de chaine2 en plus, sinon risque de plantage \*/*  char chaine1[100] = "Salut ", chaine2[] = "Mateo21";  strcat(chaine1, chaine2); *// On concatène chaine2 dans chaine1*  *// Si tout s'est bien passé, chaine1 vaut "Salut Mateo21"*  printf("chaine1 vaut : %s**\n**", chaine1);  *// chaine2 n'a pas changé :*  printf("chaine2 vaut toujours : %s**\n**", chaine2);  **return** 0;  } |

**Code : Console**

|  |
| --- |
| chaine1 vaut : Salut Mateo21  chaine2 vaut toujours : Mateo21 |

Vérifiez absolument que chaine1 est assez grande pour qu'on puisse lui rajouter le contenu de chaine2, sinon vous ferez un débordement en mémoire qui peut conduire à un plantage.  
C'est pour cela que j'ai défini chaine1 de taille 100. Quant à chaine2, j'ai laissé l'ordinateur calculer sa taille (je n'ai donc pas précisé la taille) car cette chaîne n'est pas modifiée, il n'y a donc pas besoin de la rendre plus grande que nécessaire :)  
  
Schématiquement il s'est passé ça :



Le tableau chaine2 a été ajouté à la suite de chaine1 (qui comprenait une centaine de cases)  
L'\0 de chaine1 a été supprimé (en fait il a été remplacé par le M de Mateo21). En effet, il ne faut pas laisser un \0 au milieu de la chaîne sinon celle-ci aurait été "coupée" au milieu ! On ne met qu'un \0 à la fin de la chaîne, une fois qu'elle est finie.

**strcmp : comparer 2 chaînes**

strcmp compare 2 chaînes entre elles. Voici son prototype :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | int strcmp(**const** char\* chaine1, **const** char\* chaine2); |

Les variables chaine1 et chaine2 sont comparées. Comme vous le voyez, aucune d'elles n'est modifiée car elles sont indiquées comme constantes.  
  
Il est important de récupérer ce que la fonction renvoie. En effet, strcmp renvoie :

* 0 si les chaînes sont identiques
* Une autre valeur (positive ou négative) si les chaînes sont différentes

Il aurait été plus logique, je le reconnais, que la fonction renvoie 1 si les chaînes sont identiques pour dire "vrai" (rappelez-vous des booléens). Toutefois, comme ce n'est pas moi qui ai codé la fonction... :p  
Plus sérieusement, la fonction compare les valeurs de chacun des caractères un à un. Si tous les caractères sont identiques, elle renvoie 0. Si les caractères de la chaine1 sont supérieurs à ceux de la chaine2, la fonction renvoie un nombre positif. Si c'est l'inverse, la fonction renvoie un nombre négatif.  
Dans la pratique, on se sert surtout de strcmp pour vérifier si 2 chaînes sont identiques, point barre ;)

Voici un code de test :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | int main(int argc, char \*argv[])  {  char chaine1[] = "Texte de test", chaine2[] = "Texte de test";  **if** (strcmp(chaine1, chaine2) == 0) *// Si strcmp renvoie 0 (chaînes identiques)*  {  printf("Les chaines sont identiques**\n**");  }  **else**  {  printf("Les chaines sont differentes**\n**");  }  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Les chaines sont identiques |

Les chaînes étant identiques, la fonction strcmp a renvoyé le nombre 0.  
Notez que j'aurais pu stocker ce que renvoie strcmp dans une variable de type int. Toutefois, ce n'est pas obligatoire, on peut directement mettre la fonction dans le if comme je l'ai fait.  
  
Je n'ai pas grand-chose à rajouter à propos de cette fonction. Elle est assez simple à utiliser en fait, mais il ne faut pas oublier que 0 signifie "identique" et une autre valeur signifie "différent". C'est la seule source d'erreurs possible ici.

**strchr : rechercher un caractère**

La fonction strchr recherche un caractère dans une chaîne.  
Prototype :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | char\* strchr(**const** char\* chaine, int caractereARechercher); |

La fonction prend 2 paramètres :

* chaine : la chaîne dans laquelle la recherche doit être faite.
* caractereARechercher : le caractère que l'on doit rechercher dans la chaîne.

Vous remarquerez que caractereARechercher est de type int et non de type char. Ce n'est pas réellement un problème car, au fond, un caractère est et restera toujours un nombre ;)  
Néanmoins, on utilise quand même plus souvent un char qu'un int pour stocker un caractère en mémoire.

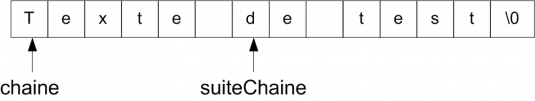
La fonction renvoie un pointeur vers **le premier caractère** qu'elle a trouvé, c'est-à-dire qu'elle renvoie l'adresse de ce caractère dans la mémoire. Elle renvoie NULL si elle n'a rien trouvé.  
Dans l'exemple suivant, je récupère ce pointeur dans suiteChaine :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | int main(int argc, char \*argv[])  {  char chaine[] = "Texte de test", \*suiteChaine = NULL;  suiteChaine = strchr(chaine, 'd');  **if** (suiteChaine != NULL) *// Si on a trouvé quelque chose*  {  printf("Voici la fin de la chaine a partir du premier d : %s", suiteChaine);  }  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Voici la fin de la chaine a partir du premier d : de test |

Avez-vous bien compris ce qu'il se passe ici ? C'est un peu particulier.  
En fait, suiteChaine est un pointeur comme chaine. Sauf que chaine pointe sur le premier caractère (le 'T' majuscule), tandis que suiteChaine pointe sur le premier caractère 'd' qui a été trouvé dans chaine.  
  
Le schéma suivant vous montre où pointe chaque pointeur :



chaine commence au début de la chaine ('T' majuscule), tandis que suiteChaine pointe sur le 'd' minuscule.  
  
Lorsque je fais un printf de suiteChaine, il est donc normal que l'on m'affiche juste "de test". La fonction printf affiche tous les caractères qu'elle rencontre ('d', 'e', ' ', 't', 'e', 's', 't') jusqu'à ce qu'elle tombe sur l'\0 qui lui dit que la chaîne s'arrête là.

***Variante***

Il existe une fonction strrchr strictement identique à strchr, sauf que celle-là renvoie un pointeur vers **le dernier caractère** qu'elle a trouvé dans la chaîne au lieu du premier ;)

**strpbrk : premier caractère de la liste**

Cette fonction ressemble beaucoup à la précédente. Celle-ci recherche un des caractères dans la liste que vous lui donnez sous forme de chaîne, contrairement à strchr qui ne peut rechercher qu'un seul caractère à la fois.  
  
Par exemple, si on forme la chaîne "xds" et qu'on en fait une recherche dans "Texte de test", la fonction renvoie un pointeur vers le premier de ces caractères qu'elle a trouvé dedans. En l'occurrence, le premier caractère de "xds" qu'elle trouve dans "Texte de test" est le x, donc strpbrk renverra un pointeur sur 'x'.  
  
Prototype :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | char\* strpbrk(**const** char\* chaine, **const** char\* lettresARechercher); |

Test :  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | int main(int argc, char \*argv[])  {  char \*suiteChaine;  *// On cherche la première occurrence de x, d ou s dans "Texte de test"*  suiteChaine = strpbrk("Texte de test", "xds");  **if** (suiteChaine != NULL)  {  printf("Voici la fin de la chaine a partir du premier des caracteres trouves : %s", suiteChaine);  }  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Voici la fin de la chaine a partir du premier des caracteres trouves : xte de test |

Pour cet exemple, j'ai directement écrit les valeurs à envoyer à la fonction (entre guillemets). On n'est en effet pas obligés d'employer une variable à tous les coups, on peut très bien écrire la chaîne directement.  
Il faut simplement retenir la règle suivante :

* Si vous utilisez les guillemets "", cela signifie **chaîne**.
* Si vous utilisez les apostrophes '', cela signifie **caractère**.

**strstr : rechercher une chaîne dans une autre**

Cette fonction recherche la première occurrence d'une chaîne dans une autre chaîne.  
Son prototype est :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | char\* strstr(**const** char\* chaine, **const** char\* chaineARechercher); |

Le prototype est similaire à strpbrk, mais attention à ne pas confondre : strpbrk recherche UN des caractères, tandis que strstr recherche toute la chaîne.  
  
Exemple :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | int main(int argc, char \*argv[])  {  char \*suiteChaine;  *// On cherche la première occurrence de "test" dans "Texte de test" :*  suiteChaine = strstr("Texte de test", "test");  **if** (suiteChaine != NULL)  {  printf("Premiere occurence de test dans Texte de test : %s**\n**", suiteChaine);  }  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Premiere occurence de test dans Texte de test : test |

La fonction strstr recherche la chaîne "test" dans "Texte de test".   
Elle renvoie, comme les autres, un pointeur quand elle a trouvé ce qu'elle cherchait. Elle renvoie NULL si elle n'a rien trouvé.

Pensez à vérifier si vos fonctions de recherche n'ont pas renvoyé NULL. Si vous ne le faites pas et que vous essayez d'afficher une chaîne qui pointe sur NULL, votre programme plantera. Votre OS fermera brutalement votre programme car il aura essayé d'accéder à l'adresse NULL à laquelle il n'a pas le droit d'accéder.

Jusqu'ici, je me suis contenté d'afficher la chaîne à partir du pointeur retourné par les fonctions. Dans la pratique, ça n'est pas très utile ;) Vous ferez juste un if (resultat != NULL) pour savoir si la recherche a trouvé quelque chose ou si elle n'a rien trouvé, et vous afficherez "*Le texte que vous recherchiez a été trouvé*".  
Enfin, cela dépend de votre programme, mais en tout cas ces fonctions sont la base si vous voulez faire un traitement de texte :)

**sprintf : écrire dans une chaîne**

Cette fonction se trouve dans stdio.h contrairement aux autres fonctions que nous avons étudiées jusqu'ici, qui étaient dans string.h

Ce nom doit vaguement vous rappeler quelque chose ^^  
Cette fonction ressemble énormément au printf que vous connaissez mais, au lieu d'écrire à l'écran, sprintf écrit dans... une chaîne ! D'où son nom d'ailleurs, qui commence par le "s" de "string" (chaîne en anglais).  
  
C'est une fonction très pratique pour mettre en forme une chaîne. Petit exemple :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | #include <stdio.h>  #include <stdlib.h>  int main(int argc, char \*argv[])  {  char chaine[100];  long age = 15;  *// On écrit "Tu as 15 ans" dans chaine*  sprintf(chaine, "Tu as %ld ans !", age);  *// On affiche chaine pour vérifier qu'elle contient bien cela :*  printf("%s", chaine);  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Tu as 15 ans ! |

Elle s'utilise de la même manière que printf, mis à part le fait que vous devez lui donner en premier paramètre un pointeur vers la chaîne qui doit recevoir le texte.  
Dans mon exemple, j'écris dans chaine "Tu as %ld ans", où %ld est remplacé par le contenu de la variable age. Toutes les règles du printf s'appliquent, vous pouvez donc si vous le voulez mettre des %s pour insérer d'autres chaînes à l'intérieur de votre chaîne :)  
  
Comme d'hab, vérifiez que votre chaîne est suffisamment grande pour accueillir tout le texte que le sprintf va lui envoyer. Sinon, ben... boum :-°

**Les includes**

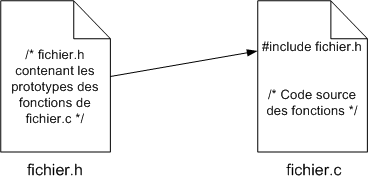
Comme je vous l'ai expliqué dans les tous premiers chapitres du cours, on trouve dans les codes sources des lignes un peu particulières appelées **directives de préprocesseur**.  
Ces directives de préprocesseur ont la caractéristique suivante : elles commencent toujours par le symbole #. Elles sont 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, je vous l'ai dit plus tôt.  
On s'en sert en particulier pour inclure des fichiers .h comme les fichiers .h des librairies (stdlib.h, stdio.h, string.h, math.h...) et vos propres fichiers .h.  
  
Pour inclure un fichier .h se trouvant dans **le dossier où est installé votre IDE**, vous devez utiliser les chevrons < > :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | #include <stdlib.h> |

Pour inclure un fichier .h se trouvant dans **le dossier de votre projet**, vous devez utiliser les guillemets :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | #include "monfichier.h" |

Concrètement, le préprocesseur est démarré avant la compilation. Il parcourt tous vos fichiers à la recherche de directives de préprocesseur, ces fameuses lignes qui commencent par un #  
Lorsqu'il rencontre la directive #include, il met littéralement le contenu du fichier indiqué à l'endroit du #include.  
  
Supposons que j'aie un "fichier.c" contenant le code de mes fonctions et un "fichier.h" contenant les prototypes des fonctions de fichier.c. On pourrait résumer la situation dans ce schéma tout simple :



Tout le contenu de fichier.h est mis à l'intérieur de fichier.c, à l'endroit où il y a la directive #include fichier.h  
  
Imaginons qu'on ait dans le fichier.c :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | #include "fichier.h"  long maFonction(int truc, double bidule)  {  */\* Code de la fonction \*/*  }  void autreFonction(long valeur)  {  */\* Code de la fonction \*/*  } |

Et dans le fichier.h :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2 | long maFonction(int truc, double bidule);  void autreFonction(long valeur); |

Lorsque le préprocesseur passe par là, juste avant la compilation de fichier.c, il met fichier.h dans fichier.c. Au final, le code source de fichier.c *juste avant* la compilation ressemble à ça :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | long maFonction(int truc, double bidule);  void autreFonction(long valeur);  long maFonction(int truc, double bidule)  {  */\* Code de la fonction \*/*  }  void autreFonction(long valeur)  {  */\* Code de la fonction \*/*  } |

Le contenu du .h est venu se mettre à l'emplacement de la ligne #include :)  
  
Ce n'est pas bien compliqué à comprendre, je pense d'ailleurs que bon nombre d'entre vous devaient se douter que ça fonctionnait comme ça.  
Au moins là avec ces explications supplémentaires, j'espère avoir mis tout le monde d'accord ;) Le #include ne fait rien d'autre qu'insérer un fichier dans un autre, c'est important de bien le comprendre.

Si on a décidé de mettre les prototypes dans les .h, au lieu de tout mettre dans les .c, c'est principalement par principe. On pourrait à priori mettre les prototypes en haut des .c (d'ailleurs dans certains très petits programmes on le fait parfois), mais pour des questions d'organisation il est *vivement*conseillé de placer ses prototypes dans des .h.

**Les defines**

Nous allons découvrir maintenant une nouvelle directive de préprocesseur : le #define.  
  
Cette directive permet de définir une **constante de préprocesseur**. Cela permet d'associer une valeur à un mot. Voici un exemple :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | #define NOMBRE\_VIES\_INITIALES 3 |

Vous mettez dans l'ordre :

* le #define
* le mot auquel la valeur va être associée
* la valeur du mot

Attention, malgré les apparences (notamment le nom que l'on a l'habitude de mettre en majuscules) c'est très différent des constantes que nous avons étudiées jusqu'ici, telles que :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | **const** long NOMBRE\_VIES\_INITIALES = 3; |

Les constantes occupaient de la place en mémoire. Même si la valeur ne changeait pas, votre nombre "3" était stocké quelque part dans la mémoire. Ce n'est pas le cas des constantes de préprocesseur :)  
  
Comment ça fonctionne ? En fait, le #define remplace dans votre code source tous les mots par leur valeur correspondante. C'est comme la fonction "rechercher / remplacer" de Word si vous voulez ;)  
Ainsi, la ligne :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | #define NOMBRE\_VIES\_INITIALES 3 |

... remplace dans le fichier chaque NOMBRE\_VIES\_INITIALES par 3.  
  
  
Voici un exemple de fichier .c avant passage du préprocesseur :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #define NOMBRE\_VIES\_INITIALES 3  int main(int argc, char \*argv[])  {  long vies = NOMBRE\_VIES\_INITIALES;  */\* Code ...\*/* |

Après passage du préprocesseur :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5 | int main(int argc, char \*argv[])  {  long vies = 3;  */\* Code ...\*/* |

Avant la compilation, tous les #define auront donc été remplacés par les valeurs correspondantes. Le compilateur "voit" le fichier après passage du préprocesseur, dans lequel tous les remplacements auront été effectués.

Quel intérêt par rapport à l'utilisation de constantes comme on l'a vu jusqu'ici ?

Eh bien, comme je vous l'ai dit ça **ne prend pas de place en mémoire**. C'est logique, vu que lors de la compilation il ne reste plus que des nombres dans le code source.  
  
Un autre intérêt est que le remplacement se fait dans **tout le fichier** dans lequel se trouve le #define. Si vous aviez défini une constante en mémoire dans une fonction, celle-ci n'aurait été valable que dans la fonction puis aurait été supprimée. Le #define en revanche s'appliquera à toutes les fonctions du fichier, ce qui peut s'avérer vraiment pratique pour le programmeur.  
  
Un exemple concret d'utilisation des #define ?  
En voici un que vous ne tarderez pas à utiliser. Lorsque vous ouvrirez une fenêtre en C, vous aurez probablement envie de définir des constantes de préprocesseur pour indiquer les dimensions de la fenêtre :  
  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2 | #define LARGEUR\_FENETRE 800  #define HAUTEUR\_FENETRE 600 |

L'avantage est que si plus tard vous décidez de changer la taille de la fenêtre (parce que ça vous semble trop petit), il vous suffira de modifier les #define puis de recompiler.  
  
**A noter : les #define sont généralement placés dans des .h, à côté des prototypes** (vous pouvez d'ailleurs aller voir les .h des librairies comme stdlib.h, vous verrez qu'il y a des #define !).  
Les #define sont donc "faciles d'accès", vous pouvez changer les dimensions de la fenêtre en modifiant les #define plutôt que d'aller chercher au fond de vos fonctions l'endroit où vous ouvrez la fenêtre pour modifier les dimensions. C'est donc du temps de gagné pour le programmeur ;)  
  
En résumé, les constantes de préprocesseur permettent de "configurer" votre programme avant sa compilation. C'est une sorte de mini-configuration en fait ;)

**Un define pour la taille des tableaux**

On utilise souvent les defines pour définir la taille des tableaux. On fait par exemple :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | #define TAILLE\_MAX 1000  int main(int argc, char \*argv[])  {  char chaine1[TAILLE\_MAX], chaine2[TAILLE\_MAX];  *// ...* |

Mais... Je croyais qu'on ne pouvait pas mettre de variable entre les crochets lors d'une définition de tableau ?

Oui, mais TAILLE\_MAX n'est PAS une variable ;)  
En effet je vous l'ai dit, le préprocesseur transforme le fichier avant compilation en :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | int main(int argc, char \*argv[])  {  char chaine1[1000], chaine2[1000];  *// ...* |

... et cela est valide :)  
  
En définissant TAILLE\_MAX ainsi, vous pouvez vous en servir pour créer des tableaux d'une certaine taille. Si cela s'avère insuffisant par le futur, vous n'aurez qu'à modifier la ligne du #define, recompiler, et vos tableaux de char prendront tous la nouvelle taille que vous aurez indiquée :)

**Calculs dans les defines**

Il est possible de faire quelques petits calculs dans les defines.  
Par exemple, ce code crée une constante LARGEUR\_FENETRE, une autre HAUTEUR\_FENETRE, puis une troisième NOMBRE\_PIXELS qui contiendra le nombre de pixels affichés à l'intérieur de la fenêtre (le calcul est simple : largeur \* hauteur) :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3 | #define LARGEUR\_FENETRE 800  #define HAUTEUR\_FENETRE 600  #define NOMBRE\_PIXELS (LARGEUR\_FENETRE \* HAUTEUR\_FENETRE) |

La valeur de NOMBRE\_PIXELS est remplacée avant la compilation par (LARGEUR\_FENETRE \* HAUTEUR\_FENETRE), c'est-à-dire par (800 \* 600), ce qui fait 480000 ;)  
Mettez toujours votre calcul entre parenthèses comme je l'ai fait.  
  
Vous pouvez faire toutes les opérations de base que vous connaissez : addition (+), soustraction (-), multiplication (\*), division (/) et modulo (%)

**Les constantes prédéfinies**

En plus des constantes que vous pouvez définir vous-mêmes, il existe quelques constantes prédéfinies par le préprocesseur. A l'heure où j'écris ces lignes, je dois vous dire que je ne les ai encore jamais utilisées, mais il n'est pas impossible que vous leur en trouviez une utilité donc je vais vous les présenter ;)  
  
Chacune de ces constantes commence et se termine par 2 symboles underscore \_ (que vous trouverez sous le chiffre 8, tout du moins si vous avez un clavier AZERTY).

* \_\_LINE\_\_ : donne le numéro de la ligne actuelle
* \_\_FILE\_\_ : donne le nom du fichier actuel
* \_\_DATE\_\_ : donne la date de la compilation
* \_\_TIME\_\_ : donne l'heure de la compilation

Je pense que ces constantes peuvent être utiles pour gérer des erreurs, en faisant par exemple ceci :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2 | printf("Erreur a la ligne %ld du fichier %s**\n**", \_\_LINE\_\_, \_\_FILE\_\_);  printf("Ce fichier a ete compile le %s a %s**\n**", \_\_DATE\_\_, \_\_TIME\_\_); |

**Code : Console**

|  |
| --- |
| Erreur a la ligne 9 du fichier main.c  Ce fichier a ete compile le Jan 13 2006 a 19:21:10 |

**Les définitions simples**

Il est aussi possible de faire tout simplement :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | #define CONSTANTE |

... sans préciser de valeur.  
Cela veut dire pour le préprocesseur que le mot CONSTANTE est défini, tout simplement. Il n'a pas de valeur, mais il "existe".

J'en vois vraiment pas l'intérêt !?

L'intérêt est moins évident que tout à l'heure, mais il y en a un et nous allons le découvrir tout à l'heure :)

**Les macros**

Nous avons vu qu'avec le #define on pouvait demander au préprocesseur de remplacer un mot par une valeur. Par exemple :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | #define NOMBRE 9 |

... signifie que tous les "NOMBRE" de votre code seront remplacés par 9. Nous avons vu qu'il s'agissait en fait d'un simple rechercher / remplacer fait par le préprocesseur avant la compilation.  
  
J'ai du nouveau ! :)  
En fait, le #define est encore plus puissant que ça. Il permet de remplacer aussi par... du code source tout entier ! Quand on utilise #define pour rechercher / remplacer un mot par un code source, on dit qu'on crée **une macro**.

**Macro sans paramètres**

Voici un exemple de macro très simple :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | #define COUCOU() printf("Coucou"); |

Ce qui change ici, c'est les parenthèses qu'on a ajoutées après le mot-clé (ici **COUCOU()**). Nous verrons à quoi elles peuvent servir tout à l'heure.  
  
Testons la macro dans un code source :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | #define COUCOU() printf("Coucou");  int main(int argc, char \*argv[])  {  COUCOU()  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Coucou |

Je vous l'accorde, ce n'est pas original pour le moment ;)  
  
Ce qu'il faut bien comprendre déjà, c'est que les macros ne sont en fait que des bouts de code qui sont directement remplacés dans votre code source juste avant la compilation.  
Le code qu'on vient de voir ressemblera en fait à ça lors de la compilation :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int main(int argc, char \*argv[])  {  printf("Coucou");  **return** 0;  } |

Si vous avez compris ça, vous avez compris le principe de base des macros déjà :)

Mais... On ne peut mettre qu'une seule ligne de code par macro ?

Non, heureusement il est possible de mettre plusieurs lignes de code à la fois. Il suffit de mettre un \ avant chaque nouvelle ligne, comme ceci :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #define RACONTER\_SA\_VIE() printf("Coucou, je m'appelle Brice\n"); \  printf("J'habite a Nice\n"); \  printf("J'aime la glisse\n");  int main(int argc, char \*argv[])  {  RACONTER\_SA\_VIE()  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Coucou, je m'appelle Brice  J'habite a Nice  J'aime la glisse |

Remarquez dans le main que l'appel de la macro ne prend pas de point-virgule à la fin. En effet, c'est une ligne pour le préprocesseur, elle ne nécessite donc pas d'être terminée par un point-virgule

**Macro avec paramètres**

Pour le moment, on a vu comment faire une macro sans paramètre, c'est-à-dire avec rien entre les parenthèses. Le principal intérêt de ce type de macros, c'est de pouvoir "raccourcir" un code un peu long, surtout s'il est amené à être répété de nombreuses fois dans votre code source.  
  
Cependant, les macros deviennent réellement intéressantes lorsqu'on leur met des paramètres. Cela marche quasiment comme avec les fonctions.  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #define MAJEUR(age) if (age >= 18) \  printf("Vous etes majeur\n");  int main(int argc, char \*argv[])  {  MAJEUR(22)  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Vous etes majeur |

Notez qu'on aurait aussi pu rajouter un else pour afficher "Vous êtes mineur". Essayez de le faire pour vous entraîner, ce n'est pas bien difficile. N'oubliez pas de mettre un antislash \ avant chaque nouvelle ligne

Le principe de notre macro est assez intuitif je trouve :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2 | #define MAJEUR(age) if (age >= 18) \  printf("Vous etes majeur\n"); |

On met entre parenthèses le nom d'une "variable" qu'on nomme age. Dans tout notre code de remplacement, age sera remplacé par le nombre qui est indiqué lors de l'appel à la macro (ici c'est 22).  
  
Ainsi, notre code source de tout à l'heure ressemblera à ceci juste après le passage du préprocesseur :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int main(int argc, char \*argv[])  {  **if** (22 >= 18)  printf("Vous etes majeur**\n**");  **return** 0;  } |

Le code source a été mis à la place de l'appel de la macro, et la valeur de la "variable" age a été mise directement dans le code source de remplacement.  
  
Il est possible aussi de créer une macro qui prend plusieurs paramètres :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #define MAJEUR(age, nom) if (age >= 18) \  printf("Vous etes majeur %s\n", nom);  int main(int argc, char \*argv[])  {  MAJEUR(22, "Maxime")  **return** 0;  } |

Voilà en gros tout ce qu'on peut dire sur les macros. Il faut donc retenir que c'est un simple remplacement de code source qui a l'avantage de pouvoir prendre des paramètres.

Normalement vous ne devriez pas avoir besoin d'utiliser les macros très souvent. Toutefois, certaines librairies assez complexes comme wxWidgets ou QT (librairies de création de fenêtres que nous étudierons bien plus tard) utilisent beaucoup de macros. Il est donc préférable de savoir comment cela fonctionne dès maintenant pour ne pas être bêtement perdu plus tard ;)

**Les conditions**

Tenez-vous bien : il est possible de réaliser des conditions en langage préprocesseur :D  
Voici comment cela fonctionne :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5 | #if condition  */\* Code source à compiler si la condition est vraie \*/*  #elif condition2  */\* Sinon` si la condition 2 est vraie` compiler ce code source \*/*  #endif |

Le mot-clé #if permet d'insérer une condition de préprocesseur. #elif signifie "else if" (sinon si).  
La condition s'arrête lorsque vous insérez un #endif. Vous noterez qu'il n'y a pas d'accolades en préprocesseur.  
  
L'intérêt, c'est qu'on peut ainsi faire des **compilations conditionnelles**.  
En effet, si la condition est vraie, le code qui suit sera compilé. Sinon, il sera tout simplement supprimé du fichier pour le temps de la compilation. Il n'apparaîtra donc pas dans le programme final.

**#ifdef, #ifndef**

Nous allons voir maintenant l'intérêt de faire un #define d'une constante sans préciser de valeur, comme je vous l'ai montré tout à l'heure :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | #define CONSTANTE |

En effet, il est possible d'utiliser #ifdef pour dire "Si la constante est définie".  
#ifndef, lui, sert à dire "Si la constante n'est pas définie".  
  
On peut alors imaginer ceci :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #define WINDOWS  #ifdef WINDOWS  */\* Code source pour Windows \*/*  #endif  #ifdef LINUX  */\* Code source pour Linux \*/*  #endif  #ifdef MAC  */\* Code source pour Mac \*/*  #endif |

C'est comme ça que font les programmes multi plateformes pour s'adapter à l'OS par exemple :)  
Alors, bien entendu, il faut recompiler le programme pour chaque OS (ce n'est pas magique :D). Si vous êtes sous Windows, vous mettez un #define WINDOWS en haut, puis vous compilez.  
Si vous voulez compiler votre programme pour Linux (avec la partie du code source spécifique à Linux), alors vous devrez modifier le define et mettre à la place : #define LINUX. Recompilez, et cette fois c'est la portion de code source pour Linux qui sera compilée, les autres parties étant ignorées.

**#ifndef pour éviter les inclusions infinies**

#ifndef est très utilisé dans les .h pour éviter les "inclusions infinies".

Comment ça une inclusion infinie ? ?

Imaginez, c'est très simple.  
J'ai un fichier A.h et un fichier B.h. Le fichier A.h contient un include du fichier B.h. Le fichier B est donc inclus dans le fichier A.  
Mais, et c'est là le hic, supposez que le fichier B.h contienne à son tour un include du fichier A.h ? Ca arrive quelques fois en programmation ! Le premier fichier a besoin du second pour fonctionner, et le second a besoin du premier.  
  
Si on y réfléchit 10 petites secondes, on imagine vite ce qu'il va se passer :

1. L'ordinateur lit A.h et voit qu'il faut inclure B.h
2. Il lit B.h pour l'inclure, et là il voit qu'il faut inclure A.h
3. Donc il inclut A.h dans B.h, mais dans A.h on lui indique qu'il doit inclure B.h !
4. Rebelote, il va voir B.h et voit à nouveau qu'il faut inclure A.h
5. etc etc.

Pas besoin d'être un pro pour comprendre que ça ne s'arrêtera jamais !  
En fait, à force de faire trop d'inclusions, le préprocesseur s'arrêtera en disant "*y'en a marre des inclusions !*" et du coup bah votre compilation plantera :p  
  
Comment diable faire pour éviter cet affreux cauchemar ?  
Voici l'astuce. Désormais, je vous demande de faire comme ça **dans TOUS vos fichiers .h** sans exception :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | #ifndef DEF\_NOMDUFICHIER *// Si la constante n'a pas été définie` le fichier n'a jamais été inclus*  #define DEF\_NOMDUFICHIER *// On définit la constante pour que la prochaine fois le fichier ne soit plus inclus*  */\* Contenu de votre fichier .h (autres includes` prototypes de vos fonctions` defines...) \*/*  #endif |

Vous mettrez en fait tout le contenu de votre fichier .h (à savoir vos autres includes, vos prototypes, vos defines...) *entre*le #ifndef et le #endif.  
Comprenez-vous bien comment ce code fonctionne ? Je ne l'ai pas compris du premier coup quand on me l'a expliqué moi pour être tout à fait franc :D  
  
Imaginez que le fichier .h est inclus pour la première fois. Il lit la condition "Si la constante DEF\_NOMDUFICHIER n'a pas été définie". Comme c'est la première fois que le fichier est lu, la constante n'est pas définie, donc le préprocesseur rentre à l'intérieur du if.  
  
La première instruction qu'il rencontre est justement :  
**Code : C**

|  |  |
| --- | --- |
| 1 | #define DEF\_NOMDUFICHIER |

Maintenant, la constante est définie. La prochaine fois que le fichier sera inclus, la condition ne sera plus vraie, et donc le fichier ne risque plus d'être réinclus.  
  
Bien entendu, vous appelez votre constante comme vous voulez. Moi je l'appelle DEF\_NOMDUFICHIER par habitude, maintenant c'est chacun ses petites manies ;)  
Ce qui compte en revanche, et j'espère que vous l'aviez bien compris, c'est de changer de nom de constante à chaque fichier .h différent. Il ne faut pas que ça soit la même constante pour tous les fichiers .h, sinon seul le premier fichier .h serait lu et pas les autres ^^  
Vous remplacerez donc NOMDUFICHIER par le nom de votre fichier .h.

Si vous voulez vérifier que je ne suis pas en train de vous raconter des bêtises, je vous invite à aller consulter les .h des librairies standard sur votre disque dur. Vous verrez qu'ils sont TOUS construits sur le même principe (un ifndef au début et un endif à la fin). Ils s'assurent ainsi qu'il ne pourra pas y avoir d'inclusions infinies.

Créez vos propres types de variables !

**Définir une structure**

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 type long, char, int, double à la fois :)  
  
Les structures sont généralement **définies dans les fichiers .h**, au même titre donc que les prototypes et les defines.  
Voici un exemple de structure :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **struct** NomDeVotreStructure  {  long variable1;  long 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.  
Vous n'êtes pas obligés de faire comme moi. Le principal est que vous soyez organisés :)

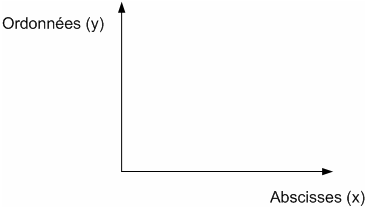
Après le nom de votre structure, vous ouvrez les accolades et les fermez plus loin, comme pour une fonction.

**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 mettez les variables dont est composée votre structure. Une structure est généralement composée d'au moins 2 "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... Il n'y a pas de miracle, un type "Fichier" n'est donc composé que de nombres de base :)

**Exemple de structure**

Des idées de structures, j'en ai des centaines en tête ^^  
Imaginons par exemple que vous vouliez créer une variable qui stocke les coordonnées d'un point à l'écran. Vous aurez très certainement besoin d'une structure comme cela lorsque vous ferez des jeux 2D dans la partie III, c'est donc l'occasion de s'avancer un peu là :)  
  
Pour ceux chez qui le mot "géométrie" provoque des apparitions de boutons inexplicables sur tout le visage, voici un petit rappel fondamental sur la 2D :



Lorsqu'on travaille en 2D (2 dimensions), on a 2 axes : l'axe des abscisses (de gauche à droite) et l'axe des ordonnées (de haut en bas).  
On a l'habitude d'exprimer les abscisses par une variable appelée x, et les ordonnées par y.  
  
Etes-vous capables d'écrire une structure "Coordonnees" capable de stocker à la fois la valeur de l'abscisse (x) et celle de l'ordonnée (y) d'un point ?  
Allons allons, ce n'est pas bien difficile :)  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5 | **struct** Coordonnees  {  long x; *// Abscisses*  long y; *// Ordonnées*  }; |

Notre structure s'appelle Coordonnee et est composée de 2 variables : x et y, c'est-à-dire l'abscisse et l'ordonnée.  
  
Si on le voulait, on pourrait facilement faire une structure Coordonnees pour de la 3D : il suffirait d'ajouter une troisième variable (par exemple "z") qui indiquerait la hauteur. Et hop, on aurait une structure faite pour gérer des points en 3D dans l'espace ;)

**Tableaux dans une structure**

Les structures peuvent contenir des tableaux. Ca tombe bien, on va pouvoir ainsi placer des tableaux de char (chaînes de caractères) sans problème !  
Allez, imaginons une structure "Personne" qui stockerait diverses informations sur une personne :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **struct** Personne  {  char nom[100];  char prenom[100];  char adresse[1000];    long age;  int garcon; *// Booléen : 1 = garçon, 0 = fille*  }; |

Cette structure est composée de 5 sous-variables. Les 3 premières sont des chaînes, qui stockeront le nom, le prénom et l'adresse de la personne.  
Les 2 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 (enfin évitez d'en mettre une centaine ça va faire beaucoup quand même :D ).

**Utilisation d'une structure**

Maintenant que notre structure est définie dans le .h, on va pouvoir l'utiliser dans une fonction de notre fichier .c.  
Voici comment créer une variable de type Coordonnees (la structure qu'on a définie plus haut) :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | #include "main.h" *// Inclusion du .h qui contient les prototypes et structures*  int main(int argc, char \*argv[])  {  **struct** Coordonnees point; *// Création d'une variable appelée "point" de type Coordonnees*  **return** 0;  } |

Nous avons ainsi créé une variable "point" de type Coordonnees. Cette variable est automatiquement composée de 2 sous-variables : x et y (son abscisse et son ordonnées).

On est obligés de mettre le mot "struct" lors de la définition de la variable ?

Oui, cela permet à l'ordinateur de différencier un type de base (comme "int") d'un type personnalisé, comme "Coordonnees".  
Toutefois, les programmeurs trouvent un peu lourd de mettre le mot "struct" à chaque définition de variable personnalisée. Pour régler ce problème, ils ont inventé une instruction spéciale : **le typedef**.

**Le typedef**

Retournons dans le fichier .h qui contient la définition de notre structure Coordonnees.  
Nous allons ajouter une instruction appelée typedef qui sert à créer un alias de structure, c'est-à-dire à dire que telle chose est équivalente à écrire telle autre chose.  
  
Nous allons ajouter une ligne commençant par typedef juste avant la définition de la structure :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **typedef** **struct** Coordonnees Coordonnees;  **struct** Coordonnees  {  long x;  long y;  }; |

Cette ligne doit être découpée en 3 morceaux (non je n'ai pas bégayé le mot "Coordonnees" :lol: ) :

* **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 "Ecrire 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.  
On peut donc retourner dans notre main et écrire tout simplement :  
  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5 | int main(int argc, char \*argv[])  {  Coordonnees point; *// L'ordinateur comprend qu'il s'agit de "struct Coordonnees" grâce au typedef*  **return** 0;  } |

Je vous recommande de faire un typedef comme je l'ai fait ici pour le type Coordonnees. La plupart des programmeurs font comme cela. Ca leur évite d'avoir à écrire le mot "struct" partout, et vu que ce sont des feignasses ça les arrange :D (je rigole hein les gars, tapez pas ^^ )

**Modifier les composantes de la structure**

Maintenant que notre variable point est créée, nous voulons modifier ses coordonnées.  
Comment accéder au x et au y de point ? Comme cela :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | int main(int argc, char \*argv[])  {  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 comme cela qu'on note une coordonnée en mathématiques :p )  
  
Pour accéder donc à chaque composante de la structure, vous devez écrire :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | variable.nomDeLaComposante |

Le point fait la séparation entre la variable et la composante.  
  
  
Si on prend la structure "Personne" de tout à l'heure et qu'on demande le nom et le prénom, on devra faire comme ça :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | int main(int argc, char \*argv[])  {  Personne utilisateur;  printf("Quel est votre nom ? ");  scanf("%s", utilisateur.nom);  printf("Votre prenom ? ");  scanf("%s", utilisateur.prenom);  printf("Vous vous appelez %s %s", utilisateur.prenom, utilisateur.nom);  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Quel est votre nom ? Dupont  Votre prenom ? Jean  Vous vous appelez Jean Dupont |

On envoie la variable "utilisateur.nom" à scanf qui écrira directement dans notre variable "utilisateur".  
On fait de même pour "prenom", et on pourrait aussi le faire pour l'adresse, l'âge et le sexe, mais j'ai la flemme de le faire ici (je dois être programmeur, c'est pour ça :D ).  
  
  
Vous auriez pu faire la même chose sans connaître les structures, en créant juste une variable nom et une autre "prenom".  
Mais l'intérêt ici est que vous pouvez créer une autre variable de type "Personne" qui aura aussi son propre nom, son propre prénom etc etc :)  
On peut donc faire :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | Personne joueur1, joueur2; |

... et stocker ainsi les informations sur chaque joueur.  
  
Mais le C, c'est plus fort encore ! On peut créer un tableau de Personne !  
C'est facile à faire :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | Personne joueurs[2]; |

Et ensuite, vous accédez par exemple au nom du joueur n°0 en tapant :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | joueurs[0].nom |

L'avantage d'utiliser un tableau ici, c'est que vous pouvez faire une boucle pour demander les infos du joueur 1 et du joueur 2, sans avoir à répéter 2 fois le même code. Il suffit de parcourir le tableau joueur et de demander à chaque fois nom, prénom, adresse...  
  
**Exercice**: créez ce tableau de type Personne et demandez les infos de chacun grâce à une boucle (qui se répète tant qu'il y a des joueurs). Faites un petit tableau de 2 joueurs pour commencer, mais si ça vous amuse vous pourrez agrandir la taille du tableau plus tard.  
Affichez à la fin du programme les infos que vous avez recueillies sur chacun des joueurs ;)  
  
Ce code est facile à écrire, vous n'avez pas besoin de correction, vous êtes des grands maintenant :p

**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 vaut une valeur qui n'a aucun sens, comme -84570.  
  
Pour rappel, voici comment on initialise :

* **Une variable** : on met sa valeur à 0 (ça c'est 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, ça va un peu ressembler à l'initialisation d'un tableau qu'on avait vue. En effet, on peut faire à la déclaration de la variable :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | Coordonnees point = {0, 0}; |

Cela mettra, 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 mettant juste "" (rien entre les guillemets). Je ne vous ai pas parlé de cette possibilité dans le chapitre sur les chaînes je crois, mais il n'est jamais trop tard pour l'apprendre ^^  
On peut donc initialiser dans l'ordre nom, prenom, adresse, age et garcon comme ceci :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | 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.  
Toutefois, 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. Revoyez le "fil rouge" du chapitre sur les pointeurs si vous avez un trou de mémoire à ce sujet ;)  
  
Il va donc falloir apprendre à utiliser des pointeurs sur des structures (hummm ça donne faim ça :D ).  
Nous allons justement voir comment faire ça maintenant ^^

**Pointeur de structure**

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

|  |  |
| --- | --- |
| 1 | Coordonnees\* point = NULL; |

On a ainsi un pointeur de Coordonnees appelé "point".  
Comme un rappel ne fera de mal à personne, je tiens à vous répéter que l'on aurait aussi pu mettre l'étoile devant le nom du pointeur, cela revient exactement au même :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | Coordonnees \*point = NULL; |

Je fais d'ailleurs assez souvent comme cela, car si l'on veut définir plusieurs pointeurs sur la même ligne, on est obligés de mettre l'étoile devant chaque nom de pointeur :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | Coordonnees \*point1 = NULL, \*point2 = NULL; |

Bon, mais ça pour le moment c'est du domaine des pointeurs, ça n'a pas de rapport direct avec les structures ;)

**Envoi de la structure à une fonction**

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 pour cet exemple : on va juste créer une variable de type Coordonnees dans le main et envoyer son adresse à la fonction initialiserCoordonnees. Cette fonction aura pour rôle de mettre tous les éléments de la structure à 0.

Nous faisons cela juste à titre d'exemple. En pratique, il serait plus simple d'initialiser la structure comme on a appris à le faire un peu plus haut :  
**Code : C**

|  |  |
| --- | --- |
| 1 | Coordonnees point = {0}; |

Cela mettra tous les éléments de la structure à 0.

Notre fonction initialiserCoordonnees va prendre un paramètre : un pointeur sur une structure de type Coordonnees (un Coordonnees\* donc ;) ).  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | int main(int argc, char \*argv[])  {  Coordonnees monPoint;  initialiserCoordonnees(&monPoint);  **return** 0;  }  void initialiserCoordonnees(Coordonnees\* point)  {  *// Initialisation de chacun des membres de la structure ici*  } |

Ma variable monPoint est donc créée dans le main. On envoie son adresse à initialiserCoordonnees qui récupère cette variable sous la forme d'un pointeur appelé "point" (on aurait d'ailleurs pu l'appeler n'importe comment dans la fonction, même "monPoint" ou "trucBidule" ;) ).  
  
  
Bien, et maintenant que nous sommes dans initialiserCoordonnees, nous allons initialiser chacune des valeurs une à une.  
Il ne faut pas oublier de mettre une étoile devant le nom du pointeur pour accéder à la variable. Si vous ne le faites pas, vous risquez de modifier l'adresse, et ce n'est pas ce que nous voulons faire ;)  
  
Oui mais voilà, problème... On ne peut pas vraiment faire :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5 | void initialiserCoordonnees(Coordonnees\* point)  {  \*point.x = 0;  \*point.y = 0;  } |

C'aurait été trop facile bien entendu :D  
  
Pourquoi on ne peut pas faire ça ? Parce que le point de séparation s'applique sur le mot "point" et non \*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 mettre des parenthèses autour de \*point. Comme cela, le point de séparation s'appliquera à \*point et non juste à point :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5 | void initialiserCoordonnees(Coordonnees\* point)  {  (\*point).x = 0;  (\*point).y = 0;  } |

Voilà :)  
  
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.

En 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".  
Nous n'en sommes pas encore à étudier le C++, mais vous verrez que, lorsque nous y arriverons, le C++ n'est en fait rien d'autre qu'une sorte de "super-amélioration" des structures. Bien entendu, beaucoup de choses découleront de cela, mais nous les étudierons en temps voulu. En attendant, je tiens à vous rappeler de vous concentrer sur le C, car tout ce que vous apprenez ici vous le réutiliserez avec le C++ ;)

**Un raccourci pratique et très utilisé**

Vous allez voir qu'on manipulera très souvent des pointeurs de structures. Pour être franc, je dois même vous dire qu'en C on utilise plus souvent des pointeurs de structures que des structures tout court :D  
Quand je vous disais que les pointeurs vous poursuivraient jusque dans votre tombe, je ne le disais pas en rigolant ^^ Vous ne pouvez y échapper, c'est votre destinée.  
  
Hum...  
En fait je voulais vous parler d'un truc sérieux là :-°  
Comme les pointeurs de structures sont très utilisés, on sera souvent amenés à écrire ceci :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | (\*point).x = 0; |

Oui mais voilà, encore une fois les programmeurs trouvent ça trop long. Les parenthèses autour de \*point, quelle plaie ! Alors, comme les programmeurs sont de grosses feignasses (comment ça je l'ai déjà dit ?! o_O ), ils ont inventé le raccourci suivant :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | point->x = 0; |

Ce raccourci consiste à former une flèche avec un tiret suivi d'un chevron ">".  
**Ecrire point->x est donc STRICTEMENT équivalent à écrire (\*point).x**  
Ca éclaircira votre code vous verrez ;)

N'oubliez pas qu'on ne peut utiliser la flèche que sur un pointeur !  
Si vous travaillez directement sur la variable, vous devez utiliser le symbole "point" comme on l'a vu au début.

Reprenons notre fonction initialiserCoordonnees. Nous pouvons donc l'écrire comme ceci :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5 | void initialiserCoordonnees(Coordonnees\* point)  {  point->x = 0;  point->y = 0;  } |

Retenez bien ce raccourci de la flèche, nous allons le réutiliser et pas qu'une seule fois ;)  
  
Et surtout, surtout, ne confondez pas la flèche avec le symbole "point".  
La flèche est réservée aux pointeurs, le "point" est réservé aux variables. Utilisez ce petit exemple pour vous en souvenir :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | int main(int argc, char \*argv[])  {  Coordonnees monPoint;  Coordonnees \*pointeur = &monPoint;  monPoint.x = 10; *// On travaille sur une variable, on utilise le "point"*  pointeur->x = 10; *// On travaille sur un pointeur, on utilise la flèche*  **return** 0;  } |

On met le x à 10 de deux manières différentes ici, la première fois en travaillant directement sur la variable, la seconde fois en passant par le pointeur.

**Les énumérations**

Les énumérations sont une façon un peu différente de créer ses propres types de variables.  
  
Une énumération ne contient pas de "sous-variables" comme c'était le cas pour les structures. C'est une liste de "valeurs possibles" pour une variable. Une énumération ne prend donc qu'une case en mémoire, et cette case peut prendre une des valeurs que vous définissez (et une seule à la fois).  
  
Voici un exemple d'énumération :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5 | **typedef** **enum** Volume Volume;  **enum** Volume  {  FAIBLE, MOYEN, FORT  }; |

Vous noterez qu'on utilise un typedef là aussi, comme on l'a fait jusqu'ici.  
  
Pour créer une énumération, on utilise le mot-clé enum. Notre énumération s'appelle ici "Volume". C'est un type de variable personnalisé qui peut prendre **une des 3 valeurs**qu'on a indiquées : soit FAIBLE, soit MOYEN, soit FORT.  
  
  
On va pouvoir créer une variable de type Volume, par exemple *musique*, qui stockera le volume actuel de la musique.  
On peut par exemple initialiser la musique au volume MOYEN :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | Volume musique = MOYEN; |

Voilà qui est fait :)  
Plus tard dans le programme, on pourra modifier la valeur du volume et la mettre soit à FAIBLE, soit à FORT.

**Association de nombres aux valeurs**

Vous avez remarqué que j'ai mis les valeurs possibles de l'énumération en majuscules. Ca devrait vous rappeler les constantes et les defines non ? ;)  
  
En effet, c'est assez similaire mais ce n'est pourtant pas exactement la même chose.  
Le compilateur associe automatiquement un nombre à chacune des valeurs possibles de l'énumération.  
  
Dans le cas de notre énumération Volume, FAIBLE vaut 0, MOYEN vaut 1 et FORT vaut 2. L'association est automatique et commence à 0.  
  
Contrairement au #define, c'est le compilateur qui associe MOYEN à 1 par exemple, et non le préprocesseur. Au bout du compte, ça revient un peu au même ;)  
Quand on a initialisé la variable *musique*à MOYEN, on a donc en fait mis la case en mémoire à la valeur 1.

En pratique, est-ce utile de savoir que MOYEN vaut 1, FORT vaut 2 etc. ?

Non. En général vous vous en moquez ;)  
C'est le compilateur qui associe automatiquement un nombre à chaque valeur. Grâce à ça, vous n'avez plus qu'à faire :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | **if** (musique == MOYEN)  {  *// Jouer la musique au volume moyen*  } |

Peu importe la valeur de MOYEN, vous laissez le compilateur se charger de gérer les nombres.  
  
L'intérêt de tout ça ? C'est que du coup votre code est très lisible. En effet, tout le monde peut facilement lire le if précédent (on comprend bien que la condition signifie "Si la musique est au volume moyen").

Notez qu'on aurait très bien pu faire ça sans utiliser d'énumération. On aurait par exemple pu créer une variable musique de type long, et "retenir" que 1 signifie "volume moyen" par exemple.  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | **if** (musique == 1)  {  } |

Mais comme vous le voyez dans l'exemple ci-dessus, c'est moins facile à lire pour le programmeur ;)

**Associer une valeur précise**

Pour le moment, c'est le compilateur qui décide d'associer le nombre 0 à la première valeur, puis 1, 2, 3 dans l'ordre.  
Il est possible de demander d'associer une valeur précise à chaque élément de l'énumération. Mais quel intérêt est-ce que ça peut bien avoir ? ^^  
  
Eh bien, supposons que sur votre ordinateur le volume soit géré entre 0 et 100 (0 = pas de son, 100 = 100% du son). Il est alors pratique d'associer une valeur précise à chaque élément :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5 | **typedef** **enum** Volume Volume;  **enum** Volume  {  FAIBLE = 10, MOYEN = 50, FORT = 100  }; |

Ici, le volume FAIBLE correspondra à 10% de volume, le volume MOYEN à 50% etc.  
On pourrait facilement ajouter de nouvelles valeurs possibles comme MUET. On mettrait dans ce cas MUET à la valeur... 0 ! Bravo vous avez compris le truc :D

Lire et écrire dans des fichiers

## Ouvrir et fermer un fichier

Pour lire et écrire dans des fichiers, nous allons nous servir de fonctions situées dans la librairie **stdio** que nous avons déjà utilisée.  
Oui, cette librairie-là contient aussi les fonctions printf et scanf que nous avons longuement utilisées jusqu'ici ! Mais elle ne contient pas que ça : il y a aussi d'autres fonctions, notamment des fonctions faites pour travailler sur des fichiers.

Toutes les librairies que je vous ai fait utiliser jusqu'ici (stdlib.h, stdio.h, math.h, string.h...) sont ce qu'on appelle **des librairies standard**. Ce sont des librairies automatiquement livrées avec votre IDE qui ont la particularité de fonctionner sur tous les OS. Vous pouvez donc les utiliser partout, que vous soyez sous Windows, Linux, Mac ou autre ;)  
Les librairies standard ne sont pas très nombreuses et ne permettent de faire que des choses très basiques, comme ce que nous avons fait jusqu'ici. Pour faire des choses plus avancées, comme ouvrir des fenêtres, il faudra télécharger et installer de nouvelles librairies. Nous verrons cela bientôt ;)

Assurez-vous donc, pour commencer, que vous incluez bien au moins les librairies stdio.h et stdlib.h en haut de votre fichier .c :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2 | #include <stdlib.h>  #include <stdio.h> |

Ces librairies sont tellement fondamentales, tellement basiques, que je vous recommande d'ailleurs de les inclure dans tous vos futurs programmes, quels qu'ils soient.  
  
  
Bien. Maintenant que les bonnes librairies sont incluses, nous allons pouvoir attaquer les choses sérieuses ^^  
Voici ce qu'il faut faire à chaque fois dans l'ordre quand on veut ouvrir un fichier (que ce soit pour lire ou pour écrire dedans) :

* On appelle la fonction d'**ouverture de fichier** *fopen*qui nous renvoie un pointeur sur le fichier.
* **On vérifie si l'ouverture a réussi** (c'est-à-dire si le fichier existait) en testant la valeur du pointeur qu'on a reçu. Si le pointeur vaut NULL, c'est que l'ouverture du fichier n'a pas marché, dans ce cas on ne peut pas continuer (il faut afficher un message d'erreur).
* Si l'ouverture a marché (si le pointeur est différent de NULL donc), alors on peut s'amuser à **lire et écrire dans le fichier** à travers des fonctions que nous verrons un peu plus loin.
* Une fois qu'on a **terminé de travailler sur le fichier**, il faut penser à le "fermer" avec la fonction *fclose*.

Nous allons dans un premier temps apprendre à nous servir de *fopen*et *fclose*. Une fois que vous saurez faire cela, nous apprendrons à lire le contenu du fichier et à écrire dedans.

### fopen : ouverture du fichier

Dans le chapitre sur les chaînes, nous nous sommes servis des prototypes des fonctions comme de "mode d'emploi". C'est comme ça que les programmeurs font en général : ils lisent le prototype et comprennent alors le fonctionnement de la fonction (bon, je reconnais que des fois même eux ont besoin de quelques petites explications à côté ;) )  
  
Voyons voir justement le prototype de la fonction *fopen*:  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | FILE\* fopen(**const** char\* nomDuFichier, **const** char\* modeOuverture); |

Cette fonction attend 2 paramètres :

* Le nom du fichier à ouvrir.
* Le mode d'ouverture du fichier, c'est-à-dire une indication qui dit si vous voulez juste écrire dans le fichier, juste lire dans le fichier, ou les deux à la fois ;)

Cette fonction renvoie... un pointeur sur FILE o_O  
  
Eh bien ça les amis, vous devriez comprendre ce que c'est maintenant ^^  
C'est un pointeur sur une structure de type FILE. Cette structure est définie dans stdio.h. Vous pouvez ouvrir ce fichier pour voir de quoi est constitué le type FILE, mais ça n'a aucun intérêt je vous le dis de suite ;)

Pourquoi le nom de la structure est-il tout en majuscules (FILE) ? Je croyais que les noms tout en majuscules étaient réservés aux constantes et aux defines ?

Cette "règle", c'est moi qui me la suis fixée (et pas mal d'autres programmeurs la suivent d'ailleurs). Ca n'a jamais été une obligation. Force est de croire que ceux qui ont programmé stdio ne suivaient pas exactement les mêmes règles ^^  
Cela ne doit pas vous perturber pour autant. Vous verrez d'ailleurs que les librairies que nous étudierons ensuite respectent les mêmes règles que moi, à savoir juste la première lettre en majuscule pour une structure.  
  
Revenons à notre fonction fopen. Elle renvoie un FILE\*. Il est extrêmement important de récupérer ce pointeur, pour pouvoir ensuite lire et écrire dans le fichier.  
Nous allons donc créer un pointeur de FILE au début de notre fonction (par exemple la fonction main) :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int main(int argc, char \*argv[])  {  FILE\* fichier = NULL;  **return** 0;  } |

Le pointeur est initialisé à NULL dès le début. Je vous rappelle que c'est une règle **fondamentale**que d'initialiser ses pointeurs à NULL dès le début si on n'a pas d'autre valeur à leur donner. Si vous ne le faites pas, vous risquez moultes plantages par la suite ;)

Vous noterez qu'il n'est pas nécessaire d'écrire struct FILE\* fichier = NULL. Les créateurs de stdio ont donc fait un typedef comme je vous ai appris à le faire il n'y a pas longtemps :)  
Notez que la forme de la structure peut changer d'un OS à l'autre (elle ne contient pas forcément les mêmes sous-variables partout). Pour cette raison, on ne modifiera jamais le contenu d'un FILE directement (on ne fera pas *fichier.truc* par exemple). On passera par des fonctions qui manipulent le FILE à notre place.

Maintenant, nous allons appeler la fonction *fopen*et récupérer la valeur qu'elle renvoie dans le pointeur "fichier". Mais avant ça, il faut que je vous explique comment se servir du second paramètre, le paramètre "modeOuverture". En effet, il y a un code à envoyer qui indiquera à l'ordinateur si vous ouvrez le fichier en mode de lecture seule, d'écriture seule, ou des deux à la fois.  
Voici les modes d'ouvertures possibles :

* **"r" : lecture seule**. Vous pourrez lire le contenu du fichier, mais pas écrire dedans. *Le fichier doit avoir été créé au préalable.*
* **"w" : écriture seule**. Vous pourrez écrire dans le fichier, mais pas lire son contenu. *Si le fichier n'existe pas, il sera créé.*
* **"a" : mode d'ajout**. Vous écrirez dans le fichier, en partant de la fin du fichier. Vous rajouterez donc du texte à la fin du fichier. *Si le fichier n'existe pas, il sera créé.*
* **"r+" : lecture et écriture**. Vous pourrez lire et écrire dans le fichier. *Le fichier doit avoir été créé au préalable.*
* **"w+" : lecture et écriture, avec suppression du contenu au préalable**. Le fichier est donc d'abord vidé de son contenu, et vous écrivez et lisez ensuite dedans. *Si le fichier n'existe pas, il sera créé.*
* **"a+" : ajout en lecture / écriture à la fin**. Vous écrivez et lisez du texte à partir de la fin du fichier. *Si le fichier n'existe pas, il sera créé.*

Et encore, je n'ai pas tout mis là ! Il y a le double de ça en réalité ! Pour chaque mode qu'on a vu là, si vous rajoutez un "b" après le premier caractère ("rb", "wb", "ab", "rb+", "wb+", "ab+"), alors le fichier est ouvert en mode binaire. C'est un mode un peu particulier que nous ne verrons pas ici. En fait, le mode texte est fait pour stocker... du texte comme le nom l'indique (uniquement des caractères affichables), tandis que le mode binaire permet de stocker... des informations octet par octet (des nombres principalement). C'est sensiblement différent. Vous utiliseriez par exemple le mode binaire pour lire et écrire des fichiers Word octet par octet.  
Le fonctionnement est quasiment le même de toute façon que ce que nous allons voir ici.  
  
On a déjà fort à faire avec ces **6 modes d'ouverture** à retenir ^^  
Personnellement, j'utilise souvent "r" (lecture), "w" (écriture) et "r+" (lecture et écriture). Le mode "w+" est un peu dangereux parce qu'il vide de suite le contenu du fichier, sans demande de confirmation. Il ne doit être utilisé que si vous voulez d'abord réinitialiser le fichier.  
Le mode d'ajout ("a") peut être utile dans certains cas, si vous voulez juste rajouter des informations à la fin du fichier.

Si vous avez juste l'intention de lire un fichier, il est conseillé de mettre "r". Certes, le mode "r+" aurait fonctionné lui aussi, mais en mettant "r" vous vous assurez que le fichier ne pourra pas être modifié, ce qui est en quelque sorte une sécurité.

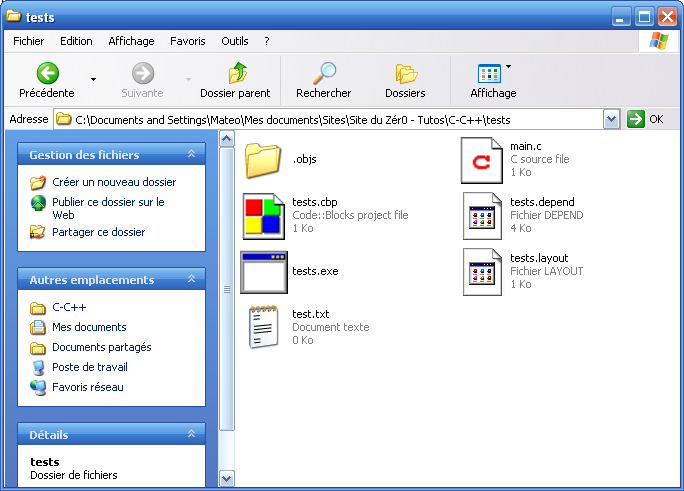
Si vous écrivez une fonction "chargerNiveau" (pour charger le niveau d'un jeu par exemple), le mode "r" suffit. Si vous écrivez une fonction "enregistrerNiveau", le mode "w" sera alors adapté.  
  
Le code suivant ouvre le fichier test.txt en mode "r+" (lecture / écriture) :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int main(int argc, char \*argv[])  {  FILE\* fichier = NULL;  fichier = fopen("test.txt", "r+");  **return** 0;  } |

Le pointeur "fichier" devient alors un pointeur sur "test.txt".

Où doit être situé test.txt ?

Il doit être situé dans le même dossier que votre exécutable (.exe).  
Pour les besoins de ce chapitre, créez un fichier "test.txt" comme moi dans le même dossier que le .exe :



Comme vous le voyez, je travaille actuellement avec l'IDE Code::Blocks, ce qui explique la présence d'un fichier de projet .cbp (au lieu de .dev si vous avez Dev-C++, ou .sln si vous avez Visual C++).  
Bref, ce qui compte c'est de bien voir que mon programme (tests.exe) est situé dans le même dossier que le fichier dans lequel on va lire et écrire (test.txt).

Le fichier doit-il être de type .txt ?

Non. C'est vous qui choisissez l'extension lorsque vous ouvrez le fichier. Vous pouvez très bien inventer votre propre format de fichier ".niveau" pour enregistrer les niveaux de vos jeux par exemple ;)

Le fichier doit-il être obligatoirement dans le même répertoire que l'exécutable ?

Non plus. Il peut être dans un sous-dossier :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | fichier = fopen("dossier/test.txt", "r+"); |

Ici, le fichier test.txt est dans un sous-dossier appelé "dossier". Cette méthode, que l'on appelle *chemin relatif* est plus pratique. Comme ça, cela fonctionnera peu importe l'endroit où est installé votre programme. C'est donc plus pratique.  
  
Il est aussi possible d'ouvrir un autre fichier n'importe où ailleurs sur le disque dur. Dans ce cas, il faut écrire le chemin complet (ce qu'on appelle le *chemin absolu*) :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | fichier = fopen("C:**\\**Program Files**\\**Notepad++**\\**readme.txt", "r+"); |

Ce code ouvre le fichier readme.txt situé dans "C:\Program Files\Notepad++"

J'ai dû mettre 2 antislash \ comme vous l'avez remarqué. En effet, si j'en avais mis un seul, votre ordinateur aurait cru que vous essayiez d'insérer un symbole spécial comme \n ou \t. Pour mettre un antislash dans une chaîne, il faut donc l'écrire 2 fois. Votre ordinateur comprend alors que c'est bien le symbole \ que vous vouliez utiliser.

Le défaut des chemins absolus, c'est qu'ils ne marchent que sur un OS précis. Ce n'est pas une solution *portable*donc.  
Si vous aviez été sous Linux, vous auriez dû écrire un chemin à-la-linux, tel que :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | fichier = fopen("/home/mateo/dossier/readme.txt", "r+"); |

Je vous recommande donc d'utiliser des chemins relatifs plutôt que des chemins absolus. N'utilisez les chemins absolus que si votre programme est fait pour un OS précis et doit modifier un fichier précis quelque part sur votre disque dur.

### Tester l'ouverture du fichier

Le pointeur "fichier" devrait contenir l'adresse de la structure de type FILE qui sert de descripteur de fichier. Celui-ci a été chargé en mémoire pour vous par la fonction fopen().  
A partir de là, 2 possibilités :

* Soit l'ouverture a réussi, et vous pouvez continuer (c'est-à-dire commencer à lire et écrire dans le fichier).
* Soit l'ouverture a échoué parce que le fichier n'existait pas ou était utilisé par un autre programme. Dans ce cas, vous devez arrêter de travailler sur le fichier.

Juste après l'ouverture du fichier, **il FAUT absolument vérifier si l'ouverture a réussi ou pas**. Pour faire ça, c'est très simple : si le pointeur vaut NULL, l'ouverture a échoué. S'il vaut autre chose que NULL, l'ouverture a réussi.  
On va donc suivre systématiquement le schéma suivant :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | int main(int argc, char \*argv[])  {  FILE\* fichier = NULL;  fichier = fopen("test.txt", "r+");  **if** (fichier != NULL)  {  *// On peut lire et écrire dans le fichier*  }  **else**  {  *// On affiche un message d'erreur si on veut*  printf("Impossible d'ouvrir le fichier test.txt");  }  **return** 0;  } |

Faites toujours ça lorsque vous ouvrez un fichier. Si vous ne le faites pas et que le fichier n'existe pas, vous risquez un plantage du programme ensuite.

### fclose : fermer le fichier

Si l'ouverture du fichier a réussi, vous pouvez lire et écrire dedans (nous allons voir de suite après comment faire).  
Une fois que vous aurez fini de travailler avec le fichier, il faudra le "fermer". On utilise pour cela la fonction fclose qui a pour rôle de libérer la mémoire, c'est-à-dire supprimer votre fichier chargé dans la mémoire vive.  
  
Son prototype est :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | int fclose(FILE\* pointeurSurFichier); |

Cette fonction prend un paramètre : votre pointeur sur le fichier.  
Elle renvoie un int qui indique si elle a réussi à fermer le fichier. Cet int vaut :

* 0 : si la fermeture a marché
* EOF : si la fermeture a échoué. EOF est un define situé dans stdio.h qui correspond à un nombre spécial, utilisé pour dire soit qu'il y a eu une erreur, soit qu'on est arrivés à la fin du fichier. Dans le cas présent cela signifie qu'il y a eu une erreur.

A priori, la fermeture se passe toujours bien donc je n'ai pas l'habitude de tester si le fclose a marché. Vous pouvez le faire néanmoins si vous le voulez.  
  
Pour fermer le fichier, on va donc écrire :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | fclose(fichier); |

Tout simplement :)  
  
Au final, le schéma que nous allons suivre pour ouvrir et fermer un fichier sera le suivant :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | int main(int argc, char \*argv[])  {  FILE\* fichier = NULL;  fichier = fopen("test.txt", "r+");  **if** (fichier != NULL)  {  *// On lit et on écrit dans le fichier*    *// ...*    fclose(fichier); *// On ferme le fichier qui a été ouvert*  }  **return** 0;  } |

Je n'ai pas mis le else ici pour afficher un message d'erreur si l'ouverture a échoué, mais vous pouvez le faire si vous le désirez.  
  
Il faut toujours penser à fermer son fichier une fois que l'on a fini de travailler avec. Cela permet de libérer de la mémoire.  
Si vous oubliez de libérer la mémoire, votre programme risque à la fin de prendre énormément de mémoire qu'il n'utilise plus. Sur un petit exemple comme ça ce n'est pas flagrant, mais sur un gros programme, bonjour les dégâts :p  
  
Oublier de libérer la mémoire, ça arrive. Ca vous arrivera d'ailleurs très certainement. Dans ce cas, vous serez témoins de ce que l'on appelle des **fuites mémoire**. Votre programme se mettra alors à utiliser plus de mémoire que nécessaire sans que vous arriviez à comprendre pourquoi. Bien souvent, il s'agit simplement d'un ou deux trucs comme des petits *fclose*oubliés. Comme quoi la solution à un problème mystique est parfois toute bête (je dis bien "parfois" :D )

## Différentes méthodes de lecture / écriture

Maintenant que nous avons écrit le code qui ouvre et ferme le fichier, nous n'avons plus qu'à insérer le code qui lit et écrit dedans ^^  
  
Nous allons commencer par voir comment écrire dans un fichier (ce qui est un peu plus simple), puis nous verrons ensuite comment lire dans un fichier.

### Ecrire dans le fichier

Il existe plusieurs fonctions capables d'écrire dans un fichier. Ce sera à vous de choisir celle qui est la plus adaptée à votre cas.  
Voici les 3 fonctions que nous allons étudier :

* fputc : écrit un caractère dans le fichier (UN SEUL caractère à la fois).
* fputs : écrit une chaîne dans le fichier
* fprintf : écrit une chaîne "formatée" dans le fichier, fonctionnement quasi-identique à printf

#### fputc

Cette fonction écrit un caractère à la fois dans le fichier. Son prototype est :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | int fputc(int caractere, FILE\* pointeurSurFichier); |

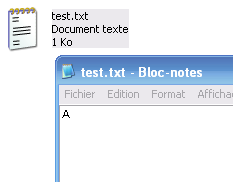
Elle prend 2 paramètres :

* Le caractère à écrire (de type int, ce qui comme je vous l'ai dit revient plus ou moins à utiliser un char, sauf que le nombre de caractères utilisables est ici plus grand). Vous pouvez donc écrire directement 'A' par exemple.
* Le pointeur sur le fichier dans lequel écrire. Dans notre exemple, notre pointeur s'appelle "fichier". L'avantage de demander le pointeur de fichier à chaque fois, c'est que vous pouvez ouvrir plusieurs fichiers en même temps et donc lire et écrire dans chacun de ces fichiers. Vous n'êtes pas limités à un seul fichier ouvert à la fois.

La fonction retourne un int, c'est un code d'erreur. Cet int vaut EOF si l'écriture a échoué, sinon il vaut autre chose.  
Comme le fichier a été ouvert avec succès normalement, je n'ai pas l'habitude de tester si chacun de mes fputc a réussi, mais vous pouvez le faire encore une fois si vous le voulez.  
  
Le code suivant écrit la lettre 'A' dans test.txt (si le fichier existe, il est remplacé ; si il n'existe pas, il est créé). Il y a tout dans ce code : ouverture, test de l'ouverture, écriture et fermeture.  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | int main(int argc, char \*argv[])  {  FILE\* fichier = NULL;    fichier = fopen("test.txt", "w");    **if** (fichier != NULL)  {  fputc('A', fichier); *// Ecriture du caractère A*  fclose(fichier);  }    **return** 0;  } |

Ouvrez votre fichier "test.txt". Que voyez-vous ?  
C'est magique (enfin pas tellement :p ), le fichier contient maintenant la lettre 'A' !  
  
La preuve en image :



#### fputs

Cette fonction est très similaire à fputc, à la différence près qu'elle écrit tout une chaîne, ce qui est en général plus pratique que d'écrire caractère par caractère ^^  
Ceci dit, fputc reste utile lorsque vous devez écrire caractère par caractère, ce qui arrive fréquemment :)  
  
Prototype de la fonction :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | char\* fputs(**const** char\* chaine, FILE\* pointeurSurFichier); |

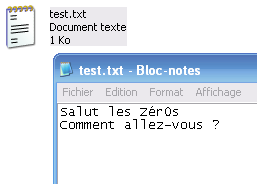
Les 2 paramètres sont faciles à comprendre :

* chaine : la chaîne à écrire. Notez que le type ici est **const char\*** : en rajoutant le mot const dans le prototype, la fonction indique que pour elle la chaîne sera considérée comme une constante. En 1 mot comme en 100 : elle s'interdit de modifier le contenu de votre chaîne. C'est logique quand on y pense : fputs doit juste lire votre chaîne, pas la modifier. C'est donc pour vous une information (et une sécurité) comme quoi votre chaîne ne subira pas de modification
* pointeurSurFichier : comme pour fputc, il s'agit de votre pointeur de type FILE\* sur le fichier que vous avez ouvert.

La fonction renvoie EOF s'il y a eu une erreur, sinon c'est que cela a fonctionné. Là non plus, je ne teste en général pas la valeur de retour.  
  
Testons l'écriture d'une chaîne dans le fichier :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | int main(int argc, char \*argv[])  {  FILE\* fichier = NULL;    fichier = fopen("test.txt", "w");    **if** (fichier != NULL)  {  fputs("Salut les Zér0s**\n**Comment allez-vous ?", fichier);  fclose(fichier);  }    **return** 0;  } |

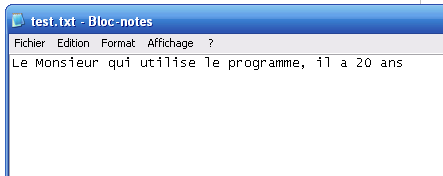
Preuve que ce code fonctionne :



#### fprintf

Voici un autre exemplaire de la fonction printf. Celle-ci peut être utilisée pour écrire dans un fichier. Elle s'utilise de la même manière que printf d'ailleurs, excepté le fait que vous devez indiquer un pointeur de FILE en premier paramètre.  
  
Ce code demande l'âge de l'utilisateur et l'écrit dans le fichier :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | int main(int argc, char \*argv[])  {  FILE\* fichier = NULL;  long age = 0;    fichier = fopen("test.txt", "w");    **if** (fichier != NULL)  {  *// On demande l'âge*  printf("Quel age avez-vous ? ");  scanf("%ld", &age);    *// On l'écrit dans le fichier*  fprintf(fichier, "Le Monsieur qui utilise le programme, il a %ld ans", age);  fclose(fichier);  }    **return** 0;  } |



Vous pouvez ainsi facilement réutiliser ce que vous savez de printf pour écrire dans un fichier, c'est pas génial ça ? :)  
C'est pour cette raison d'ailleurs que j'utilise le plus souvent fprintf pour écrire dans des fichiers, car c'est pratique et pas trop compliqué :)

### Lire dans un fichier

Nous pouvons utiliser quasiment les mêmes fonctions que pour l'écriture, le nom change juste un petit peu :

* fgetc : lit un caractère
* fgets : lit une chaîne
* fscanf : lit une chaîne formatée

Je vais aller un peu plus vite cette fois dans l'explication de ces fonctions, si vous avez compris ce que j'ai écrit plus haut ça ne devrait pas poser de problème ;)

#### fgetc

Tout d'abord le prototype :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | int fgetc(FILE\* pointeurDeFichier); |

Cette fonction retourne un int : c'est le caractère qui a été lu.  
Si la fonction n'a pas pu lire de caractère, elle retourne EOF.

Mais comment savoir quel caractère on lit ? Si on veut lire le 3ème caractère, ainsi que le 10ème caractère, comment doit-on faire ?

En fait, au fur et à mesure que vous lisez un fichier, vous avez un "curseur" qui avance. C'est un curseur virtuel hein, vous ne le voyez pas à l'écran ;)  
Mais vous pouvez imaginer que ce curseur est comme la barre clignotante lorsque vous éditez un fichier sous Bloc-Notes. Il indique où vous en êtes dans la lecture du fichier.  
Nous verrons peu après comment savoir à quelle position le curseur est situé dans le fichier, et aussi comment modifier la position du curseur (pour le remettre au début du fichier par exemple, ou le placer à un caractère précis, comme le 10ème caractère).  
  
fgetc avance le curseur d'un caractère à chaque fois que vous en lisez un. Si vous appelez fgetc une seconde fois, la fonction lira donc le second caractère, puis le troisième et ainsi de suite :)  
Vous pouvez faire une boucle pour lire les caractères un par un dans le fichier donc :)  
  
On va écrire un code qui lit tous les caractères d'un fichier un à un, et qui les écrit à chaque fois à l'écran.  
La boucle s'arrête quand fgetc renvoie EOF (qui signifie End Of File, c'est-à-dire "fin du fichier").  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | int main(int argc, char \*argv[])  {  FILE\* fichier = NULL;  int caractereActuel = 0;    fichier = fopen("test.txt", "r");    **if** (fichier != NULL)  {  *// Boucle de lecture des caractères un à un*  **do**  {  caractereActuel = fgetc(fichier); *// On lit le caractère*  printf("%c", caractereActuel); *// On l'affiche*  } **while** (caractereActuel != EOF); *// On continue tant que fgetc n'a pas retourné EOF (fin de fichier)*    fclose(fichier);  }    **return** 0;  } |

La console affichera tout le contenu du fichier, par exemple :  
  
**Code : Console**

|  |
| --- |
| Coucou, je suis le contenu du fichier test.txt ! |

#### fgets

Cette fonction lit une chaîne dans le fichier. Ca vous évite d'avoir à lire tous les caractères un par un. La fonction **lit au maximum une ligne** (elle s'arrête au premier \n qu'elle rencontre). Si vous voulez lire plusieurs lignes, il faudra faire une boucle.  
  
Voici le prototype de fgets :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | char\* fgets(char\* chaine, int nombreDeCaracteresALire, FILE\* pointeurSurFichier); |

Cette fonction demande un paramètre un peu particulier, qui va en fait s'avérer très pratique : le nombre de caractères à lire. Cela demande à la fonction fgets de s'arrêter de lire la ligne si elle contient plus de X caractères.  
Avantage : ça nous permet de nous assurer que l'on ne fera pas de dépassement de mémoire ! En effet, si la ligne est trop grosse pour rentrer dans chaine, la fonction aurait lu plus de caractères qu'il n'y a de place, ce qui aurait probablement provoqué un plantage du programme.  
  
**Lire une ligne avec fgets**  
  
Nous allons d'abord voir comment lire une ligne avec fgets (nous verrons ensuite comment lire tout le fichier).  
  
On crée une chaîne suffisamment grande pour stocker le contenu de la ligne qu'on va lire (du moins on espère ^^ ). Vous allez voir là tout l'intérêt d'utiliser un define pour définir la taille du tableau :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #define TAILLE\_MAX 1000 *// Tableau de taille 1000*    int main(int argc, char \*argv[])  {  FILE\* fichier = NULL;  char chaine[TAILLE\_MAX] = ""; *// Chaîne vide de taille TAILLE\_MAX*    fichier = fopen("test.txt", "r");    **if** (fichier != NULL)  {  fgets(chaine, TAILLE\_MAX, fichier); *// On lit maximum TAILLE\_MAX caractères du fichier, on stocke le tout dans "chaine"*  printf("%s", chaine); *// On affiche la chaîne*    fclose(fichier);  }    **return** 0;  } |

Le résultat est le même que pour le code de tout à l'heure, à savoir que le contenu s'écrit dans la console :  
  
**Code : Console**

|  |
| --- |
| Coucou, je suis le contenu du fichier test.txt ! |

La différence, c'est qu'ici on ne fait pas de boucle. On affiche toute la chaîne d'un coup.

Vous aurez sûrement remarqué maintenant l'intérêt que peut avoir un #define dans son code pour définir la taille maximale d'un tableau par exemple. En effet, TAILLE\_MAX est ici utilisé à 2 endroits du code :

* Une première fois pour définir la taille du tableau à créer
* Une autre fois dans le fgets pour limiter le nombre de caractères à lire.

L'avantage ici, c'est que si vous vous rendez compte que la chaîne n'est pas assez grande pour lire le fichier, vous n'avez qu'à changer la ligne du define et recompiler :) Cela vous évite d'avoir à chercher tous les endroits du code qui indiquent la taille du tableau. Le préprocesseur remplacera tous les TAILLE\_MAX dans le code par la nouvelle valeur :)

**Lire tout le fichier avec fgets**  
  
Comme je vous l'ai dit, fgets lit au maximum toute une ligne à la fois. Elle s'arrête de lire la ligne si elle dépasse le nombre de caractères maximum que vous autorisez.  
  
Oui mais voilà, pour le moment on ne sait lire qu'une seule ligne à la fois avec fgets. Comment diable lire tout le fichier ?  
La réponse est simple : avec une boucle :)  
  
La fonction fgets renvoie NULL si elle n'est pas parvenue à lire ce que vous avez demandé.  
La boucle doit donc s'arrêter dès que fgets se met à renvoyer NULL.  
  
  
On n'a plus qu'à faire un while pour boucler tant que fgets ne renvoit pas NULL, et zou !  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | #define TAILLE\_MAX 1000    int main(int argc, char \*argv[])  {  FILE\* fichier = NULL;  char chaine[TAILLE\_MAX] = "";    fichier = fopen("test.txt", "r");    **if** (fichier != NULL)  {  **while** (fgets(chaine, TAILLE\_MAX, fichier) != NULL) *// On lit le fichier tant qu'on ne reçoit pas d'erreur (NULL)*  {  printf("%s", chaine); *// On affiche la chaîne qu'on vient de lire*  }    fclose(fichier);  }    **return** 0;  } |

Ce code source lit et affiche tout le contenu de mon fichier, ligne par ligne.  
  
La ligne de code la plus intéressante c'est celle du while (en fait c'est la seule nouvelle chose par rapport à tout à l'heure :D ) :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | **while** (fgets(chaine, TAILLE\_MAX, fichier) != NULL) |

La ligne du while fait 2 choses : elle lit une ligne dans le fichier et vérifie si fgets ne renvoie pas NULL. Elle peut donc se traduire comme ceci : "*Lire une ligne du fichier tant qu'on****n'est pas****arrivés à la fin du fichier*".

#### fscanf

C'est le même principe que la fonction scanf là encore.  
Cette fonction lit dans un fichier qui doit avoir été écrit d'une manière précise.  
  
Supposons que votre fichier contienne 3 nombres séparés par un espace, qui sont par exemple les 3 plus hauts scores obtenus à votre jeu :  
  
15 20 30  
  
Vous voudriez récupérer chacun de ces nombres dans une variable de type long.  
La fonction fscanf va vous permettre de faire ça rapidement.  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | int main(int argc, char \*argv[])  {  FILE\* fichier = NULL;  long score[3] = {0}; *// Tableau des 3 meilleurs scores*    fichier = fopen("test.txt", "r");    **if** (fichier != NULL)  {  fscanf(fichier, "%ld %ld %ld", &score[0], &score[1], &score[2]);  printf("Les meilleurs scores sont : %ld, %ld et %ld", score[0], score[1], score[2]);    fclose(fichier);  }    **return** 0;  } |

**Code : Console**

|  |
| --- |
| Les meilleurs scores sont : 15, 20 et 30 |

Comme vous le voyez, la fonction fscanf attend 3 nombres séparés par un espace ("%ld %ld %ld"). Elle les stocke ici dans notre tableau de 3 blocs.  
  
On affiche ensuite chacun des nombres récupérés.

Jusqu'ici, je ne vous avais fait mettre qu'un seul "%ld" entre guillemets pour la fonction scanf. Vous découvrez aujourd'hui qu'on peut en mettre plusieurs, les combiner. Si votre fichier est écrit d'une façon bien précise, cela permet d'aller plus vite pour récupérer chacune des valeurs :)

## Se déplacer dans un fichier

Je vous ai parlé d'une espèce de "curseur" virtuel tout à l'heure. Nous allons l'étudier maintenant plus en détail.  
A chaque fois que vous ouvrez un fichier, il existe en effet un curseur qui indique votre position dans le fichier. Vous pouvez imaginer que c'est exactement comme le curseur de votre éditeur de texte (tel bloc-notes). Il indique où vous êtes dans le fichier, et donc où vous allez écrire.  
  
**En résumé** : le système de curseur vous permet d'aller lire et écrire à une position précise dans le fichier.  
  
Il existe 3 fonctions à connaître :

* ftell : indique à quelle position vous êtes actuellement dans le fichier
* fseek : positionne le curseur à un endroit précis
* rewind : remet le curseur au début du fichier (c'est équivalent à demander à la fonction fseek de positionner le curseur au début).

### ftell : position dans le fichier

Cette fonction est très simple à utiliser. Elle renvoie la position actuelle du curseur sous la forme d'un *long*:  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | long ftell(FILE\* pointeurSurFichier); |

Le nombre renvoyé indique donc la position du curseur dans le fichier.

### fseek : se positionner dans le fichier

Le prototype de fseek est le suivant :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | int fseek(FILE\* pointeurSurFichier, long deplacement, int origine); |

La fonction fseek permet de déplacer le "curseur" d'un certain nombre de caractères (indiqué par *deplacement*) à partir de la position indiquée par *origine*.

* Le nombre *deplacement* peut être un nombre positif (pour se déplacer en avant), nul (= 0) ou négatif (pour se déplacer en arrière).
* Quant au nombre *origine*, vous pouvez mettre comme valeur l'une des 3 constantes (généralement des defines) listées ci-dessous :
  + **SEEK\_SET** : indique le début du fichier.
  + **SEEK\_CUR** : indique la position actuelle du curseur.
  + **SEEK\_END** : indique la fin du fichier.

Voici quelques exemples pour bien comprendre comment on jongle avec *deplacement*et *origine*

* Le code suivant place le curseur 2 caractères *après*le début :  
    
  **Code : C**

|  |  |
| --- | --- |
| 1 | fseek(fichier, 2, SEEK\_SET); |

* Le code suivant place le curseur 4 caractères *avant*la position courante :  
    
  **Code : C**

|  |  |
| --- | --- |
| 1 | fseek(fichier, -4, SEEK\_CUR); |

* *(remarquez que deplacement est négatif car on se déplace en arrière)*
* Le code suivant place le curseur à la fin du fichier :  
    
  **Code : C**

|  |  |
| --- | --- |
| 1 | fseek(fichier, 0, SEEK\_END); |

Si vous écrivez après avoir fait un fseek qui mène à la fin du fichier, cela rajoutera vos informations à la suite dans le fichier (ça complètera votre fichier).  
En revanche, si vous placez le curseur au début et que vous écrivez, cela écrasera le texte qui se trouvait là. Il n'y a pas de moyen d'"insérer" de texte dans le fichier (à moins de coder soi-même une fonction qui lit les caractères d'après pour s'en souvenir avant de les écraser !).

Mais comment je sais à quelle position je dois aller lire et écrire dans le fichier ?

Alors ça, c'est vous qui gérez ;)  
Si c'est un fichier que vous avez vous-même écrit, vous savez comment il est construit. Vous savez donc où aller chercher vos informations (par exemple les meilleurs scores sont en position 0, les noms des derniers joueurs sont en position 50 etc...)  
  
On fera un TP un peu plus tard dans lequel vous comprendrez (si ce n'est pas déjà le cas :p ) comment on fait pour aller chercher l'information qui nous intéresse.  
N'oubliez pas que c'est **vous** qui définissez comment votre fichier est construit. C'est donc à vous de dire : "je place le score du meilleur joueur sur la première ligne, celui du second meilleur joueur sur la seconde ligne" etc...

La fonction fseek peut se comporter bizarrement sur des fichiers ouverts en mode texte. En général, on l'utilise plutôt pour se déplacer dans des fichiers ouverts en mode binaire.  
Quand on lit / écrit dans un fichier en mode texte, on le fait généralement caractère par caractère. La seule chose qu'on se permet en mode texte avec fseek c'est de revenir au début ou de se placer à la fin.  
  
En résumé : fseek c'est bien, mais à utiliser plutôt avec des fichiers binaires. On ne se déplace en général pas avec fseek sur des fichiers texte.

### rewind : retour au début

Cette fonction est équivalente à utiliser fseek pour nous renvoyer à la position 0 dans le fichier. Si vous avez eu un magnétoscope un jour dans votre vie, eh bien c'est le même nom que la touche qui permet de revenir en arrière.  
  
Le prototype est tout bête :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | void rewind(FILE\* pointeurSurFichier); |

L'utilisation est aussi bête que le prototype. Pour la peine, je ne vous donne pas d'exemple cette fois (je commence à avoir les doigts ankylosés à ce moment de l'écriture du chapitre :lol: )

## Renommer et supprimer un fichier

Nous terminerons ce chapitre en douceur par l'étude de 2 fonctions très simples :

* rename : renomme un fichier
* remove : supprime un fichier

La particularité de ces fonctions est **qu'elles ne nécessitent pas de pointeur de fichier pour fonctionner**. Il suffira juste d'indiquer le nom du fichier à renommer / supprimer :)  
Bref, c'est encore plus simple que simple :D

### rename : renommer un fichier

Ca va être vite étudié vous allez voir ;)  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | int rename(**const** char\* ancienNom, **const** char\* nouveauNom); |

La fonction renvoie 0 si elle a réussi à renommer, sinon elle renvoie... autre chose que 0 :p  
  
Avez-vous vraiment besoin d'un exemple ?  
Bon allez ^^  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int main(int argc, char \*argv[])  {  rename("test.txt", "test\_renomme.txt");  **return** 0;  } |

Ouahouh c'était duuuur :-°  
Et voilà mon fichier renommé ;)

Image utilisateur

### remove : supprimer un fichier

Cette fonction supprime un fichier sans demander son reste :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | int remove(**const** char\* fichierASupprimer); |

Faites très attention en utilisant cette fonction ! Elle supprime le fichier indiqué sans demander de confirmation ! Le fichier n'est pas mis dans la corbeille ni rien, il est littéralement supprimé du disque dur. Il n'est pas possible de récupérer un fichier supprimé.

Cette fonction tombe à pic pour la fin du chapitre, je n'ai justement plus besoin du fichier test.txt, je vais donc le supprimer ;)  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int main(int argc, char \*argv[])  {  remove("test.txt");  **return** 0;  } |

L'allocation dynamique

**La taille des variables**

Selon le type de variable que vous demandez à créer (char, int, double, float...), vous avez besoin de plus ou moins de mémoire.  
  
En effet, pour stocker un nombre compris entre -128 et 127 (un char), on n'a besoin que d'un octet en mémoire (c'est tout petit ^^ ).  
En revanche, un int occupe généralement 4 octets en mémoire. Quant au double, il occupe 8 octets.  
  
Le problème est... que ce n'est pas toujours le cas. Cela dépend de votre ordinateur : peut-être que chez vous un int occupe 8 octets, qui sait ?  
Notre objectif ici est de vérifier quelle taille occupe chacun des types sur votre ordinateur.  
  
Il y a un moyen très facile pour savoir cela : utiliser l'opérateur sizeof().  
Contrairement aux apparences, ce n'est pas une fonction mais une fonctionnalité *de base* du langage C. Vous devez juste indiquer entre parenthèses le type que vous voulez analyser.  
Pour connaître la taille d'un int, on devra donc écrire :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | **sizeof**(int) |

A la compilation, cela sera remplacé par un nombre : le nombre d'octets que prend int en mémoire. Chez moi, sizeof(int) vaut 4, ce qui signifie que int occupe 4 octets. Chez vous, c'est probablement la même valeur, mais ce n'est pas une règle. Testez, vous verrez ;)  
Vous pouvez faire des printf pour afficher cela :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | printf("char : %ld octets**\n**", **sizeof**(char));  printf("int : %ld octets**\n**", **sizeof**(int));  printf("long : %ld octets**\n**", **sizeof**(long));  printf("double : %ld octets**\n**", **sizeof**(double)); |

Chez moi, cela affiche :  
  
**Code : Console**

|  |
| --- |
| char : 1 octets  int : 4 octets  long : 4 octets  double : 8 octets |

Je n'ai pas mis tous les types que nous connaissons, je vous laisse le soin de tester vous-même la taille des autres types ;)  
Vous remarquerez que long et int occupent la même place en mémoire. Créer un long revient donc ici exactement à créer un int, cela prend 4 octets dans la mémoire.

En fait, le type "long" est équivalent à un type appelé "long int", qui est ici équivalent au type... "int". Bref, ça fait beaucoup de noms différents pour pas grand-chose au final ;) Avoir de nombreux types différents était utile à une époque où on n'avait pas beaucoup de mémoire. On cherchait à utiliser le minimum de mémoire possible en utilisant le type le plus adapté.  
Aujourd'hui, cela ne sert plus vraiment car la mémoire d'un ordinateur est très grande.

Peut-on afficher la taille d'un type personnalisé qu'on a créé (une structure) ?

Oui ! sizeof marche aussi sur les structures !  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **typedef** **struct** Coordonnees Coordonnees;  **struct** Coordonnees  {  long x;  long y;  };  int main(int argc, char \*argv[])  {  printf("Coordonnees : %ld octets**\n**", **sizeof**(Coordonnees));  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Coordonnees : 8 octets |

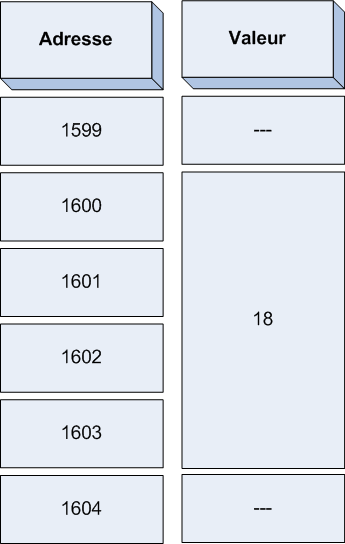
Plus une structure contient de sous-variables, plus elle prend de mémoire. Terriblement logique n'est-ce pas ? :D

**Une nouvelle façon de voir la mémoire**

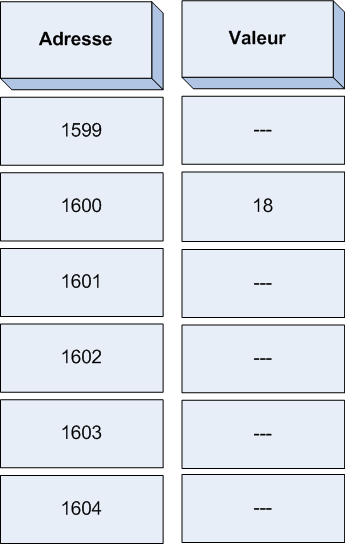
Jusqu'ici, mes schémas de mémoire étaient encore assez imprécis. On va enfin pouvoir les rendre précis et corrects maintenant qu'on connaît la taille de chacun des types de variables (c'est pas trop tôt ^^ )  
  
Si on déclare une variable de type long :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | long nombre = 18; |

... et que sizeof(long) indique 4 octets sur notre ordinateur, alors la variable occupera 4 octets en mémoire !  
  
Supposons que la variable nombre soit allouée à l'adresse 1600 en mémoire. On aurait alors le schéma suivant :



Ici, on voit bien que notre variable "nombre" de type long qui vaut 18 occupe 4 octets dans la mémoire.  
Elle commence à l'adresse 1600 (c'est son adresse) et termine à l'adresse 1603. La prochaine variable ne pourra donc être stockée qu'à partir de l'adresse 1604 !  
  
Si on avait fait la même chose avec un char, alors on n'aurait occupé qu'un seul octet en mémoire :



Imaginez maintenant un tableau de long !  
Chaque "case" du tableau occupera 4 octets. Si notre tableau fait 100 cases :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | long tableau[100]; |

Alors on occupera en réalité 4 \* 100 = 400 octets en mémoire :)

Même si le tableau est vide il prend 400 octets ?

Bien sûr ! La place en mémoire est réservée, aucun autre programme n'a le droit d'y toucher (à part le vôtre). Une fois qu'une variable est déclarée, elle prend immédiatement de la place en mémoire.  
  
  
Notez que si on crée un tableau de type "Coordonnees" :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | Coordonnees tableau[100]; |

... on utilisera (allez c'est facile :D ) : 8 \* 100 = 800 octets en mémoire.  
  
Il est important de bien comprendre ces petits calculs pour la suite du chapitre.  
C'est de la multiplication de niveau Primaire ça :p

**Allocation de mémoire dynamique**

Rentrons maintenant dans le vif du sujet.  
Le but du chapitre, c'était quoi justement ? :-°   
  
Ah oui : apprendre à demander de la mémoire manuellement.  
On va avoir besoin d'inclure la librairie <stdlib.h> (si vous avez suivi mes conseils, vous devriez avoir inclus cette librairie dans tous vos programmes de toute façon ;) ).  
  
Cette librairie contient 2 fonctions dont nous allons avoir besoin :

* **malloc**(*"Memory ALLOCation", c'est-à-dire "Allocation de mémoire"*) : demande au système d'exploitation la permission d'utiliser de la mémoire.
* **free**(*"Libérer"*) : 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.

Quand vous faites une allocation manuelle de mémoire (ce qu'on va apprendre à faire maintenant), vous devez toujours suivre ces 3 étapes :

1. Appeler **malloc**pour demander de la mémoire
2. **Vérifier la valeur retournée par malloc** pour savoir si l'OS a bien réussi à allouer la mémoire.
3. Une fois qu'on a fini d'utiliser la mémoire, on doit la libérer avec **free**. Si on ne le fait pas, on s'expose à des fuites de mémoire, c'est-à-dire que votre programme risque au final de prendre beaucoup de mémoire alors qu'il n'a en réalité plus besoin de tout cet espace.

Tiens, ces 3 étapes ça vous rappelle pas le chapitre sur les fichiers ça ? :D  
Ben moi si :p

Le principe est exactement le même qu'avec les fichiers : on alloue, on vérifie si l'allocation a marché, on utilise la mémoire, puis on libère quand on a fini d'utiliser.

Nous allons maintenant étudier la fonction malloc.

**malloc : demande d'allocation de mémoire**

Le prototype de la fonction malloc est assez comique vous allez voir :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | void\* malloc(size\_t nombreOctetsNecessaires); |

La fonction prend un paramètre : le nombre d'octets à réserver. Ainsi, il suffira d'écrire sizeof(long) dans ce paramètre pour réserver suffisamment d'espace pour stocker un long.  
  
Mais c'est surtout ce que la fonction renvoie qui est curieux : elle renvoie un... void\* :-°  
Si vous vous souvenez du chapitre sur les fonctions, je vous avais dit que "void" signifiait "vide" et qu'on utilisait ce type pour indiquer que la fonction ne retournait aucune valeur.  
Alors ici, on aurait une fonction qui retourne un "pointeur sur vide" ? o_O o_O o_O  
  
En voilà une bien bonne !  
Ces programmeurs ont décidemment un sens de l'humour très développé :lol:

Ca te dérangerait pas trop de nous donner quelques explications ?

Oui oui j'y viens ^^ Je me rappelle juste la première fois que j'ai vu le prototype de malloc, je suis resté la bouche ouverte un petit moment devant mon écran avant de comprendre :p  
  
En fait, cette fonction renvoie un pointeur indiquant l'adresse que l'OS a réservé pour votre variable. Si l'OS a trouvé de la place pour vous à l'adresse 1600, la fonction renvoie donc un pointeur contenant l'adresse 1600.  
Le problème, c'est que la fonction malloc ne sait pas quel type de variable vous cherchez à créer. En effet, vous ne lui donnez qu'un paramètre : le nombre d'octets en mémoire dont vous avez besoin. Si vous demandez 4 octets, ça pourrait aussi bien être un int qu'un long par exemple !  
  
Comme malloc ne sait pas quel type elle doit retourner, elle renvoie le type void\*. **Ce sera un pointeur sur n'importe quel type**. On peut dire que c'est un pointeur universel.  
  
  
Passons à la pratique. Si je veux m'amuser (*hahem*) à créer manuellement une variable de type long en mémoire, je devrai indiquer à malloc que j'ai besoin de sizeof(long) octets en mémoire.  
Je récupère le résultat du malloc dans un pointeur sur long.  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3 | long\* memoireAllouee = NULL; *// On crée un pointeur sur long*  memoireAllouee = malloc(**sizeof**(long)); *// La fonction malloc inscrit dans notre pointeur l'adresse qui a été reservee.* |

A la fin de ce code, memoireAllouee est un pointeur contenant une adresse qui vous a été réservée par l'OS, par exemple l'adresse 1600 (pour reprendre mes schémas de tout à l'heure).

**Tester le pointeur**

La fonction malloc a donc renvoyé dans notre pointeur memoireAllouee l'adresse qui a été réservée pour vous en mémoire.  
2 possibilités :

* Si l'allocation a marché, notre pointeur contient une adresse.
* Si l'allocation a échoué, notre pointeur contient l'adresse NULL.

Il est peu probable qu'une allocation échoue, mais cela peut arriver. Imaginez que vous demandiez à utiliser 34 Go de mémoire vive, il y a très peu de chances que l'OS vous réponde favorablement :lol:  
  
Il est néanmoins recommandé de **tester si l'allocation a marché**. On va faire ceci : si l'allocation a échoué, c'est qu'il n'y avait plus de mémoire de libre (c'est un cas critique). Dans un tel cas, le mieux est d'arrêter immédiatement le programme parce qu'il ne pourra pas continuer convenablement de toute manière.  
  
On va utiliser une fonction standard qu'on n'avait pas encore vue jusqu'ici : exit(). Elle arrête immédiatement le programme. Elle prend un paramètre : la valeur que le programme doit retourner (ça correspond au return du main()).  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | int main(int argc, char \*argv[])  {  long\* memoireAllouee = NULL;  memoireAllouee = malloc(**sizeof**(long));  **if** (memoireAllouee == NULL) *// Si l'allocation a échoué*  {  exit(0); *// On arrête immédiatement le programme*  }  *// On peut continuer le programme normalement sinon.*  **return** 0;  } |

Si le pointeur est différent de NULL, le programme peut continuer, sinon il faut afficher un message d'erreur ou même mettre fin au programme, parce qu'il ne pourra pas continuer correctement s'il n'y a plus de place en mémoire.

**free : libérer de la mémoire**

Tout comme on utilisait la fonction fclose pour fermer un fichier dont on n'avait plus besoin, on va utiliser la fonction free pour libérer la mémoire quand on n'en a plus besoin.  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | void free(void\* pointeur); |

La fonction free a juste besoin de l'adresse mémoire à libérer. On va donc lui envoyer notre pointeur, c'est-à-dire memoireAllouee dans notre exemple.  
Voici le schéma complet et final, ressemblant à s'y méprendre à ce qu'on a vu dans le chapitre sur les fichiers :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | int main(int argc, char \*argv[])  {  long\* memoireAllouee = NULL;  memoireAllouee = malloc(**sizeof**(long));  **if** (memoireAllouee == NULL) *// On vérifie si la mémoire a été allouée*  {  exit(0); *// Erreur : on arrête tout !*  }  *// On peut utiliser ici la mémoire*  free(memoireAllouee); *// On n'a plus besoin de la mémoire, on la libère*  **return** 0;  } |

**Exemple concret d'utilisation**

On va faire quelque chose qu'on a appris à faire il y a longtemps : on va demander l'âge de l'utilisateur et on va le lui afficher.  
La seule différence avec ce qu'on faisait avant, c'est qu'ici la variable va être allouée *manuellement* (on dit aussi *dynamiquement*) au lieu d'automatiquement comme auparavant. Alors oui, du coup, le code est un peu plus compliqué. Mais faites l'effort de bien essayer de le comprendre, c'est important :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | int main(int argc, char \*argv[])  {  long\* memoireAllouee = NULL;  memoireAllouee = malloc(**sizeof**(long)); *// Allocation de la mémoire*  **if** (memoireAllouee == NULL)  {  exit(0);  }  *// Utilisation de la mémoire*  printf("Quel age avez-vous ? ");  scanf("%ld", memoireAllouee);  printf("Vous avez %ld ans**\n**", \*memoireAllouee);  free(memoireAllouee); *// Libération de mémoire*  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Quel age avez-vous ? 31  Vous avez 31 ans |

Attention : comme memoireAllouee est un pointeur, on ne l'utilise pas de la même manière qu'une vraie variable. Pour obtenir la valeur de la variable, il faut mettre une étoile devant : "*\*memoireAllouee*" (regardez le printf). Tandis que pour indiquer l'adresse, on a juste besoin d'écrire le nom du pointeur "*memoireAllouee*" (regardez le scanf)  
Tout cela a été expliqué dans le chapitre sur les pointeurs. Toutefois, on met généralement du temps à s'y faire, et il est probable que vous confondiez encore. Si c'est votre cas, vous DEVEZ relire le chapitre sur les pointeurs, qui est fondamental.

Revenons à notre code. On y a alloué dynamiquement une variable de type long.  
Au final, ce qu'on a écrit revient exactement au même que d'utiliser la méthode "automatique" qu'on connaît bien maintenant :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | int main(int argc, char \*argv[])  {  long maVariable = 0; *// Allocation de la mémoire (automatique)*  *// Utilisation de la mémoire*  printf("Quel age avez-vous ? ");  scanf("%ld", &maVariable);  printf("Vous avez %ld ans**\n**", maVariable);  **return** 0;  } *// Libération de la mémoire (automatique à la fin de la fonction)* |

**Code : Console**

|  |
| --- |
| Quel age avez-vous ? 31  Vous avez 31 ans |

**En résumé** : il y a 2 façons de créer une variable, c'est-à-dire d'allouer de la mémoire. Soit on le fait :

* *Automatiquement*: c'est la méthode que vous connaissez et qu'on a utilisée jusqu'ici.
* *Manuellement*(= *dynamiquement*) : c'est la méthode que je vous enseigne dans ce chapitre.

Je trouve la méthode dynamique compliquée et inutile !

Un peu plus compliquée... certes.  
Mais inutile, non ! On est parfois obligé d'allouer manuellement de la mémoire, comme on va le voir maintenant :)

**Allocation dynamique d'un tableau**

Pour le moment, on s'est servis de l'allocation dynamique uniquement pour créer une petite variable. Or en général, on ne se sert pas de l'allocation dynamique pour ça ;) On utilise la méthode automatique qui est plus simple.  
  
Quand a-t-on besoin de l'allocation dynamique me direz-vous ?  
Le plus souvent, on se sert de l'allocation dynamique pour créer un tableau dont on ne connaît pas la taille avant l'exécution du programme.  
  
Imaginons par exemple un programme qui stocke l'âge de tous les amis de l'utilisateur dans un tableau. Vous pourriez créer un tableau de long pour stocker les âges, comme ceci :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | long ageAmis[15]; |

Mais qui vous dit que l'utilisateur a 15 amis ? Peut-être qu'il en a plus que ça !  
Lorsque vous rédigez le code source, vous ne connaissez pas la taille que vous devez donner à votre tableau. Vous ne le saurez qu'à l'exécution, lorsque vous demanderez à l'utilisateur combien il a d'amis.  
L'intérêt de l'allocation dynamique est là : on va demander le nombre d'amis à l'utilisateur, puis on fera une allocation dynamique pour créer un tableau ayant exactement la taille nécessaire (ni trop petit, ni trop grand ^^ ). Si l'utilisateur a 15 amis, on créera un tableau de 15 long, s'il en a 28 on créera un tableau de 28 long etc.  
  
Comme je vous l'ai appris, il est interdit en C de créer un tableau en indiquant sa taille à l'aide d'une variable :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | long amis[nombreDAmis]; |

*(Notez : ce code marche peut-être sur certains compilateurs mais uniquement dans des cas précis, il est recommandé de ne pas l'utiliser !)*  
L'avantage de l'allocation dynamique, c'est qu'elle nous permet de créer un tableau qui a exactement la taille de la variable nombreDAmis, et cela grâce à un code qui marchera partout !  
  
  
On va demander au malloc de nous réserver nombreDAmis \* sizeof(long) octets en mémoire :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | amis = malloc(nombreDAmis \* **sizeof**(long)); |

Ce code permet de créer un tableau de type long qui a une taille correspondant exactement au nombre de ses amis !  
  
  
  
Voici ce que va faire le programme dans l'ordre :

1. On demande à l'utilisateur combien il a d'amis
2. On crée un tableau de long faisant une taille égale à son nombre d'amis (via malloc)
3. On demande l'âge de chacun de ses amis un à un, qu'on stocke dans le tableau
4. On affiche l'âge des amis pour montrer qu'on a bien mémorisé tout cela
5. A la fin, on n'a plus besoin du tableau contenant l'âge des amis : on le libère avec la fonction free.

**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | int main(int argc, char \*argv[])  {  long nombreDAmis = 0, i = 0;  long\* ageAmis = NULL; *// Ce pointeur va servir de tableau après l'appel du malloc*  *// On demande le nombre d'amis à l'utilisateur*  printf("Combien d'amis avez-vous ? ");  scanf("%ld", &nombreDAmis);  **if** (nombreDAmis > 0) *// Il faut qu'il ait au moins un ami (je le plains un peu sinon :p)*  {  ageAmis = malloc(nombreDAmis \* **sizeof**(long)); *// On alloue de la mémoire pour le tableau*  **if** (ageAmis == NULL) *// On vérifie si l'allocation a marché ou pas*  {  exit(0); *// On arrête tout*  }  *// On demande l'âge des amis un à un*  **for** (i = 0 ; i < nombreDAmis ; i++)  {  printf("Quel age a l'ami numero %ld ? ", i + 1);  scanf("%ld", &ageAmis[i]);  }  *// On affiche les âges stockés un à un*  printf("**\n\n**Vos amis ont les ages suivants :**\n**");  **for** (i = 0 ; i < nombreDAmis ; i++)  {  printf("%ld ans**\n**", ageAmis[i]);  }  *// On libère la mémoire allouée avec malloc, on n'en a plus besoin*  free(ageAmis);  }  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Combien d'amis avez-vous ? 5  Quel age a l'ami numero 1 ? 16  Quel age a l'ami numero 2 ? 18  Quel age a l'ami numero 3 ? 20  Quel age a l'ami numero 4 ? 26  Quel age a l'ami numero 5 ? 27  Vos amis ont les ages suivants :  16 ans  18 ans  20 ans  26 ans  27 ans |

Ce programme est tout à fait inutile : il demande les âges et les affiche ensuite. J'ai choisi de faire cela car c'est un exemple "simple" (enfin si vous avez compris le malloc ;) ).  
  
Que je vous rassure, dans la suite du cours nous aurons l'occasion d'utiliser le malloc pour des choses plus intéressantes que le stockage de l'âge de ses amis ;)

**La taille des variables**

Selon le type de variable que vous demandez à créer (char, int, double, float...), vous avez besoin de plus ou moins de mémoire.  
  
En effet, pour stocker un nombre compris entre -128 et 127 (un char), on n'a besoin que d'un octet en mémoire (c'est tout petit ^^ ).  
En revanche, un int occupe généralement 4 octets en mémoire. Quant au double, il occupe 8 octets.  
  
Le problème est... que ce n'est pas toujours le cas. Cela dépend de votre ordinateur : peut-être que chez vous un int occupe 8 octets, qui sait ?  
Notre objectif ici est de vérifier quelle taille occupe chacun des types sur votre ordinateur.  
  
Il y a un moyen très facile pour savoir cela : utiliser l'opérateur sizeof().  
Contrairement aux apparences, ce n'est pas une fonction mais une fonctionnalité *de base* du langage C. Vous devez juste indiquer entre parenthèses le type que vous voulez analyser.  
Pour connaître la taille d'un int, on devra donc écrire :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | **sizeof**(int) |

A la compilation, cela sera remplacé par un nombre : le nombre d'octets que prend int en mémoire. Chez moi, sizeof(int) vaut 4, ce qui signifie que int occupe 4 octets. Chez vous, c'est probablement la même valeur, mais ce n'est pas une règle. Testez, vous verrez ;)  
Vous pouvez faire des printf pour afficher cela :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | printf("char : %ld octets**\n**", **sizeof**(char));  printf("int : %ld octets**\n**", **sizeof**(int));  printf("long : %ld octets**\n**", **sizeof**(long));  printf("double : %ld octets**\n**", **sizeof**(double)); |

Chez moi, cela affiche :  
  
**Code : Console**

|  |
| --- |
| char : 1 octets  int : 4 octets  long : 4 octets  double : 8 octets |

Je n'ai pas mis tous les types que nous connaissons, je vous laisse le soin de tester vous-même la taille des autres types ;)  
Vous remarquerez que long et int occupent la même place en mémoire. Créer un long revient donc ici exactement à créer un int, cela prend 4 octets dans la mémoire.

En fait, le type "long" est équivalent à un type appelé "long int", qui est ici équivalent au type... "int". Bref, ça fait beaucoup de noms différents pour pas grand-chose au final ;) Avoir de nombreux types différents était utile à une époque où on n'avait pas beaucoup de mémoire. On cherchait à utiliser le minimum de mémoire possible en utilisant le type le plus adapté.  
Aujourd'hui, cela ne sert plus vraiment car la mémoire d'un ordinateur est très grande.

Peut-on afficher la taille d'un type personnalisé qu'on a créé (une structure) ?

Oui ! sizeof marche aussi sur les structures !  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **typedef** **struct** Coordonnees Coordonnees;  **struct** Coordonnees  {  long x;  long y;  };  int main(int argc, char \*argv[])  {  printf("Coordonnees : %ld octets**\n**", **sizeof**(Coordonnees));  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Coordonnees : 8 octets |

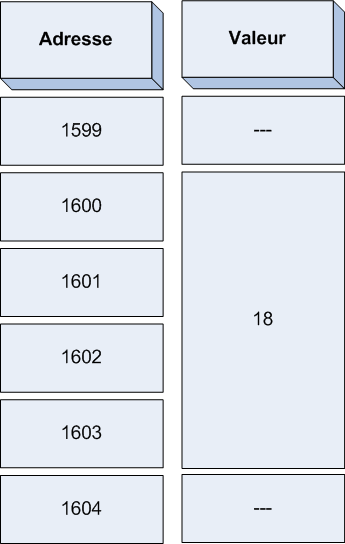
Plus une structure contient de sous-variables, plus elle prend de mémoire. Terriblement logique n'est-ce pas ? :D

**Une nouvelle façon de voir la mémoire**

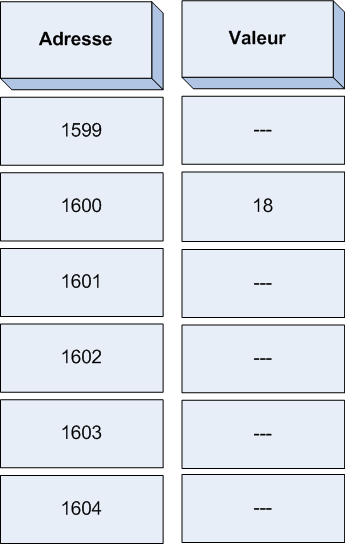
Jusqu'ici, mes schémas de mémoire étaient encore assez imprécis. On va enfin pouvoir les rendre précis et corrects maintenant qu'on connaît la taille de chacun des types de variables (c'est pas trop tôt ^^ )  
  
Si on déclare une variable de type long :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | long nombre = 18; |

... et que sizeof(long) indique 4 octets sur notre ordinateur, alors la variable occupera 4 octets en mémoire !  
  
Supposons que la variable nombre soit allouée à l'adresse 1600 en mémoire. On aurait alors le schéma suivant :



Ici, on voit bien que notre variable "nombre" de type long qui vaut 18 occupe 4 octets dans la mémoire.  
Elle commence à l'adresse 1600 (c'est son adresse) et termine à l'adresse 1603. La prochaine variable ne pourra donc être stockée qu'à partir de l'adresse 1604 !  
  
Si on avait fait la même chose avec un char, alors on n'aurait occupé qu'un seul octet en mémoire :



Imaginez maintenant un tableau de long !  
Chaque "case" du tableau occupera 4 octets. Si notre tableau fait 100 cases :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | long tableau[100]; |

Alors on occupera en réalité 4 \* 100 = 400 octets en mémoire :)

Même si le tableau est vide il prend 400 octets ?

Bien sûr ! La place en mémoire est réservée, aucun autre programme n'a le droit d'y toucher (à part le vôtre). Une fois qu'une variable est déclarée, elle prend immédiatement de la place en mémoire.  
  
  
Notez que si on crée un tableau de type "Coordonnees" :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | Coordonnees tableau[100]; |

... on utilisera (allez c'est facile :D ) : 8 \* 100 = 800 octets en mémoire.  
  
Il est important de bien comprendre ces petits calculs pour la suite du chapitre.  
C'est de la multiplication de niveau Primaire ça :p

**Allocation de mémoire dynamique**

Rentrons maintenant dans le vif du sujet.  
Le but du chapitre, c'était quoi justement ? :-°   
  
Ah oui : apprendre à demander de la mémoire manuellement.  
On va avoir besoin d'inclure la librairie <stdlib.h> (si vous avez suivi mes conseils, vous devriez avoir inclus cette librairie dans tous vos programmes de toute façon ;) ).  
  
Cette librairie contient 2 fonctions dont nous allons avoir besoin :

* **malloc**(*"Memory ALLOCation", c'est-à-dire "Allocation de mémoire"*) : demande au système d'exploitation la permission d'utiliser de la mémoire.
* **free**(*"Libérer"*) : 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.

Quand vous faites une allocation manuelle de mémoire (ce qu'on va apprendre à faire maintenant), vous devez toujours suivre ces 3 étapes :

1. Appeler **malloc**pour demander de la mémoire
2. **Vérifier la valeur retournée par malloc** pour savoir si l'OS a bien réussi à allouer la mémoire.
3. Une fois qu'on a fini d'utiliser la mémoire, on doit la libérer avec **free**. Si on ne le fait pas, on s'expose à des fuites de mémoire, c'est-à-dire que votre programme risque au final de prendre beaucoup de mémoire alors qu'il n'a en réalité plus besoin de tout cet espace.

Tiens, ces 3 étapes ça vous rappelle pas le chapitre sur les fichiers ça ? :D  
Ben moi si :p

Le principe est exactement le même qu'avec les fichiers : on alloue, on vérifie si l'allocation a marché, on utilise la mémoire, puis on libère quand on a fini d'utiliser.

Nous allons maintenant étudier la fonction malloc.

**malloc : demande d'allocation de mémoire**

Le prototype de la fonction malloc est assez comique vous allez voir :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | void\* malloc(size\_t nombreOctetsNecessaires); |

La fonction prend un paramètre : le nombre d'octets à réserver. Ainsi, il suffira d'écrire sizeof(long) dans ce paramètre pour réserver suffisamment d'espace pour stocker un long.  
  
Mais c'est surtout ce que la fonction renvoie qui est curieux : elle renvoie un... void\* :-°  
Si vous vous souvenez du chapitre sur les fonctions, je vous avais dit que "void" signifiait "vide" et qu'on utilisait ce type pour indiquer que la fonction ne retournait aucune valeur.  
Alors ici, on aurait une fonction qui retourne un "pointeur sur vide" ? o_O o_O o_O  
  
En voilà une bien bonne !  
Ces programmeurs ont décidemment un sens de l'humour très développé :lol:

Ca te dérangerait pas trop de nous donner quelques explications ?

Oui oui j'y viens ^^ Je me rappelle juste la première fois que j'ai vu le prototype de malloc, je suis resté la bouche ouverte un petit moment devant mon écran avant de comprendre :p  
  
En fait, cette fonction renvoie un pointeur indiquant l'adresse que l'OS a réservé pour votre variable. Si l'OS a trouvé de la place pour vous à l'adresse 1600, la fonction renvoie donc un pointeur contenant l'adresse 1600.  
Le problème, c'est que la fonction malloc ne sait pas quel type de variable vous cherchez à créer. En effet, vous ne lui donnez qu'un paramètre : le nombre d'octets en mémoire dont vous avez besoin. Si vous demandez 4 octets, ça pourrait aussi bien être un int qu'un long par exemple !  
  
Comme malloc ne sait pas quel type elle doit retourner, elle renvoie le type void\*. **Ce sera un pointeur sur n'importe quel type**. On peut dire que c'est un pointeur universel.  
  
  
Passons à la pratique. Si je veux m'amuser (*hahem*) à créer manuellement une variable de type long en mémoire, je devrai indiquer à malloc que j'ai besoin de sizeof(long) octets en mémoire.  
Je récupère le résultat du malloc dans un pointeur sur long.  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3 | long\* memoireAllouee = NULL; *// On crée un pointeur sur long*  memoireAllouee = malloc(**sizeof**(long)); *// La fonction malloc inscrit dans notre pointeur l'adresse qui a été reservee.* |

A la fin de ce code, memoireAllouee est un pointeur contenant une adresse qui vous a été réservée par l'OS, par exemple l'adresse 1600 (pour reprendre mes schémas de tout à l'heure).

**Tester le pointeur**

La fonction malloc a donc renvoyé dans notre pointeur memoireAllouee l'adresse qui a été réservée pour vous en mémoire.  
2 possibilités :

* Si l'allocation a marché, notre pointeur contient une adresse.
* Si l'allocation a échoué, notre pointeur contient l'adresse NULL.

Il est peu probable qu'une allocation échoue, mais cela peut arriver. Imaginez que vous demandiez à utiliser 34 Go de mémoire vive, il y a très peu de chances que l'OS vous réponde favorablement :lol:  
  
Il est néanmoins recommandé de **tester si l'allocation a marché**. On va faire ceci : si l'allocation a échoué, c'est qu'il n'y avait plus de mémoire de libre (c'est un cas critique). Dans un tel cas, le mieux est d'arrêter immédiatement le programme parce qu'il ne pourra pas continuer convenablement de toute manière.  
  
On va utiliser une fonction standard qu'on n'avait pas encore vue jusqu'ici : exit(). Elle arrête immédiatement le programme. Elle prend un paramètre : la valeur que le programme doit retourner (ça correspond au return du main()).  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | int main(int argc, char \*argv[])  {  long\* memoireAllouee = NULL;  memoireAllouee = malloc(**sizeof**(long));  **if** (memoireAllouee == NULL) *// Si l'allocation a échoué*  {  exit(0); *// On arrête immédiatement le programme*  }  *// On peut continuer le programme normalement sinon.*  **return** 0;  } |

Si le pointeur est différent de NULL, le programme peut continuer, sinon il faut afficher un message d'erreur ou même mettre fin au programme, parce qu'il ne pourra pas continuer correctement s'il n'y a plus de place en mémoire.

**free : libérer de la mémoire**

Tout comme on utilisait la fonction fclose pour fermer un fichier dont on n'avait plus besoin, on va utiliser la fonction free pour libérer la mémoire quand on n'en a plus besoin.  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | void free(void\* pointeur); |

La fonction free a juste besoin de l'adresse mémoire à libérer. On va donc lui envoyer notre pointeur, c'est-à-dire memoireAllouee dans notre exemple.  
Voici le schéma complet et final, ressemblant à s'y méprendre à ce qu'on a vu dans le chapitre sur les fichiers :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | int main(int argc, char \*argv[])  {  long\* memoireAllouee = NULL;  memoireAllouee = malloc(**sizeof**(long));  **if** (memoireAllouee == NULL) *// On vérifie si la mémoire a été allouée*  {  exit(0); *// Erreur : on arrête tout !*  }  *// On peut utiliser ici la mémoire*  free(memoireAllouee); *// On n'a plus besoin de la mémoire, on la libère*  **return** 0;  } |

**Exemple concret d'utilisation**

On va faire quelque chose qu'on a appris à faire il y a longtemps : on va demander l'âge de l'utilisateur et on va le lui afficher.  
La seule différence avec ce qu'on faisait avant, c'est qu'ici la variable va être allouée *manuellement* (on dit aussi *dynamiquement*) au lieu d'automatiquement comme auparavant. Alors oui, du coup, le code est un peu plus compliqué. Mais faites l'effort de bien essayer de le comprendre, c'est important :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | int main(int argc, char \*argv[])  {  long\* memoireAllouee = NULL;  memoireAllouee = malloc(**sizeof**(long)); *// Allocation de la mémoire*  **if** (memoireAllouee == NULL)  {  exit(0);  }  *// Utilisation de la mémoire*  printf("Quel age avez-vous ? ");  scanf("%ld", memoireAllouee);  printf("Vous avez %ld ans**\n**", \*memoireAllouee);  free(memoireAllouee); *// Libération de mémoire*  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Quel age avez-vous ? 31  Vous avez 31 ans |

Attention : comme memoireAllouee est un pointeur, on ne l'utilise pas de la même manière qu'une vraie variable. Pour obtenir la valeur de la variable, il faut mettre une étoile devant : "*\*memoireAllouee*" (regardez le printf). Tandis que pour indiquer l'adresse, on a juste besoin d'écrire le nom du pointeur "*memoireAllouee*" (regardez le scanf)  
Tout cela a été expliqué dans le chapitre sur les pointeurs. Toutefois, on met généralement du temps à s'y faire, et il est probable que vous confondiez encore. Si c'est votre cas, vous DEVEZ relire le chapitre sur les pointeurs, qui est fondamental.

Revenons à notre code. On y a alloué dynamiquement une variable de type long.  
Au final, ce qu'on a écrit revient exactement au même que d'utiliser la méthode "automatique" qu'on connaît bien maintenant :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | int main(int argc, char \*argv[])  {  long maVariable = 0; *// Allocation de la mémoire (automatique)*  *// Utilisation de la mémoire*  printf("Quel age avez-vous ? ");  scanf("%ld", &maVariable);  printf("Vous avez %ld ans**\n**", maVariable);  **return** 0;  } *// Libération de la mémoire (automatique à la fin de la fonction)* |

**Code : Console**

|  |
| --- |
| Quel age avez-vous ? 31  Vous avez 31 ans |

**En résumé** : il y a 2 façons de créer une variable, c'est-à-dire d'allouer de la mémoire. Soit on le fait :

* *Automatiquement*: c'est la méthode que vous connaissez et qu'on a utilisée jusqu'ici.
* *Manuellement*(= *dynamiquement*) : c'est la méthode que je vous enseigne dans ce chapitre.

Je trouve la méthode dynamique compliquée et inutile !

Un peu plus compliquée... certes.  
Mais inutile, non ! On est parfois obligé d'allouer manuellement de la mémoire, comme on va le voir maintenant :)

**Allocation dynamique d'un tableau**

Pour le moment, on s'est servis de l'allocation dynamique uniquement pour créer une petite variable. Or en général, on ne se sert pas de l'allocation dynamique pour ça ;) On utilise la méthode automatique qui est plus simple.  
  
Quand a-t-on besoin de l'allocation dynamique me direz-vous ?  
Le plus souvent, on se sert de l'allocation dynamique pour créer un tableau dont on ne connaît pas la taille avant l'exécution du programme.  
  
Imaginons par exemple un programme qui stocke l'âge de tous les amis de l'utilisateur dans un tableau. Vous pourriez créer un tableau de long pour stocker les âges, comme ceci :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | long ageAmis[15]; |

Mais qui vous dit que l'utilisateur a 15 amis ? Peut-être qu'il en a plus que ça !  
Lorsque vous rédigez le code source, vous ne connaissez pas la taille que vous devez donner à votre tableau. Vous ne le saurez qu'à l'exécution, lorsque vous demanderez à l'utilisateur combien il a d'amis.  
L'intérêt de l'allocation dynamique est là : on va demander le nombre d'amis à l'utilisateur, puis on fera une allocation dynamique pour créer un tableau ayant exactement la taille nécessaire (ni trop petit, ni trop grand ^^ ). Si l'utilisateur a 15 amis, on créera un tableau de 15 long, s'il en a 28 on créera un tableau de 28 long etc.  
  
Comme je vous l'ai appris, il est interdit en C de créer un tableau en indiquant sa taille à l'aide d'une variable :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | long amis[nombreDAmis]; |

*(Notez : ce code marche peut-être sur certains compilateurs mais uniquement dans des cas précis, il est recommandé de ne pas l'utiliser !)*  
L'avantage de l'allocation dynamique, c'est qu'elle nous permet de créer un tableau qui a exactement la taille de la variable nombreDAmis, et cela grâce à un code qui marchera partout !  
  
  
On va demander au malloc de nous réserver nombreDAmis \* sizeof(long) octets en mémoire :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | amis = malloc(nombreDAmis \* **sizeof**(long)); |

Ce code permet de créer un tableau de type long qui a une taille correspondant exactement au nombre de ses amis !  
  
  
  
Voici ce que va faire le programme dans l'ordre :

1. On demande à l'utilisateur combien il a d'amis
2. On crée un tableau de long faisant une taille égale à son nombre d'amis (via malloc)
3. On demande l'âge de chacun de ses amis un à un, qu'on stocke dans le tableau
4. On affiche l'âge des amis pour montrer qu'on a bien mémorisé tout cela
5. A la fin, on n'a plus besoin du tableau contenant l'âge des amis : on le libère avec la fonction free.

**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | int main(int argc, char \*argv[])  {  long nombreDAmis = 0, i = 0;  long\* ageAmis = NULL; *// Ce pointeur va servir de tableau après l'appel du malloc*  *// On demande le nombre d'amis à l'utilisateur*  printf("Combien d'amis avez-vous ? ");  scanf("%ld", &nombreDAmis);  **if** (nombreDAmis > 0) *// Il faut qu'il ait au moins un ami (je le plains un peu sinon :p)*  {  ageAmis = malloc(nombreDAmis \* **sizeof**(long)); *// On alloue de la mémoire pour le tableau*  **if** (ageAmis == NULL) *// On vérifie si l'allocation a marché ou pas*  {  exit(0); *// On arrête tout*  }  *// On demande l'âge des amis un à un*  **for** (i = 0 ; i < nombreDAmis ; i++)  {  printf("Quel age a l'ami numero %ld ? ", i + 1);  scanf("%ld", &ageAmis[i]);  }  *// On affiche les âges stockés un à un*  printf("**\n\n**Vos amis ont les ages suivants :**\n**");  **for** (i = 0 ; i < nombreDAmis ; i++)  {  printf("%ld ans**\n**", ageAmis[i]);  }  *// On libère la mémoire allouée avec malloc, on n'en a plus besoin*  free(ageAmis);  }  **return** 0;  } |

**Code : Console**

|  |
| --- |
| Combien d'amis avez-vous ? 5  Quel age a l'ami numero 1 ? 16  Quel age a l'ami numero 2 ? 18  Quel age a l'ami numero 3 ? 20  Quel age a l'ami numero 4 ? 26  Quel age a l'ami numero 5 ? 27  Vos amis ont les ages suivants :  16 ans  18 ans  20 ans  26 ans  27 ans |

Ce programme est tout à fait inutile : il demande les âges et les affiche ensuite. J'ai choisi de faire cela car c'est un exemple "simple" (enfin si vous avez compris le malloc ;) ).  
  
Que je vous rassure, dans la suite du cours nous aurons l'occasion d'utiliser le malloc pour des choses plus intéressantes que le stockage de l'âge de ses amis ;)

TP : Réalisation d'un pendu

## Les consignes

Je veux tout d'abord qu'on se mette bien d'accord sur les règles du pendu à réaliser. Je vais donc vous donner ici les consignes, c'est-à-dire vous expliquer comment doit fonctionner le jeu que vous allez créer.  
  
Tout le monde connaît le pendu n'est-ce pas ? Allez, un petit rappel ne peut pas faire de mal ;)  
Le but du pendu est de retrouver un mot caché en moins de 10 coups (mais vous pouvez changer ce nombre maximal pour corser la difficulté bien sûr !).

### Déroulement d'une partie

Supposons que le mot caché soit ROUGE.  
Vous proposez une lettre à l'ordinateur, par exemple la lettre A. L'ordinateur vérifie si cette lettre se trouve dans le mot caché

Rappelez-vous : il y a une fonction toute prête dans string.h pour rechercher une lettre dans un mot ! Notez que vous n'êtes pas obligés de l'utiliser toutefois (personnellement je ne m'en suis pas servi).

A partir de là, 2 possibilités :

* La lettre se trouve effectivement dans le mot : dans ce cas on dévoile le mot avec les lettres qu'on a déjà trouvées.
* La lettre ne se trouve pas dans le mot (c'est le cas ici, car A n'est pas dans ROUGE) : on indique au joueur que la lettre ne s'y trouve pas, et on diminue le nombre de coups restants. Quand il ne nous reste plus de coups (0 coups) le jeu est terminé et on a perdu.

Dans un "vrai" pendu, il y aurait normalement un dessin d'un bonhomme qui se fait pendre au fur et à mesure que l'on fait des erreurs. Bon, comme là on travaille en console, ce serait un peu hardcore de dessiner un bonhomme qui se fait pendre rien qu'avec du texte, donc on va se contenter d'afficher une simple phrase comme "Il vous reste X coups avant une mort certaine".  
Les plus courageux d'entre vous essaieront peut-être de dessiner quand même le bonhomme en console à grands coups de printf, mais je vous préviens : ce n'est pas le but du TP :D

Supposons maintenant que le joueur tape la lettre G. Celle-ci se trouve dans le mot caché, donc on ne diminue pas le nombre de coups restants pour le joueur. On affiche le mot secret avec les lettres qu'on a déjà découvertes, c'est-à-dire qu'on affiche quelque chose comme :  
  
**Code : Console**

|  |
| --- |
| Mot secret : \*\*\*G\* |

Si ensuite on tape un R, comme la lettre s'y trouve, on la rajoute à la liste des lettres trouvées et on affiche le mot avec les lettres déjà découvertes :  
  
**Code : Console**

|  |
| --- |
| Mot secret : R\*\*G\* |

#### Le cas des lettres multiples

Dans certains mots, une même lettre peut apparaître 2, 3 fois, voire même plus !  
Par exemple, il y a 2 Z dans PUZZLE  
De même, il y a 3 E dans ELEMENT  
  
Que fait-on dans un cas comme ça ? Les règles du pendu sont claires : si le joueur tape la lettre E, toutes les lettres E du mot ELEMENT doivent être découvertes d'un seul coup :  
  
**Code : Console**

|  |
| --- |
| Mot secret : E\*E\*E\*\* |

Il ne faut donc pas taper 3 fois la lettre E pour que tous les E soient découverts. Ca peut paraître évident à certains d'entre vous, mais je préfère le dire avant on sait jamais :p

#### Exemple d'une partie complète

Voici à quoi devrait ressembler une partie complète en console de votre programme lorsqu'il sera terminé :  
  
**Code : Console**

|  |
| --- |
| Bienvenue dans le Pendu !        Il vous reste 10 coups a jouer  Quel est le mot secret ? \*\*\*\*\*\*  Proposez une lettre : E      Il vous reste 9 coups a jouer  Quel est le mot secret ? \*\*\*\*\*\*  Proposez une lettre : A      Il vous reste 9 coups a jouer  Quel est le mot secret ? \*A\*\*\*\*  Proposez une lettre : O      Il vous reste 9 coups a jouer  Quel est le mot secret ? \*A\*\*O\*  Proposez une lettre : P      Il vous reste 8 coups a jouer  Quel est le mot secret ? \*A\*\*O\*  Proposez une lettre : M      Il vous reste 8 coups a jouer  Quel est le mot secret ? MA\*\*O\*  Proposez une lettre : N      Il vous reste 8 coups a jouer  Quel est le mot secret ? MA\*\*ON  Proposez une lettre : R      Gagne ! Le mot secret etait bien : MARRON |

#### Saisie d'une lettre en console

La lecture d'une lettre dans la console est plus compliquée qu'il n'y paraît.  
Intuitivement, pour récupérer un caractère, vous devriez avoir pensé à :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | scanf("%c", &maLettre); |

Et effectivement, c'est bien. %c indique que l'on attend un caractère, qu'on stockera dans maLettre (une variable de type char).  
  
Tout se passe très bien... tant qu'on ne refait pas un scanf. En effet, vous pouvez tester le code suivant :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | int main(int argc, char\* argv[])  {  char maLettre = 0;    scanf("%c", &maLettre);  printf("%c", maLettre);    scanf("%c", &maLettre);  printf("%c", maLettre);    **return** 0;  } |

Normalement, ce code est censé vous demander une lettre et vous l'afficher, et cela 2 fois.  
Testez. Que se passe-t-il ? Vous rentrez une lettre d'accord mais... le programme s'arrête de suite après, il ne vous demande pas la seconde lettre ! On dirait qu'il ignore le second scanf.

Que s'est-il passé ?

En fait, quand vous rentrez du texte en console, tout ce que vous tapez est stocké quelque part en mémoire, y compris l'appui sur la touche Entrée (\n).   
  
Ainsi, la première fois que vous rentrez une lettre (par exemple A) puis que vous tapez Entrée, c'est la lettre A qui est renvoyée par le scanf. Mais la seconde fois, scanf renvoie le \n correspondant à la touche "Entrée" que vous aviez tapée auparavant !

Oulah, comment éviter cela ?

Le mieux, c'est de créer notre propre petite fonction lireCaractere() :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | char lireCaractere()  {  char caractere = 0;    caractere = getchar(); *// On lit le premier caractère*  caractere = toupper(caractere); *// On met la lettre en majuscule si elle ne l'est pas déjà*    *// On lit les autres caractères mémorisés un à un jusqu'à l'\n (pour les effacer)*  **while** (getchar() != '\n') ;    **return** caractere; *// On retourne le premier caractère qu'on a lu*    } |

Cette fonction utilise getchar() qui est une fonction de stdio identique à scanf("%c", &lettre);. La fonction getchar renvoie le caractère que le joueur a tapé.  
Après, j'utilise une fonction standard qu'on n'a pas eu l'occasion d'étudier dans le cours : *toupper()*. Cette fonction transforme la lettre indiquée en majuscule. Comme ça, le jeu fonctionnera même si le joueur tape des lettres minuscules. **Il faut inclure ctype.h pour pouvoir utiliser cette fonction** (ne l'oubliez pas !)  
  
Ensuite vient la partie la plus intéressante : celle où je vide les autres caractères qui auraient pu avoir été tapés. En effet, en refaisant un getchar on prend le caractère suivant que l'utilisateur a tapé (par exemple l'Entrée \n).  
Ce que je fais est simple et tient en une ligne : j'appelle la fonction getchar en boucle jusqu'à tomber sur le caractère \n. La boucle s'arrête dès qu'on tombe sur l'\n, ce qui signifie qu'on a "lu" tous les autres caractères de la mémoire, ils ont donc été vidés de la mémoire. **On dit qu'on vide le buffer**.

Euh pourquoi il y a un point-virgule à la fin du while et pourquoi il n'y a pas d'accolades ?

En fait, je fais une boucle qui ne contient pas d'instructions (la seule instruction c'est le getchar entre les parenthèses). Les accolades ne sont pas nécessaires vu que je n'ai rien d'autre à faire qu'un getchar. Je mets donc un point-virgule pour remplacer les accolades. Ce point-virgule signifie "*ne rien faire à chaque passage dans la boucle*". C'est un peu particulier je le reconnais, mais c'est une technique à connaître qu'utilisent les programmeurs pour faire des boucles très courtes et très simples.

Pour mieux comprendre, dites vous que le while aurait aussi pu être écrit comme ceci :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | **while** (getchar() != '\n')  {    } |

Il n'y a rien entre accolades c'est volontaire, vu qu'on n'a rien d'autre à faire. Ma technique de placer juste un point-virgule est plus courte que celle avec des accolades, c'est tout ;)

Enfin, la fonction lireCaractere retourne le premier caractère qu'elle a lu (la variable *caractere*).  
  
  
**En résumé** : pour récupérer une lettre dans votre code, vous n'utiliserez pas :  
**Code : C**

|  |  |
| --- | --- |
| 1 | scanf("%c", &maLettre); |

Mais vous utiliserez à la place notre super fonction :  
**Code : C**

|  |  |
| --- | --- |
| 1 | maLettre = lireCaractere(); |

### Dictionnaire de mots

Dans un premier temps pour vos tests, je vais vous demander de fixer le mot secret directement dans votre code. Vous écrirez donc par exemple :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | char motSecret[] = "MARRON"; |

Alors oui, bien sûr le mot secret sera toujours le même si on laisse ça comme ça, ce qui n'est pas très rigolo :p  
Je vous demande de faire comme ça dans un premier temps pour ne pas mélanger les problèmes. En effet, une fois que votre jeu de pendu fonctionnera correctement (et seulement à partir de ce moment-là) vous attaquerez la 2ème phase : la création du dictionnaire de mots.

Qu'est-ce que c'est le "dictionnaire de mots" ?

C'est un fichier qui contiendra plein de mots pour votre jeu de pendu. Il doit y avoir un mot par ligne. Exemple :  
  
**Code : Console**

|  |
| --- |
| MAISON  BLEU  AVION  XYLOPHONE  ABEILLE  IMMEUBLE  GOURDIN  NEIGE  ZERO |

A chaque nouvelle partie, votre programme devra ouvrir ce fichier et prendre un des mots au hasard dans la liste. Grâce à cette technique, vous aurez un fichier à part que vous pourrez éditer tant que vous voudrez pour ajouter des mots secrets possibles pour le pendu :)

Vous aurez remarqué que depuis le début je fais exprès de mettre tous mes mots en majuscule. En effet, dans le pendu on ne fait pas la distinction entre les majuscules et les minuscules, donc le mieux est de se dire dès le début : "tous mes mots seront en majuscules". A vous de prévenir le joueur (dans le mode d'emploi du jeu par exemple) qu'il est censé rentrer des lettres majuscules et non des minuscules  
Par ailleurs, on fait exprès d'ignorer les accents pour simplifier (si on doit commencer à tester le é, le è, le ê, le ë... on n'a pas fini :p ). Vous devrez donc écrire vos mots dans le dictionnaire **entièrement en majuscules et sans accents**.

Le problème qui se posera rapidement pour vous sera de savoir combien il y a de mots dans le dictionnaire. En effet, si vous voulez choisir un mot au hasard, il faudra tirer au sort un nombre entre 0 et X, et vous ne savez pas combien de mots contient votre fichier à priori.  
Pour résoudre le problème, 2 solutions :

* Soit vous indiquez sur la première ligne du fichier le nombre de mots qu'il contient :  
  **Code : Console**

|  |
| --- |
| 3  MAISON  BLEU  AVION |

* Mais cette technique est ennuyeuse car il faudra recompter manuellement le nombre de mots à chaque fois que vous rajoutez un mot. Aussi je vous propose plutôt de compter automatiquement le nombre de mots en lisant une première fois le fichier avec votre programme. Pour savoir combien il y a de mots, c'est simple : vous comptez le nombre d'\n (retours à la ligne) dans le fichier :)  
  Une fois que vous aurez lu le fichier une première fois pour compter les \n, vous ferez un rewind pour revenir au début. Vous n'aurez alors plus qu'à tirer un nombre au sort parmi le nombre de mots que vous avez compté, puis à vous rendre au mot que vous avez choisi et à le stocker dans une chaîne en mémoire.

Je vous laisse réfléchir un peu là-dessus, je vais pas trop vous aider quand même sinon ça sera plus un TP :p  
Sachez que vous avez acquis toutes les connaissances qu'il faut dans les chapitres précédents, donc vous êtes parfaitement capables de réaliser ce jeu. Ca va prendre plus ou moins de temps et c'est moins facile qu'il n'y paraît, mais en vous organisant correctement (et en créant suffisamment de fonctions) vous y arriverez ;)  
  
Bon courage, et surtout : per-sé-vé-rez !

## La solution (1 : le code du jeu)

Si vous lisez ces lignes, c'est soit que vous avez terminé le programme, soit... que vous n'arrivez pas à le terminer ;)  
  
J'ai personnellement mis plus de temps que je ne le pensais pour réaliser ce petit jeu apparemment tout bête. C'est souvent comme ça : on se dit "boah c'est facile" alors qu'en fait il y a plein de cas à gérer ;)  
Je persiste toutefois à dire que **vous êtes tous capables de le faire**. Il vous faudra plus ou moins de temps (quelques minutes, quelques heures, quelques jours ?), mais ça n'a jamais été une course. Je préfère que vous y passiez beaucoup de temps et que vous y arriviez, plutôt que vous n'essayiez que 5 minutes et que vous regardiez la solution.

Rappelez-vous que c'est précisément dans les TP que le gros du travail se fait : la pratique, c'est vraiment pas la même chose que la théorie :D  
C'est donc principalement pendant les TP que vous progressez, raison de plus pour y mettre tous vos efforts ;)

N'allez pas croire que j'ai écrit le programme d'une traite. Moi aussi, comme vous, j'y suis allé pas à pas. J'ai commencé par faire quelque chose de très simple, puis petit à petit j'ai amélioré le code pour finalement arriver au résultat final :)  
J'ai fait plusieurs erreurs en codant : j'ai oublié à un moment d'initialiser une variable correctement, j'ai oublié de mettre un prototype de fonction ou encore de supprimer une variable qui ne servait plus dans mon code. J'ai même, je l'avoue, oublié un bête point-virgule à un moment à la fin d'une ligne.  
  
Tout ça pour dire quoi ? Que je ne suis pas infaillible et que je vis à peu près les mêmes frustrations que vous : "ESPECE DE PROGRAMME DE \*\*\*\*\* TU VAS TE METTRE A MARCHER OUI OU NON !?".  
  
Je vais vous présenter la solution en 2 temps :

1. D'abord je vais vous montrer comment j'ai fait le code du jeu lui-même, en fixant le mot caché directement dans le code (j'ai pris le mot MARRON car il me permet de tester si je gère bien les lettres en double comme R ici).
2. Ensuite, je vous montrerai comment dans un second temps j'ai ajouté la gestion du dictionnaire de mots pour tirer au sort un mot secret pour le joueur.

Bien sûr, je pourrais vous montrer tout le code d'un coup mais... ça ferait beaucoup à la fois, et nombre d'entre vous n'auraient pas le courage de se pencher dans le code (je suis comme vous, quand y'a un gros code à comprendre je mets plus de temps que pour un petit code :D )  
  
  
Je vais essayer de vous expliquer pas à pas mon raisonnement. Ce qui compte, ce n'est pas le résultat, mais la façon dont on réfléchit.

### Analyse de la fonction main

Comme tout le monde le sait, tout commence par un main. On n'oublie pas d'inclure les librairies stdio, stdlib et ctype (pour la fonction toupper()) dont on aura besoin :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #include <stdio.h>  #include <stdlib.h>  #include <ctype.h>  int main(int argc, char\* argv[])  {  **return** 0;  } |

Ok, jusque là tout le monde devrait suivre :D  
Notre main va gérer la plupart du jeu et faire appel à 2-3 de nos fonctions quand il en aura besoin.  
  
Commençons par déclarer les variables dont on va avoir besoin. Rassurez-vous, je n'ai pas pensé de suite à toutes ces variables, il y en avait un peu moins la première fois que j'ai écrit le code ;)  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #include <stdio.h>  #include <stdlib.h>  #include <ctype.h>  int main(int argc, char\* argv[])  {  char lettre = 0; *// Stocke la lettre proposée par l'utilisateur (retour du scanf)*  char motSecret[] = "MARRON"; *// C'est le mot à trouver*  int lettreTrouvee[6] = {0}; *// Un tableau de booléens. Chaque case correspond à une lettre du mot secret. 0 = lettre non trouvée, 1 = lettre trouvée*  long coupsRestants = 10; *// Compteur de coups restants (0 = mort)*  long i = 0; *// Une petite variable pour parcourir les tableaux*  **return** 0;  } |

J'ai fait exprès de mettre une déclaration de variable par ligne ainsi que pas mal de commentaires pour que vous compreniez. En pratique, vous n'aurez pas forcément besoin de mettre tous ces commentaires, et vous pourrez grouper plusieurs déclarations de variables sur la même ligne.  
  
Je pense que la plupart de ces variables semblent logiques : la variable lettre stocke la lettre que l'utilisateur tape à chaque fois, le motSecret c'est le mot à trouver, coupsRestants le nombre de coups etc.  
La variable i est une petite variable que j'utilise pour parcourir mes tableaux avec des for. Elle n'est donc pas extrêmement importante mais elle est nécessaire si on veut faire nos boucles.  
  
Enfin, la variable à laquelle il fallait penser, celle qui fait la différence, c'est mon tableau de booléens *lettreTrouvee*. Vous remarquerez que je lui ai donné pour taille le nombre de lettres du mot secret (6). Ce n'est pas un hasard : chaque case de ce tableau de booléens représente une lettre du mot secret. Ainsi, la première case représente la première lettre, la seconde case représente la seconde lettre etc.  
Les cases du tableau sont au départ initialisées à 0, ce qui signifie "Lettre non trouvée". Au fur et à mesure de l'avancement du jeu, ce tableau sera modifié. Pour chaque lettre du mot secret trouvée, la case correspondante du tableau *lettreTrouvee*sera mise à 1.  
  
Par exemple, si à un moment du jeu j'ai l'affichage suivant : M\*RR\*N, c'est que mon tableau d'int a les valeurs : 101101 (1 pour chaque lettre qui a été trouvée).  
Il est ainsi facile de savoir quand on a gagné : il suffit de vérifier si le tableau de booléens ne contient que des 1.  
En revanche, on a perdu si le compteur coupsRestants tombe à 0.  
  
Passons à la suite :  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | printf("Bienvenue dans le Pendu !**\n\n**"); |

Bon, ça c'est un message de bienvenue, j'ai rien à ajouter :D  
  
Ensuite, on commence la boucle principale du jeu :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2 | **while** (coupsRestants > 0 && !gagne(lettreTrouvee))  { |

Le jeu continue tant qu'il reste des coups (coupsRestants > 0) et tant qu'on n'a pas gagné.  
Si on n'a plus de coups à jouer, c'est qu'on a perdu. Si on a gagné, c'est... qu'on a gagné ^^  
  
"gagne" est une fonction qui analyse le tableau lettreTrouvee. Elle renvoie vrai (1) si le joueur a gagné (le tableau lettreTrouvee ne contient que des 1), faux (0) si le joueur n'a pas encore gagné.  
Je ne vous explique pas le fonctionnement de cette fonction en détail pour le moment. On verra cela plus tard. Pour le moment, vous avez juste besoin de savoir ce que fait la fonction.  
  
La suite :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | printf("**\n\n**Il vous reste %ld coups a jouer", coupsRestants);  printf("**\n**Quel est le mot secret ? ");  */\* On affiche le mot secret en masquant les lettres non trouvées*  *Exemple : \*A\*\*ON \*/*  **for** (i = 0 ; i < 6 ; i++)  {  **if** (lettreTrouvee[i]) *// Si on a trouvé la lettre n°i*  printf("%c", motSecret[i]); *// On l'affiche*  **else**  printf("\*"); *// Sinon, on affiche une étoile pour les lettres non trouvées*  } |

On affiche à chaque coup le nombre de coups restants ainsi que le mot secret (masqué par des \* pour les lettres non trouvées).  
L'affichage du mot secret masqué par des \* se fait grâce à une boucle for. On analyse pour chaque lettre si elle a été trouvée (if lettreTrouvee[i]). Si c'est le cas, on affiche la lettre. Sinon, on affiche une \* de remplacement pour masquer la lettre.  
  
  
Maintenant qu'on a affiché ce qu'il fallait, on va demander au joueur de saisir une lettre :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2 | printf("**\n**Proposez une lettre : ");  lettre = lireCaractere(); |

Je fais appel à notre fonction lireCaractere(). Celle-ci lit le premier caractère tapé, le met en majuscule et vide le buffer (c'est-à-dire qu'elle vide les autres caractères qui auraient pu être restés dans la mémoire).  
  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | *// Si ce n'était PAS la bonne lettre*  **if** (!rechercheLettre(lettre, motSecret, lettreTrouvee))  {  coupsRestants--; *// On enlève un coup au joueur*  }  } |

On teste si la lettre entrée se trouve dans motSecret. On fait appel pour cela à une fonction maison appelée rechercheLettre. Nous verrons peu après le code de cette fonction.  
Pour le moment, tout ce que vous avez besoin de savoir, c'est que cette fonction renvoie "vrai" si la lettre se trouve dans le mot, "faux" si elle ne s'y trouve pas.  
  
Mon if, vous l'aurez remarqué, commence par un point d'exclamation "!" qui signifie "non". La condition se lit donc "Si la lettre n'a pas été trouvée".  
Que fait-on si la lettre n'a pas été trouvée ? On diminue le nombre de coups restants.

Notez que la fonction rechercheLettre met aussi à jour le tableau de booléens lettreTrouvee. Elle met des 1 dans les cases des lettres qui ont été trouvées.

La boucle principale du jeu s'arrête là. On recommence donc au début de la boucle et on vérifie s'il reste des coups à jouer et si on n'a pas déjà gagné.  
  
  
Lorsqu'on sort de la boucle principale du jeu, il reste à afficher si on a gagné ou pas avant que le programme ne s'arrête :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **if** (gagne(lettreTrouvee))  printf("**\n\n**Gagne ! Le mot secret etait bien : %s", motSecret);  **else**  printf("**\n\n**Perdu ! Le mot secret etait : %s", motSecret);  **return** 0;  } |

On fait appel à la fonction "gagne" pour vérifier si on a gagné. Si c'est le cas, alors on affiche le message "Gagné !", sinon c'est qu'on n'avait plus de coups à jouer, on a été pendu.

### Analyse de la fonction gagne

Voyons voir maintenant le code de la fonction gagne :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | int gagne(int lettreTrouvee[])  {  long i = 0;  int joueurGagne = 1;  **for** (i = 0 ; i < 6 ; i++)  {  **if** (lettreTrouvee[i] == 0)  joueurGagne = 0;  }  **return** joueurGagne;  } |

Cette fonction prend le tableau de booléens lettreTrouvee pour paramètre. Elle renvoie un booléen : vrai si on a gagné, faux si on a perdu.  
  
Le code de cette fonction est plutôt simple, vous devriez tous le comprendre. On parcourt lettreTrouvee et on vérifie si UNE des cases vaut faux (0). Si une des lettres n'a pas encore été trouvée, c'est qu'on a perdu : on met alors le booléen joueurGagne à faux (0). Sinon, si toutes les lettres ont été trouvée, le booléen vaut vrai (1) et la fonction renverra donc vrai.

### Analyse de la fonction rechercheLettre

La fonction rechercheLettre a 2 missions :

* Renvoyer un booléen indiquant si la lettre se trouvait bien dans le mot secret
* Mettre à jour les cases du tableau lettreTrouvee correspondant aux positions de la lettre qui a été trouvée à 1.

**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | int rechercheLettre(char lettre, char motSecret[], int lettreTrouvee[])  {  long i = 0;  int bonneLettre = 0;  *// On parcourt motSecret pour vérifier si la lettre proposée y est*  **for** (i = 0 ; motSecret[i] != '\0' ; i++)  {  **if** (lettre == motSecret[i]) *// Si la lettre y est*  {  bonneLettre = 1; *// On mémorise que c'était une bonne lettre*  lettreTrouvee[i] = 1; *// On met à 1 le case du tableau de booléens correspondant à la lettre actuelle*  }  }  **return** bonneLettre;  } |

On parcourt donc la chaîne motSecret caractère par caractère. A chaque fois, on vérifie si la lettre que le joueur a proposée est une lettre du mot. Si la lettre correspond, alors on fait 2 choses :

* On change la valeur du booléen bonneLettre à 1, pour que la fonction retourne 1 car la lettre se trouvait effectivement dans motSecret.
* On met à jour le tableau lettreTrouvee à la position actuelle pour indiquer que cette lettre a été trouvée.

L'avantage de cette technique c'est qu'ainsi on parcourt tout le tableau (on ne s'arrête pas à la première lettre trouvée). Cela nous permet de bien mettre à jour le tableau lettreTrouvee, au cas où une lettre serait présente en plusieurs exemplaires dans le mot secret (comme c'est le cas pour les 2 R dans MARRON).

### Pfiou !

Et voilà, on a fait le tour :D  
Je résume le code source que j'utilise :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109 | */\**  *Jeu du pendu*  *Par M@teo21, pour le Site du Zér0*  *<lien url="www.siteduzero.com">www.siteduzero.com</lien>*  *main.c*  *------*  *Fonctions principales de gestion du jeu*  *\*/*  #include <stdio.h>  #include <stdlib.h>  #include <ctype.h>  int gagne(int lettreTrouvee[]);  int rechercheLettre(char lettre, char motSecret[], int lettreTrouvee[]);  char lireCaractere();  int main(int argc, char\* argv[])  {  char lettre = 0; *// Stocke la lettre proposée par l'utilisateur (retour du scanf)*  char motSecret[] = "MARRON"; *// C'est le mot à trouver*  int lettreTrouvee[6] = {0}; *// Un tableau de booléens. Chaque case correspond à une lettre du mot secret. 0 = lettre non trouvée, 1 = lettre trouvée*  long coupsRestants = 10; *// Compteur de coups restants (0 = mort)*  long i = 0; *// Une petite variable pour parcourir les tableaux*  printf("Bienvenue dans le Pendu !**\n\n**");  *// On continue à jouer tant qu'il reste au moins un coup à jouer ou qu'on*  *// n'a pas gagné*  **while** (coupsRestants > 0 && !gagne(lettreTrouvee))  {  printf("**\n\n**Il vous reste %ld coups a jouer", coupsRestants);  printf("**\n**Quel est le mot secret ? ");  */\* On affiche le mot secret en masquant les lettres non trouvées*  *Exemple : \*A\*\*ON \*/*  **for** (i = 0 ; i < 6 ; i++)  {  **if** (lettreTrouvee[i]) *// Si on a trouvé la lettre n°i*  printf("%c", motSecret[i]); *// On l'affiche*  **else**  printf("\*"); *// Sinon, on affiche une étoile pour les lettres non trouvées*  }  printf("**\n**Proposez une lettre : ");  lettre = lireCaractere();  *// Si ce n'était PAS la bonne lettre*  **if** (!rechercheLettre(lettre, motSecret, lettreTrouvee))  {  coupsRestants--; *// On enlève un coup au joueur*  }  }  **if** (gagne(lettreTrouvee))  printf("**\n\n**Gagne ! Le mot secret etait bien : %s", motSecret);  **else**  printf("**\n\n**Perdu ! Le mot secret etait : %s", motSecret);  **return** 0;  }  char lireCaractere()  {  char caractere = 0;  caractere = getchar(); *// On lit le premier caractère*  caractere = toupper(caractere); *// On met la lettre en majuscule si elle ne l'est pas déjà*  *// On lit les autres caractères mémorisés un à un jusqu'à l'\n*  **while** (getchar() != '\n') ;  **return** caractere; *// On retourne le premier caractère qu'on a lu*  }  int gagne(int lettreTrouvee[])  {  long i = 0;  int joueurGagne = 1;  **for** (i = 0 ; i < 6 ; i++)  {  **if** (lettreTrouvee[i] == 0)  joueurGagne = 0;  }  **return** joueurGagne;  }  int rechercheLettre(char lettre, char motSecret[], int lettreTrouvee[])  {  long i = 0;  int bonneLettre = 0;  *// On parcourt motSecret pour vérifier si la lettre proposée y est*  **for** (i = 0 ; motSecret[i] != '\0' ; i++)  {  **if** (lettre == motSecret[i]) *// Si la lettre y est*  {  bonneLettre = 1; *// On mémorise que c'était une bonne lettre*  lettreTrouvee[i] = 1; *// On met à 1 le case du tableau de booléens correspondant à la lettre actuelle*  }  }  **return** bonneLettre;  } |

Vous noterez que j'ai mis pour le moment les prototypes en haut du fichier, au lieu de les mettre dans un .h. C'est simplement parce que pour le moment le programme n'est pas encore bien compliqué et ne nécessite pas de créer plus que le simple fichier main.c de base.  
Cependant, les choses vont changer dès qu'on attaquera la phase 2 avec la gestion du dictionnaire ;)  
  
Il y a plein de façons différentes d'imaginer le code du jeu du pendu, aussi si vous y étiez arrivés mais avec un code complètement différent, sachez que c'est normal. Je ne dis pas que mon code est le meilleur, je dis juste que c'était une bonne façon de faire.  
  
Comment savoir si vous avez bien fait ?  
C'est simple :

* Si votre code ne fait pas 2000 lignes, déjà, c'est bien. Généralement, un bon code est court et concis.
* Si votre programme met 5 minutes à vérifier si la lettre tapée se trouve dans le mot secret, c'est généralement qu'il y a un problème et que votre code n'est pas assez optimisé. Rassurez-vous, sur un jeu du pendu cela ne devrait pas arriver, mais à l'avenir il faudra y penser.
* Si votre programme prend 450 Mo de mémoire vive, il faut commencer à vous inquiéter et vérifier si vous ne prenez pas un peu trop de mémoire. Là encore, sur un petit programme comme ça c'est difficile à faire (à moins de faire une boucle infinie de malloc :D )

La meilleure façon d'obtenir des commentaires c'est bien souvent de proposer votre code sur les forums du Site du Zér0 pour que des gens plus expérimentés puissent vous donner leur avis. N'hésitez donc pas à le faire, on apprend généralement des tas de choses comme ça ;)

## La solution (2 : la gestion du dictionnaire)

Pour qu'on puisse faire des tests, la première chose à faire c'est de créer ce fameux dictionnaire de mots. Même s'il est court, c'est pas grave c'est pour les tests.  
  
Je vais donc créer un fichier dico.txt dans le même répertoire que mon projet. Pour le moment, je mets les mots suivants :  
  
**Code : Autre**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | MAISON  BLEU  AVION  XYLOPHONE  ABEILLE  IMMEUBLE  GOURDIN  NEIGE  ZERO |

Une fois que j'aurai terminé de coder le programme, je reviendrai bien sûr sur ce dictionnaire et j'y ajouterai des tooonnes de mots tordus à rallonge comme ANTICONSTITUTIONNELLEMENT :D  
  
Mais pour le moment, retournons à nos instructions.

### Préparation des nouveaux fichiers

La lecture du dico va demander pas mal de lignes de codes (du moins je le pressens :p ). Je prends donc les devants en ajoutant **un nouveau fichier à mon projet** : dico.c (qui sera chargé de la lecture du dico). Dans la foulée, je crée le dico.h qui contiendra les prototypes des fonctions de dico.h.  
  
Dans dico.c, je commence par inclure les librairies dont j'aurai besoin ainsi que mon dico.h  
A priori, comme souvent, j'aurai besoin de stdio et stdlib ici. En plus de cela, je vais être amené à piocher un nombre au hasard dans le dico, donc je vais inclure time.h comme on l'avait fait pour notre premier projet "Plus ou Moins" :)  
Je vais aussi avoir besoin de string.h pour faire un strlen vers la fin de la fonction :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #include <string.h>  #include "dico.h" |

### La fonction piocherMot

Maintenant, attaquons la fonction qui va être chargée de piocher un mot au hasard dans le dictionnaire.  
  
Cette fonction va prendre un paramètre : un pointeur sur la zone en mémoire où elle pourra écrire le mot. Ce pointeur sera fourni par le main().  
La fonction renverra un int qui sera un booléen : 1 = tout s'est bien passé, 0 = il y a eu une erreur.  
  
Voici le début de la fonction :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5 | int piocherMot(char \*motPioche)  {  FILE\* dico = NULL; *// Le pointeur de fichier qui va contenir notre fichier*  long nombreMots = 0, numMotChoisi = 0, i = 0;  int caractereLu = 0; |

Je définis quelques variables dont je vais avoir besoin. Comme pour le main(), je n'ai pas pensé à mettre toutes ces variables dès le début, il y en a certaines que j'ai rajoutées par la suite lorsque je me suis rendu compte que j'en avais besoin ;)  
  
Grosso modo, les noms des variables parlent d'eux-mêmes. On a notre pointeur sur fichier dico dont on va se servir pour lire le fichier dico.txt, des variables temporaires qui vont stocker les caractères etc.  
Notez que j'utilise ici un int pour stocker un caractère (caractereLu) car la fonction fgetc que je vais utiliser renvoie un int. Il est donc préférable de stocker le résultat dans un int.  
  
  
Passons à la suite :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | dico = fopen("dico.txt", "r"); *// On ouvre le dictionnaire en lecture seule*  *// On vérifie si on a réussi à ouvrir le dictionnaire*  **if** (dico == NULL) *// Si on n'a PAS réussi à ouvrir le fichier*  {  printf("**\n**Impossible de charger le dictionnaire de mots");  **return** 0; *// On retourne 0 pour indiquer que la fonction a échoué*  *// A la lecture du return, la fonction s'arrête immédiatement.*  } |

Je n'ai pas grand chose à rajouter là. J'ouvre le fichier dico.txt en lecture seule ("r") et je vérifie si j'ai réussi en testant si dico vaut NULL ou pas. Si dico vaut NULL, le fichier n'a pas pu être ouvert (fichier introuvable ou occupé par un autre programme). Dans ce cas j'affiche une erreur et je fais un return 0.  
  
Pourquoi un return là ? En fait, l'instruction return commande l'arrêt de la fonction. Si le dico n'a pas pu être ouvert, la fonction s'arrête là et l'ordinateur n'ira pas lire plus loin. On retourne 0 pour indiquer au main que la fonction a échoué.  
  
Dans la suite de la fonction, on suppose donc que le fichier a été bien ouvert.  
  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | *// On compte le nombre de mots dans le fichier (il suffit de compter les*  *// entrées \n*  **do**  {  caractereLu = fgetc(dico);  **if** (caractereLu == '\n')  nombreMots++;  } **while**(caractereLu != EOF); |

Là, on parcourt tout le fichier à coups de fgetc (caractère par caractère). On compte le nombre d'\n (entrées) qu'on détecte. A chaque fois qu'on tombe sur un \n, on incrémente la variable nombreMots.  
Grâce à ce bout de code, on obtient dans nombreMots le nombre de mots dans le fichier (rappelez-vous que le fichier contient un mot par ligne).  
  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | numMotChoisi = nombreAleatoire(nombreMots); *// On pioche un mot au hasard* |

Là, je fais appel à une fonction de mon crû qui va génèrer un nombre aléatoire entre 1 et nombreMots (le paramètre qu'on envoie à la fonction).  
C'est une fonction toute simple que j'ai placée aussi dans dico.c (je vous la détaillerai tout à l'heure).  
  
Bref, elle renvoie un numéro au hasard qu'on stocke dans numMotChoisi.  
  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | *// On recommence à lire le fichier depuis le début. On s'arrête lorsqu'on est arrivés au bon mot*  rewind(dico);  **while** (numMotChoisi > 0)  {  caractereLu = fgetc(dico);  **if** (caractereLu == '\n')  numMotChoisi--;  } |

Maintenant qu'on a le numéro du mot qu'on veut piocher, on repart au début grâce à un appel à rewind()  
On parcourt là encore le fichier caractère par caractère en comptant les \n. Cette fois, on décrémente la variable numMotChoisi. Si par exemple on a choisi le mot numéro 5, à chaque entrée le variable va être décrémentée de 1.  
Elle va donc valoir 5, puis 4, 3, 2, 1... et 0.  
Lorsque la variable vaut 0, on sort du while (la condition du while numMotChoisi > 0 n'est plus remplie).  
  
Ce bout de code, que vous devez impérativement comprendre, vous montre donc comment on parcourt un fichier pour se placer à la position voulue. Ce n'est pas bien compliqué, mais ce n'est pas non plus "super évident". Assurez-vous donc de bien comprendre ce que je fais là.  
  
  
Maintenant, on devrait avoir un curseur positionné juste devant le mot secret qu'on a choisi de piocher.  
On va le stocker dans motPioche (le paramètre que la fonction reçoit) grâce à un simple fgets qui va lire le mot :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | */\* Le curseur du fichier est positionné au bon endroit.*  *On n'a plus qu'à faire un fgets qui lira la ligne \*/*  fgets(motPioche, 100, dico);  *// On vire l'\n à la fin*  motPioche[strlen(motPioche) - 1] = '\0'; |

On demande au fgets de ne pas lire plus de 100 caractères (c'est la taille du tableau motPioche, qu'on a défini dans le main).  
N'oubliez pas que fgets lit toute une ligne, y compris l'\n.  
Comme on ne veut pas garder cet \n dans le mot final, on le supprime en le remplaçant par un \0. Cela aura pour effet de couper la chaîne juste avant l'\n.  
  
  
Et... voilà qui est fait :)  
On a écrit le mot secret dans la mémoire à l'adresse de motPioche.  
  
On n'a plus qu'à fermer le fichier et à retourner 1 pour que la fonction s'arrête et pour dire que tout s'est bien passé :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4 | fclose(dico);  **return** 1; *// Tout s'est bien passé, on retourne 1*  } |

Yahou :D  
C'est tout pour la fonction piocherMot ^^

### La fonction nombreAleatoire

C'est la fonction dont j'avais promis de vous parler tout à l'heure. On tire un nombre au hasard et on le renvoie :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5 | long nombreAleatoire(long nombreMax)  {  srand(time(NULL));  **return** (rand() % nombreMax);  } |

La première ligne initialise le générateur de nombres aléatoires, comme on a appris à le faire dans le premier TP "Plus ou Moins".  
La seconde ligne prend un nombre au hasard entre 0 et nombreMax et renvoie ça (j'ai tout mis dans le return comme un gros bourrin que je suis :p )

### Le fichier dico.h

Il s'agit juste des prototypes des fonctions.  
Vous remarquerez qu'il y a la "protection" #ifndef que je vous avais demandé d'inclure dans tous vos fichiers .h (revoyez le chapitre sur le préprocesseur au besoin ;) )  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | #ifndef DEF\_DICO  #define DEF\_DICO  int piocherMot(char \*motPioche);  long nombreAleatoire(long nombreMax);  #endif |

### Le fichier dico.c

Voici le fichier dico.c en entier :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82 | */\**  *Jeu du pendu*  *Par M@teo21, pour le Site du Zér0*  *www.siteduzero.com*  *dico.c*  *------*  *Ces fonctions piochent au hasard un mot dans un fichier dictionnaire*  *pour le jeu du pendu*  *\*/*  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #include <string.h>  #include "dico.h"  char\* piocherMot()  {  FILE\* dico = NULL; *// Le pointeur de fichier qui va contenir notre fichier*  long nombreMots = 0, numMotChoisi = 0, i = 0;  char caractereLu = 0;  char motPioche[100]; *// Contiendra le mot secret pioché dans le dictionnaire*  char \*motRetourne = NULL; *// Contiendra une copie de motPioche pour retourner au main*  dico = fopen("dico.txt", "r"); *// On ouvre le dictionnaire en lecture seule*  *// On vérifie si on a réussi à ouvrir le dictionnaire*  **if** (dico == NULL) *// Si on n'a PAS réussi à ouvrir le fichier*  {  printf("**\n**Impossible de charger le dictionnaire de mots");  **return** NULL; *// On retourne NULL pour indiquer que la fonction a échoué*  *// A la lecture du return, la fonction s'arrête immédiatement.*  }  *// On compte le nombre de mots dans le fichier (il suffit de compter les*  *// entrées \n*  **for** (i = 0 ; caractereLu != EOF ; i++)  {  caractereLu = fgetc(dico);  **if** (caractereLu == '\n')  nombreMots++;  }  numMotChoisi = nombreAleatoire(nombreMots); *// On pioche un mot au hasard*  *// On recommence à lire le fichier depuis le début. On s'arrête lorsqu'on est arrivés au bon mot*  rewind(dico);  **while** (numMotChoisi > 0)  {  caractereLu = fgetc(dico);  **if** (caractereLu == '\n')  numMotChoisi--;  }  */\* Le curseur du fichier est positionné au bon endroit.*  *On n'a plus qu'à faire un scanf qui lira le mot jusqu'à l'\n \*/*  fscanf(dico, "%s", motPioche);  */\**  *Notre tableau motPioche sera supprimé de la mémoire à la fin de la fonction*  *Pour éviter de "perdre" le mot pioché, on va le copier ailleurs dans la mémoire*  *On utilise la fonction malloc pour réserver l'espace en mémoire nécessaire*  *Contrairement à la variable motPioche, notre motRetourne ne sera pas supprimé*  *de la mémoire à la fin de la fonction. Il faudra d'ailleurs penser à le supprimer*  *manuellement dans le main grâce à un free().*  *\*/*  motRetourne = malloc((strlen(motPioche) + 1) \* **sizeof**(char)); *// Réservation de la mémoire*  strcpy(motRetourne, motPioche); *// Copie de motPioche dans motRetourne*  **return** motRetourne; *// On retourne le pointeur motRetourne dans le main*  }  long nombreAleatoire(long nombreMax)  {  srand(time(NULL));  **return** (rand() % (nombreMax + 1));  } |

### Il va falloir modifier le main !

Maintenant que le fichier dico.c est prêt, on retourne dans le main() pour l'adapter un petit peu aux quelques changements qu'on vient de faire.  
  
Déjà, on commence par inclure dico.h si on veut pouvoir faire appel aux fonctions de dico.c ;)  
Ensuite, on va inclure string.h en plus là aussi car on va devoir faire un strlen :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2 | #include <string.h>  #include "dico.h" |

Pour commencer, les définitions de variables vont un peu changer. Déjà, on n'initialise plus la chaine motSecret, on crée juste un grand tableau de char (100 cases)...  
  
Quant au tableau lettreTrouvee... sa taille dépendra de la longueur du mot secret qu'on aura pioché. Comme on ne connaît pas encore cette taille, on crée un simple pointeur. Tout à l'heure, on fera un malloc et on fera pointer ce pointeur vers la zone mémoire qu'on aura allouée.  
Ceci est un exemple parfait où l'allocation dynamique est indispensable : on ne connaît pas la taille du tableau avant la compilation, on est donc obligés de créer un pointeur et de faire un malloc.  
Je ne dois pas oublier de libérer la mémoire ensuite quand je n'en ai plus besoin, d'où la présence d'un free() à la fin du main().  
  
On va aussi avoir besoin d'une variable tailleMot qui va stocker... la taille du mot ^^ En effet, si vous regardez le main() tel qu'il était dans la première partie, on supposait que le mot faisait 6 caractères partout (et c'était vrai car MARRON fait 6 lettres). Mais maintenant que le mot peut changer de taille, il va falloir être capable de s'adapter à tous les mots ;)  
  
Voici donc les définitions de variables du main en version finale :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int main(int argc, char\* argv[])  {  char lettre = 0; *// Stocke la lettre proposée par l'utilisateur (retour du scanf)*  char motSecret[100] = {0}; *// Ce sera le mot à trouver*  int \*lettreTrouvee = NULL; *// Un tableau de booléens. Chaque case correspond à une lettre du mot secret. 0 = lettre non trouvée, 1 = lettre trouvée*  long coupsRestants = 10; *// Compteur de coups restants (0 = mort)*  long i = 0; *// Une petite variable pour parcourir les tableaux*  long tailleMot = 0; |

C'est principalement le début du main() qui va changer, donc analysons-le de plus près :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2 | **if** (!piocherMot(motSecret))  exit(0); |

On fait d'abord appel à piocherMot directement dans le if.  
piocherMot va placer dans motSecret le mot qu'elle aura pioché.  
  
De plus, piocherMot va renvoyer un booléen pour nous dire si la fonction a réussi ou échoué. Le rôle du if est d'analyser ce booléen. Si ça n'a PAS marché (le "!" permet d'exprimer la négation), alors on arrête tout (*exit(0)*).  
  
  
**Code : C**

|  |  |
| --- | --- |
| 1 | tailleMot = strlen(motSecret); |

On stocke la taille du motSecret dans tailleMot comme je vous l'ai dit tout à l'heure.  
  
  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3 | lettreTrouvee = malloc(tailleMot \* **sizeof**(int)); *// On alloue dynamiquement le tableau lettreTrouvee (dont on ne connaissait pas la taille au départ)*  **if** (lettreTrouvee == NULL)  exit(0); |

Maintenant on doit allouer la mémoire pour le tableau lettreTrouvee comme je vous l'ai dit. On lui donne la taille du mot (tailleMot).  
On vérifie ensuite si le pointeur n'est pas NULL. Si c'est le cas, c'est que l'allocation a échoué. Dans ce cas, on arrête immédiatement le programme (on fait appel à exit()).  
  
Si les lignes suivantes sont lues, c'est que tout s'est bien passé.  
  
  
  
  
Voilà, grosso modo ce sont tous les préparatifs qu'il vous fallait faire ici.  
J'ai dû ensuite modifier le reste du fichier main.c pour remplacer tous les nombres **6** (l'ancienne longueur de MARRON qu'on avait fixée) par la variable tailleMot :)  
Par exemple :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2 | **for** (i = 0 ; i < tailleMot ; i++)  lettreTrouvee[i] = 0; |

Ce code met toutes les cases du tableau lettreTrouvee à 0 (en s'arrêtant lorsqu'on a parcouru *tailleMot*cases).  
  
J'ai dû aussi modifier le prototype de la fonction gagne pour ajouter la variable tailleMot. Sans cela, la fonction n'aurait pas su quand arrêter sa boucle.  
  
  
Voici le fichier main.c final en entier :  
  
**Code : C**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114 | */\**  *Jeu du pendu*  *Par M@teo21, pour le Site du Zér0*  *www.siteduzero.com*  *main.c*  *------*  *Fonctions principales de gestion du jeu*  *\*/*  #include <stdio.h>  #include <stdlib.h>  #include "dico.h"  int gagne(int lettreTrouvee[], long tailleMot);  int rechercheLettre(char lettre, char motSecret[], int lettreTrouvee[]);  int main(int argc, char\* argv[])  {  char lettre = 0; *// Stocke la lettre proposée par l'utilisateur (retour du scanf)*  char \*motSecret; *// C'est le mot à trouver*  int \*lettreTrouvee; *// Un tableau de booléens. Chaque case correspond à une lettre du mot secret. 0 = lettre non trouvée, 1 = lettre trouvée*  long coupsRestants = 10; *// Compteur de coups restants (0 = mort)*  long i = 0; *// Une petite variable pour parcourir les tableaux*  long tailleMot = 0;  printf("Bienvenue dans le Pendu !**\n\n**");  motSecret = piocherMot();  **if** (motSecret == NULL)  exit(0);  tailleMot = strlen(motSecret);  lettreTrouvee = malloc(tailleMot \* **sizeof**(int)); *// On alloue dynamiquement le tableau lettreTrouvee (dont on ne connaissait pas la taille au départ)*  **if** (lettreTrouvee == NULL)  exit(0);  **for** (i = 0 ; i < tailleMot ; i++)  lettreTrouvee[i] = 0;  *// On continue à jouer tant qu'il reste au moins un coup à jouer ou qu'on*  *// n'a pas gagné*  **while** (coupsRestants > 0 && !gagne(lettreTrouvee, tailleMot))  {  printf("**\n\n**Il vous reste %ld coups a jouer", coupsRestants);  printf("**\n**Quel est le mot secret ? ");  */\* On affiche le mot secret en masquant les lettres non trouvées*  *Exemple : \*A\*\*ON \*/*  **for** (i = 0 ; i <= tailleMot ; i++)  {  **if** (lettreTrouvee[i]) *// Si on a trouvé la lettre n°i*  printf("%c", motSecret[i]); *// On l'affiche*  **else**  printf("\*"); *// Sinon, on affiche une étoile pour les lettres non trouvées*  }  printf("**\n**Proposez une lettre : ");  scanf("%c%\*[^**\n**]", &lettre);  getchar();  *// Si ce n'était PAS la bonne lettre*  **if** (!rechercheLettre(lettre, motSecret, lettreTrouvee))  {  coupsRestants--; *// On enlève un coup au joueur*  }  }  **if** (gagne(lettreTrouvee, tailleMot))  printf("**\n\n**Gagne ! Le mot secret etait bien : %s", motSecret);  **else**  printf("**\n\n**Perdu ! Le mot secret etait : %s", motSecret);  free(motSecret); *// On libère la mémoire là où se trouve le mot qui a été créé par le malloc*  **return** 0;  }  int gagne(int lettreTrouvee[], long tailleMot)  {  long i = 0;  int joueurGagne = 1;  **for** (i = 0 ; i <= tailleMot ; i++)  {  **if** (lettreTrouvee[i] == 0)  joueurGagne = 0;  }  **return** joueurGagne;  }  int rechercheLettre(char lettre, char motSecret[], int lettreTrouvee[])  {  long i = 0;  int bonneLettre = 0;  *// On parcourt motSecret pour vérifier si la lettre proposée y est*  **for** (i = 0 ; motSecret[i] != '\0' ; i++)  {  **if** (lettre == motSecret[i]) *// Si la lettre y est*  {  bonneLettre = 1; *// On mémorise que c'était une bonne lettre*  lettreTrouvee[i] = 1; *// On met à 1 le case du tableau de booléens correspondant à la lettre actuelle*  }  }  **return** bonneLettre;  } |

## Idées d'amélioration

### Télécharger le projet

Pour commencer, je vous invite à télécharger le projet de Pendu que j'ai fait :

[Télécharger le projet Pendu (10 Ko)](http://www.siteduzero.com/uploads/fr/ftp/mateo21/pendu_siteduzero.zip)

Si vous êtes sous Linux ou Mac, supprimez le fichier dico.txt et recréez-vous en un. Les fichiers sont enregistrés de manière différente sous Windows, donc si vous utilisez le mien vous risquez d'avoir des bugs.  
N'oubliez pas qu'il faut qu'il y ait une Entrée après chaque mot du dictionnaire. Pensez en particulier à mettre une Entrée après le dernier mot de la liste.

Ca va vous permettre de tester par vous-mêmes le fonctionnement du projet, de faire éventuellement des améliorations par-dessus etc etc. Bien entendu, le mieux serait que vous ayez déjà réussi le pendu par vous-mêmes et que vous n'ayez même pas besoin de voir mon projet pour voir comment j'ai fait mais... je suis réaliste, je sais que ce TP a dû être assez délicat pour bon nombre d'entre vous ;)  
  
Vous trouverez dans ce zip les fichiers .c et .h ainsi que le fichier .cbp du projet. C'est un projet fait sous Code::Blocks oui :)  
Si vous utilisez un autre IDE, pas de panique. Vous créez un nouveau projet console et vous y ajoutez manuellement les .c et .h que vous trouverez dans le zip.  
  
Vous trouverez aussi l'exécutable (.exe Windows) ainsi qu'un petit dictionnaire (dico.txt) pour commencer ;)

### Améliorez le pendu !

Mine de rien, le pendu est déjà assez évolué comme ça ^^  
On a un jeu qui lit un fichier de dictionnaire et qui prend à chaque fois un mot au hasard.  
  
Voici quand même quelques idées d'amélioration qui me passent par la tête :

* Actuellement, on ne vous propose de jouer qu'une fois. Il serait bien de pouvoir reboucler à la fin du main pour **faire une nouvelle partie** si le joueur le désire
* Vous pourriez créer **un mode 2 joueurs** dans lequel le premier joueur rentre un mot que le deuxième joueur doit deviner
* Ce n'est pas utile (donc c'est indispensable) : pourquoi ne pas **dessiner un bonhomme qui se pend** à chaque fois que l'on fait une erreur ?  
  A coups de printf bien sûr, on est en console rappelez-vous :p

Prenez bien le temps de comprendre ce jeu de Pendu et améliorez-le jusqu'au maximum. Il faut que vous soyez capables de refaire ce petit jeu de Pendu les yeux fermés !  
  
Allez courage :)