

Apprenez à programmer en C !

Par Mathieu Nebra (Mateo21)



OPENCLASSROOMS

www.openclassrooms.com

Licence Creative Commons 6 2.0
Dernière mise à jour le 8/01/2013

Sommaire

Sommaire	2
Lire aussi	6
Apprenez à programmer en C !	8
Partie 1 : Les bases de la programmation en C	8
Vous avez dit programmer ?	9
Programmer, c'est quoi ?	9
Programmer, dans quel langage ?	10
Un peu de vocabulaire	10
Pourquoi choisir d'apprendre le C ?	11
Programmer, c'est dur ?	12
Ayez les bons outils !	13
Les outils nécessaires au programmeur	14
Choisissez votre IDE	14
Code::Blocks (Windows, Mac OS, Linux)	15
Télécharger Code::Blocks	15
Créer un nouveau projet	17
Visual C++ (Windows seulement)	19
Installation	20
Créer un nouveau projet	20
Ajouter un nouveau fichier source	23
La fenêtre principale de Visual	24
Xcode (Mac OS seulement)	25
Xcode, où es-tu ?	25
Lancement de Xcode	25
La fenêtre de développement	26
Ajouter un nouveau fichier	27
En résumé	28
Votre premier programme	28
Console ou fenêtre ?	29
Les programmes en fenêtres	29
Les programmes en console	30
Un minimum de code	31
Demandez le code minimal à votre IDE	31
Analysons le code minimal	32
Testons notre programme	34
Écrire un message à l'écran	35
Dis Bonjour au Monsieur	36
Les caractères spéciaux	37
Le syndrome de Gérard	38
Les commentaires, c'est très utile !	39
Un monde de variables	41
Une affaire de mémoire	41
Les différents types de mémoire	41
La mémoire vive en photos	42
Le schéma de la mémoire vive	43
Déclarer une variable	45
Donner un nom à ses variables	45
Les types de variables	45
Déclarer une variable	47
Affecter une valeur à une variable	48
La valeur d'une nouvelle variable	49
Les constantes	50
Afficher le contenu d'une variable	50
Afficher plusieurs variables dans un même printf	52
Récupérer une saisie	52
Une bête de calcul	55
Les calculs de base	55
La division	56
Le modulo	57
Des calculs entre variables	57
Les raccourcis	58
L'incréméntation	59
La décréméntation	59
Les autres raccourcis	60
La bibliothèque mathématique	60
fabs	61
ceil	62
floor	62
pow	62
sqrt	62
sin, cos, tan	63
asin, acos, atan	63
exp	63
log	63
log10	63

En résumé	63
Les conditions	63
La condition if... else	64
Quelques symboles à connaître	64
Un if simple	64
Le else pour dire « sinon »	66
Le else if pour dire « sinon si »	67
Plusieurs conditions à la fois	67
Quelques erreurs courantes de débutant	69
Les booléens, le cœur des conditions	69
Quelques petits tests pour bien comprendre	69
Des explications s'imposent	70
Un test avec une variable	70
Cette variable majeur est un booléen	71
Les booléens dans les conditions	71
La condition switch	72
Construire un switch	72
Gérer un menu avec un switch	73
Les ternaires : des conditions condensées	75
Une condition if... else bien connue	75
La même condition en ternaire	75
En résumé	76
Les boucles	76
Qu'est-ce qu'une boucle ?	77
La boucle while	77
Attention aux boucles infinies	79
La boucle do... while	80
La boucle for	80
En résumé	81
TP : Plus ou Moins, votre premier jeu	82
Préparatifs et conseils	82
Le principe du programme	82
Tirer un nombre au sort	82
Les bibliothèques à inclure	83
J'en ai assez dit !	83
Correction !	84
La correction de « Plus ou Moins »	84
Exécutable et sources	84
Explications	85
Idées d'amélioration	85
Les fonctions	86
Créer et appeler une fonction	87
Schéma d'une fonction	88
Créer une fonction	88
Plusieurs paramètres, aucun paramètre	89
Appeler une fonction	90
Des exemples pour bien comprendre	93
Conversion euros / francs	93
La punition	94
Aire d'un rectangle	95
Un menu	96
En résumé	97
Partie 2 : Techniques « avancées » du langage C	98
La programmation modulaire	98
Les prototypes	98
Le prototype pour annoncer une fonction	98
Les headers	99
Plusieurs fichiers par projet	99
Fichiers .h et .c	100
Les include des bibliothèques standard	103
La compilation séparée	104
La portée des fonctions et variables	106
Les variables propres aux fonctions	106
Les variables globales : à éviter	107
Variable statique à une fonction	108
Les fonctions locales à un fichier	109
En résumé	109
À l'assaut des pointeurs	111
Un problème bien ennuyeux	111
La mémoire, une question d'adresse	112
Rappel des faits	112
Adresse et valeur	113
Le scoop du jour	114
Utiliser des pointeurs	115
Créer un pointeur	115
À retenir absolument	118
Envoyer un pointeur à une fonction	119
Une autre façon d'envoyer un pointeur à une fonction	120
Qui a dit : "Un problème bien ennuyeux" ?	122
En résumé	123
Les tableaux	123

Les tableaux dans la mémoire	124
Définir un tableau	124
Les tableaux à taille dynamique	126
Parcourir un tableau	127
Initialiser un tableau	128
Une autre façon d'initialiser	128
Passage de tableaux à une fonction	129
Quelques exercices !	130
En résumé	132
Les chaînes de caractères	133
Le type char	133
Afficher un caractère	133
Les chaînes sont des tableaux de char	134
Création et initialisation de la chaîne	136
Récupération d'une chaîne via un scanf	138
Fonctions de manipulation des chaînes	138
Pensez à inclure string.h	139
strlen : calculer la longueur d'une chaîne	139
strcpy : copier une chaîne dans une autre	140
strcat : concaténer 2 chaînes	142
strcmp : comparer 2 chaînes	143
strchr : rechercher un caractère	144
strpbrk : premier caractère de la liste	145
strstr : rechercher une chaîne dans une autre	146
sprintf : écrire dans une chaîne	147
En résumé	148
Le préprocesseur	148
Les include	149
Les define	150
Un define pour la taille des tableaux	152
Calculs dans les define	153
Les constantes prédéfinies	153
Les définitions simples	153
Les macros	154
Macro sans paramètres	154
Macro avec paramètres	155
Les conditions	157
#ifdef, #ifndef	157
#ifdef pour éviter les inclusions infinies	158
En résumé	159
Créez vos propres types de variables	159
Définir une structure	160
Exemple de structure	160
Tableaux dans une structure	161
Utilisation d'une structure	162
Le typedef	162
Modifier les composantes de la structure	163
Initialiser une structure	164
Pointeur de structure	165
Envoi de la structure à une fonction	166
Un raccourci pratique et très utilisé	167
Les énumérations	168
Association de nombres aux valeurs	168
Associer une valeur précise	169
En résumé	169
Lire et écrire dans des fichiers	170
Ouvrir et fermer un fichier	171
fopen : ouverture du fichier	171
Tester l'ouverture du fichier	175
fclose : fermer le fichier	176
Différentes méthodes de lecture / écriture	177
Écrire dans le fichier	177
Lire dans un fichier	180
Se déplacer dans un fichier	184
ftell : position dans le fichier	184
fseek : se positionner dans le fichier	184
rewind : retour au début	185
Renommer et supprimer un fichier	186
rename : renommer un fichier	186
remove : supprimer un fichier	186
L'allocation dynamique	187
La taille des variables	188
Une nouvelle façon de voir la mémoire	190
Allocation de mémoire dynamique	193
malloc : demande d'allocation de mémoire	193
Tester le pointeur	194
free : libérer de la mémoire	194
Exemple concret d'utilisation	195
Allocation dynamique d'un tableau	196
En résumé	198
TP : réalisation d'un Pendu	199
Les consignes	199

Déroulement d'une partie	199
Dictionnaire de mots	202
La solution (1 : le code du jeu)	204
Analyse de la fonction main	204
Analyse de la fonction gagne	207
Analyse de la fonction rechercheLettre	207
La solution (2 : la gestion du dictionnaire)	208
Préparation des nouveaux fichiers	209
La fonction piocherMot	209
La fonction nombreAleatoire	211
Le fichier dico.h	212
Le fichier dico.c	212
Il va falloir modifier le main !	213
Idées d'amélioration	217
Télécharger le projet	217
Améliorez le Pendu !	217
La saisie de texte sécurisée	218
Les limites de la fonction scanf	219
Entrer une chaîne de caractères avec des espaces	219
Entrer une chaîne de caractères trop longue	220
Récupérer une chaîne de caractères	221
La fonction fgets	221
Créer sa propre fonction de saisie utilisant fgets	222
Convertir la chaîne en nombre	227
strtol : convertir une chaîne en long	227
strtod : convertir une chaîne en double	229
En résumé	229
Partie 3 : Création de jeux 2D en SDL	230
Installation de la SDL	231
Pourquoi avoir choisi la SDL ?	231
Choisir une bibliothèque : pas facile !	231
La SDL est un bon choix !	231
Les possibilités offertes par la SDL	232
Téléchargement de la SDL	233
Créer un projet SDL : Windows	234
Création d'un projet SDL sous Code::Blocks	234
Création d'un projet SDL sous Visual C++	238
Créer un projet SDL : Mac OS (Xcode)	240
Créer un projet SDL : Linux	244
En résumé	245
Création d'une fenêtre et de surfaces	246
Charger et arrêter la SDL	246
SDL_Init : chargement de la SDL	246
SDL_Quit : arrêt de la SDL	247
Canevas de programme SDL	248
Gérer les erreurs	248
Ouverture d'une fenêtre	249
Choix du mode vidéo	250
Mettre en pause le programme	251
Changer le titre de la fenêtre	253
Manipulation des surfaces	255
Votre première surface : l'écran	255
Colorer une surface	257
Dessiner une nouvelle surface à l'écran	261
Centrer la surface à l'écran	265
Exercice : créer un dégradé	266
Correction !	267
« Je veux des exercices pour m'entraîner ! »	269
En résumé	269
Afficher des images	269
Charger une image BMP	270
Le format BMP	270
Charger un Bitmap	270
Associer une icône à son application	272
Gestion de la transparence	273
Le problème de la transparence	273
Rendre une image transparente	275
La transparence Alpha	277
Charger plus de formats d'image avec SDL_Image	279
Installer SDL_image sous Windows	280
Installer SDL_image sous Mac OS X	282
Charger les images	282
En résumé	284
La gestion des événements	284
Le principe des événements	285
La variable d'événement	285
La boucle des événements	286
Récupération de l'événement	286
Analyse de l'événement	287
Le code complet	288
Le clavier	289
Les événements du clavier	289

Récupérer la touche	289
Exercice : diriger Zozor au clavier	291
Charger l'image	291
Schéma de la programmation événementielle	292
Traiter l'événement SDL_KEYDOWN	293
Quelques optimisations	295
La souris	298
Gérer les clics de la souris	299
Gérer le déplacement de la souris	301
Quelques autres fonctions avec la souris	302
Les événements de la fenêtre	303
Redimensionnement de la fenêtre	303
Visibilité de la fenêtre	303
En résumé	305
TP : Mario Sokoban	306
Cahier des charges du Sokoban	306
À propos du Sokoban	306
Le cahier des charges	307
Récupérer les sprites du jeu	307
Le main et les constantes	309
Les différents fichiers du projet	309
Les constantes : constantes.h	309
Le main : main.c	311
Le jeu	312
Les paramètres envoyés à la fonction	312
Les déclarations de variables	313
Initialisations	316
La boucle principale	318
Fin de la fonction : déchargements	322
La fonction deplacerJoueur	322
Chargement et enregistrement de niveaux	325
chargerNiveau	325
sauvegarderNiveau	327
L'éditeur de niveaux	328
Initialisations	328
La gestion des événements	329
Blit time !	332
Résumé et améliorations	333
Alors résumons !	333
Améliorez !	334
Maîtrisez le temps !	335
Le Delay et les Ticks	335
SDL_Delay	335
SDL_GetTicks	336
Utiliser SDL_GetTicks pour gérer le temps	337
Les timers	342
Initialiser le système de timers	342
Ajouter un timer	342
En résumé	344
Écrire du texte avec SDL_ttf	345
Installer SDL_ttf	346
Comment fonctionne SDL_ttf ?	346
Installer SDL_ttf	346
Configurer un projet pour SDL_ttf	346
La documentation	347
Chargement de SDL_ttf	347
L'include	347
Démarrage de SDL_ttf	347
Arrêt de SDL_ttf	348
Chargement d'une police	348
Les différentes méthodes d'écriture	350
Exemple d'écriture de texte en Blended	352
Code complet d'écriture de texte	353
Attributs d'écriture du texte	355
Exercice : le compteur	356
En résumé	359
Jouer du son avec FMOD	360
Installer FMOD	360
Pourquoi FMOD ?	360
Télécharger FMOD	360
Installer FMOD	361
Initialiser et libérer un objet système	361
Inclure le header	361
Créer et initialiser un objet système	362
Fermer et libérer un objet système	363
Les sons courts	363
Trouver des sons courts	363
Les étapes à suivre pour jouer un son	364
Exemple : un jeu de tir	366
Les musiques (MP3, OGG, WMA...)	369
Trouver des musiques	369
Les étapes à suivre pour jouer une musique	370
Code complet de lecture du MP3	372

En résumé	374
TP : visualisation spectrale du son	375
Les consignes	375
1/ Lire un MP3	375
2/ Récupérer les données spectrales du son	376
4/ Réaliser le dégradé	378
La solution	379
Idées d'amélioration	383
Partie 4 : Les structures de données	385
Les listes chaînées	385
Représentation d'une liste chaînée	385
Construction d'une liste chaînée	386
Un élément de la liste	386
La structure de contrôle	386
Le dernier élément de la liste	387
Les fonctions de gestion de la liste	388
Initialiser la liste	388
Ajouter un élément	389
Supprimer un élément	391
Afficher la liste chaînée	392
Aller plus loin	393
En résumé	393
Les piles et les files	394
Les piles	394
Fonctionnement des piles	394
Création d'un système de pile	396
Les files	400
Fonctionnement des files	400
Création d'un système de file	400
À vous de jouer !	402
En résumé	402
Les tables de hachage	404
Pourquoi utiliser une table de hachage ?	404
Qu'est-ce qu'une table de hachage ?	404
Écrire une fonction de hachage	406
Gérer les collisions	407
L'adressage ouvert	408
Le chaînage	408
En résumé	408



Apprenez à programmer en C !



Par [Mathieu Nebra \(Mateo21\)](#)

Mise à jour : 08/01/2013

Difficulté : Intermédiaire Durée d'étude : 2 mois, 15 jours



1 visites depuis 7 jours, classé 26/807

Vous aimeriez apprendre à programmer, mais vous ne savez pas par où commencer ?
(autrement dit : vous en avez marre des cours trop compliqués que vous ne comprenez pas ? 🤔)

C'est votre jour de chance ! 😊

Vous venez de tomber sur un cours de programmation pour débutants, vraiment pour débutants.

Il n'y a aucune honte à être débutant, tout le monde est passé par là, moi y compris. 😊

Ce qu'il vous faut est pourtant simple. Il faut qu'on vous explique tout, progressivement, depuis le début :

- Comment s'y prend-on pour créer des programmes comme des jeux, des fenêtres ?
- De quels logiciels a-t-on besoin pour programmer ?
- Dans quel langage commencer à programmer ? D'ailleurs, c'est quoi un langage ? 🤔

Ce tutoriel est constitué de 2 parties théoriques sur le langage C ([parties I et II](#)) suivies d'une partie pratique ([partie III](#)) portant sur la bibliothèque SDL dans laquelle vous réutiliserez tout ce que vous avez appris pour créer des jeux vidéo !



Exemples de réalisations tirés de la partie III sur la SDL



Ce cours vous plaît ?

Si vous avez aimé ce cours, vous pouvez retrouver le livre "[Apprenez à programmer en C](#)" du même auteur, en vente [sur le Site du Zéro](#), en [librairie](#) et dans les boutiques en ligne comme [Amazon.fr](#) et [FNAC.com](#). Vous y trouverez ce cours adapté au format papier avec une série de chapitres inédits.

[Plus d'informations](#)

Partie 1 : Les bases de la programmation en C

Vous avez dit programmer ?

Vous avez déjà entendu parler de programmation et nul doute que si vous avez ce livre entre les mains, c'est parce que vous voulez « enfin » comprendre comment ça fonctionne.

Mais programmer en langage C... ça veut dire quoi ? Est-ce que c'est bien pour commencer ? Est-ce que vous avez le niveau pour programmer ? Est-ce qu'on peut tout faire avec ?

Ce chapitre a pour but de répondre à toutes ces questions apparemment bêtes et pourtant très importantes. Grâce à ces questions simples, vous saurez à la fin de ce premier chapitre ce qui vous attend. C'est quand même mieux de savoir à quoi sert ce que vous allez apprendre, vous ne trouvez pas ?

Programmer, c'est quoi ?

On commence par la question la plus simple qui soit, la plus basique de toutes les questions basiques. 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 » ?

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.



Clone de Metal Slug réalisé par le membre joe78

En bref, les programmes sont partout et permettent de faire a 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).

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 demande beaucoup de travail.

Comme vous débutez, nous n'allons pas commencer en réalisant un jeu 3D. Ce serait suicidaire.

Nous allons devoir passer par des programmes 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 transcendant, mais rien que ça croyez-moi, ce n'est pas aussi facile que ça en a l'air.

Ça impressionne moins les amis, 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 en sortir dans n'importe quel programme écrit en C.

Mais tenez, au fait, vous savez ce que c'est vous, ce fameux « langage C » ?

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 (j'invente, je ne connais quand même pas la traduction informatique par cœur) :

```
0010110110010011010011110
```

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.

Donc voilà notre premier vrai problème :

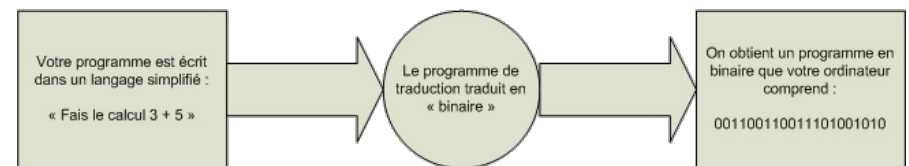


Comment parler à l'ordinateur plus simplement qu'en binaire avec des 0 et des 1 ?

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 !). 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 ».

Le schéma suivante résume ce que je viens de vous expliquer.



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 à utiliser un vocabulaire approprié.

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 si vous étiez des extra-terrestres, mais ça il ne faudra pas y faire attention !

Reprenons le schéma que l'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, 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.

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

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).

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**.

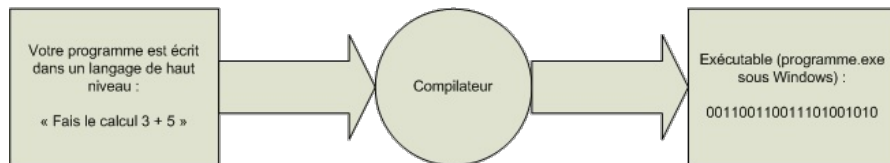
Très important : il existe un compilateur différent pour chaque langage de haut niveau. C'est d'ailleurs tout à fait logique : les langages étant différents, on ne traduit pas le C de la même manière qu'on traduit le Delphi.



Vous verrez par la suite que même pour le langage C il existe plusieurs compilateurs différents ! Il y a le compilateur écrit par Microsoft, le compilateur GNU, etc. On verra tout cela 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 précédent, et utilisons cette fois des vrais mots tordus d'informaticien (fig. suivante).



Pourquoi choisir d'apprendre le 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, vous allez en apprendre beaucoup plus sur la programmation et sur le fonctionnement de votre ordinateur. Vous serez ensuite largement capables d'apprendre un autre langage de programmation si vous le désirez. Vous serez donc plus autonomes.

Par ailleurs, le C est un langage très populaire. Il est utilisé pour programmer une grande partie des logiciels que vous connaissez.

Enfin, le langage C est un des langages les plus connus et les plus utilisés qui existent. Il est très fréquent qu'il soit enseigné lors d'études supérieures en informatique.

Voilà 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.



On pourrait citer d'autres raisons : certains langages de programmation sont plus destinés au Web (comme PHP) qu'à la réalisation de programmes informatiques.

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 pourra pas vous faire de mal de reprendre à zéro.



Il y a quelque chose que je ne comprends pas... Quelle est la différence entre le langage « C » et cet autre langage dont on parle, le langage « C++ » ?

Le langage C et le langage C++ sont très similaires. Ils sont tous les deux toujours très utilisés. Pour bien comprendre comment ils sont nés, il faut faire un peu d'histoire.

- 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**.
- Les choses évoluant, on a créé un nouveau langage appelé le **CPL**, qui évolua lui-même en **BCPL**, qui prit ensuite le nom de **langage B**.
- Puis un beau jour, on en est arrivé à créer un autre langage encore, qu'on a appelé... le **langage C**. Ce langage, s'il a subi quelques modifications, reste encore un des 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 permettant de programmer d'une façon différente.



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. Disons aussi qu'il permet au final de programmer un peu plus efficacement et de mieux hiérarchiser le code de son programme. Malgré tout, il ressemble beaucoup au C. Si vous voulez passer au C++ par la suite, cela vous sera facile.

Ce n'est PAS parce que le C++ est une « évolution » du C qu'il faut absolument faire du C++ pour réaliser des programmes. 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 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 deux manières de programmer assez différentes.

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 quatre opérations de base :

- l'addition ;
- la soustraction ;
- la multiplication ;
- la division.

Ce n'est pas trop intimidant, avouez ! Je vous expliquerai dans un prochain chapitre comment l'ordinateur réalise ces opérations de base dans vos programmes.

Bref, niveau maths, il n'y a pas de difficulté insurmontable. En fait, tout dépend du programme que vous allez réaliser : 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. Mais pour apprendre le langage C lui-même, 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 en C. De ce point de vue-là, rassurez-vous, je vous apprendrai tout au fur et à mesure.

Notez qu'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 forts en maths certes, mais ça ne vous empêchera pas d'avoir à réfléchir. Désolé pour ceux qui pensaient que ça allait tomber tout cuit sans effort !
- **le calme** : non, on ne tape pas sur son ordinateur avec un marteau. 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 d'ailleurs beaucoup qui découvrent qu'ils adorent ça !

En résumé

- Pour réaliser des programmes informatiques, on doit écrire dans un **langage** que l'ordinateur « comprend ».
- Il existe de nombreux langages informatiques que l'on peut classer par niveau. Les langages dits de « haut niveau » sont parfois plus faciles à maîtriser au détriment souvent d'une perte de performances dans le programme final.
- Le **langage C** que nous allons étudier dans ce livre est considéré comme étant de bas niveau. C'est un des langages de programmation les plus célèbres et les plus utilisés au monde.
- Le **code source** est une série d'instructions écrites dans un langage informatique.
- Le **compilateur** est un programme qui transforme votre code source en **code binaire**, qui peut alors être exécuté par votre processeur. Les `.exe` que l'on connaît sont des programmes binaires, il n'y a plus de code source à l'intérieur.
- La programmation ne requiert pas en elle-même de connaissances mathématiques poussées (sauf dans quelques cas précis où votre application doit faire appel à des formules mathématiques, comme c'est le cas des logiciels de cryptage). Néanmoins, il est nécessaire d'avoir un bon sens de la logique et d'être méthodique.



Ayez les bons outils !

Après un premier chapitre plutôt introductif, nous commençons à entrer dans le vif du sujet. Nous allons répondre à la question suivante : « De quels logiciels a-t-on besoin pour programmer ? ».

Il n'y aura rien de difficile à faire dans ce chapitre, on va prendre le temps de se familiariser avec de nouveaux logiciels.

Profitez-en ! Dans le chapitre suivant, nous commencerons à vraiment programmer et il ne sera plus l'heure de faire la sieste !

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

A priori, si vous êtes aventuriers, vous pouvez vous passer de débogueur. Mais bon, je sais pertinemment que vous ne tarderez pas à en avoir besoin. :-)

À partir de maintenant on a deux possibilités :

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

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

Choisissez votre IDE

Il m'a semblé intéressant de vous montrer quelques 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 le jour.

- Un des IDE que je préfère s'appelle **Code::Blocks**. Il est gratuit et fonctionne sur la plupart des systèmes d'exploitation. Je conseille d'utiliser celui-ci pour débiter (et même pour la suite s'il vous plaît bien !). Fonctionne sous Windows, Mac et Linux.
- Le plus célèbre IDE sous Windows, c'est celui de Microsoft : **Visual C++**. Il existe à la base en version payante (chère !), mais il existe heureusement une version gratuite intitulée **Visual C++ Express** qui est vraiment très bien (il y a peu de différences avec la version payante). Il est très complet et possède un puissant module de correction des erreurs (débogage). Fonctionne sous Windows uniquement.
- Sur Mac OS X, vous pouvez utiliser Xcode, généralement fourni sur le CD d'installation de Mac OS X. C'est un IDE très apprécié par tous ceux qui font de la programmation sur Mac. Fonctionne sous Mac OS X uniquement.



Note pour les utilisateurs de Linux : il existe de nombreux IDE sous Linux, mais les programmeurs expérimentés préfèrent parfois se passer d'IDE et compiler « à la main », ce qui est un peu plus difficile. En ce qui nous concerne



nous allons commencer par utiliser un IDE. Je vous conseille d'installer Code::Blocks si vous êtes sous Linux, pour suivre mes explications.



Quel est le meilleur de tous ces IDE ?

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

Tout au long de tout ce cours, j'utiliserai Code::Blocks. Si vous voulez obtenir exactement les mêmes écrans que moi, surtout pour ne pas être perdus au début, je vous recommande donc de commencer par installer Code::Blocks.

Code::Blocks (Windows, Mac OS, Linux)

Code::Blocks est un IDE libre et gratuit, disponible pour Windows, Mac et Linux.

Code::Blocks n'est disponible pour le moment qu'en anglais. Cela ne devrait PAS vous repousser à l'utiliser. Croyez-moi, nous aurons quoi qu'il en soit peu affaire aux menus : c'est le langage C qui nous intéresse.

Sachez toutefois que quand vous programmerez, vous serez de toute façon confrontés bien souvent à des documentations en anglais. Voilà donc une raison de plus pour s'entraîner à utiliser cette langue.

Télécharger Code::Blocks

Rendez-vous sur la page de téléchargements de Code::Blocks.

- 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-10.05mingw-setup.exe). L'autre version étant sans compilateur, vous auriez eu du mal à compiler vos programmes !
- Si vous êtes sous Linux, choisissez le package qui correspond à votre distribution.
- Enfin, sous Mac, choisissez le fichier le plus récent de la liste. Ex : codeblocks-10.05-p2-mac.zip.



J'insiste là-dessus : si vous êtes sous Windows, téléchargez la version incluant mingw dans le nom du programme d'installation. Si vous prenez la mauvaise version, vous ne pourrez pas compiler vos programmes par la suite !

L'installation est très simple et rapide. Laissez toutes les options par défaut et lancez le programme. Vous devriez voir une fenêtre similaire à la fig. suivante.



Windows 2000 / XP / Vista / 7 :

File	Date	Size	Download from
codeblocks-10.05-setup.exe	27 May 2010	23.3 MB	Berlios
codeblocks-10.05mingw-setup.exe	27 May 2010	74.0 MB	Berlios

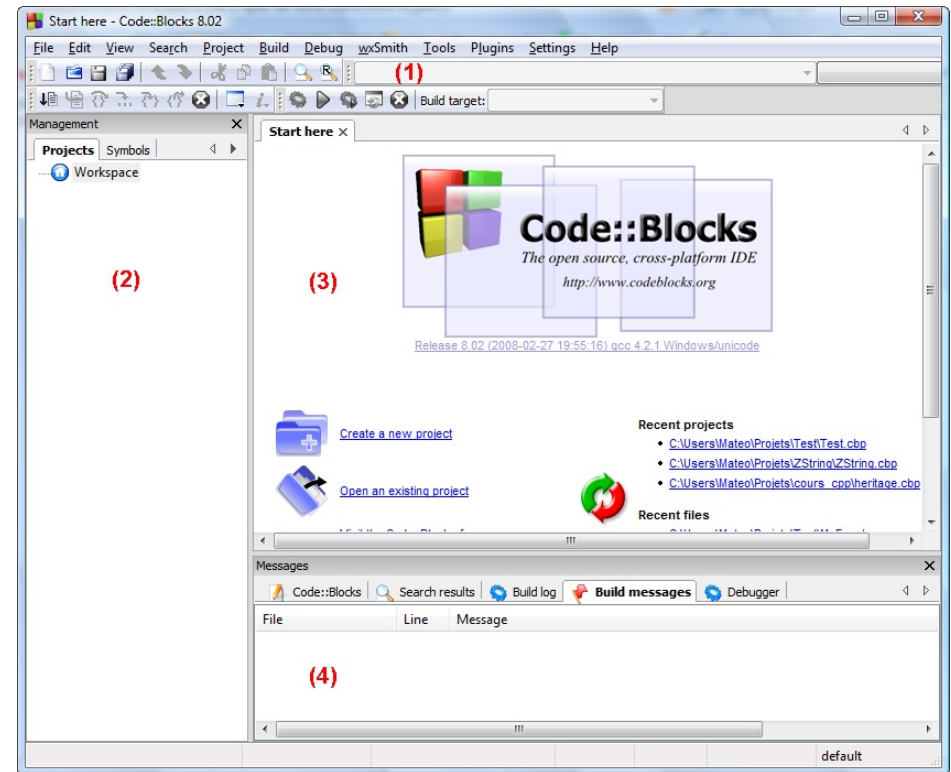
NOTE: The codeblocks-10.05mingw-setup.exe file includes the GCC compiler and GDB debugger from MinGW.

On distingue 4 grandes sections dans la fenêtre, numérotées sur l'image :

1. **la barre d'outils** : elle comprend de nombreux boutons, mais seuls quelques-uns nous seront régulièrement utiles. J'y reviendrai plus loin ;
2. **la liste des fichiers du projet** : c'est à gauche que s'affiche la liste de tous les fichiers source de votre programme. Notez

que sur cette capture aucun projet n'a été créé, on ne voit donc pas encore de fichiers à l'intérieur de la liste. Vous verrez cette section se remplir dans cinq minutes en lisant la suite du cours ;

3. **la zone principale** : c'est là que vous pourrez écrire votre code en langage C ;
4. **la zone de notification** : aussi appelée la « zone de la mort », c'est ici que vous verrez les erreurs de compilation s'afficher si votre code comporte des erreurs. Cela arrive très régulièrement !



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

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

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

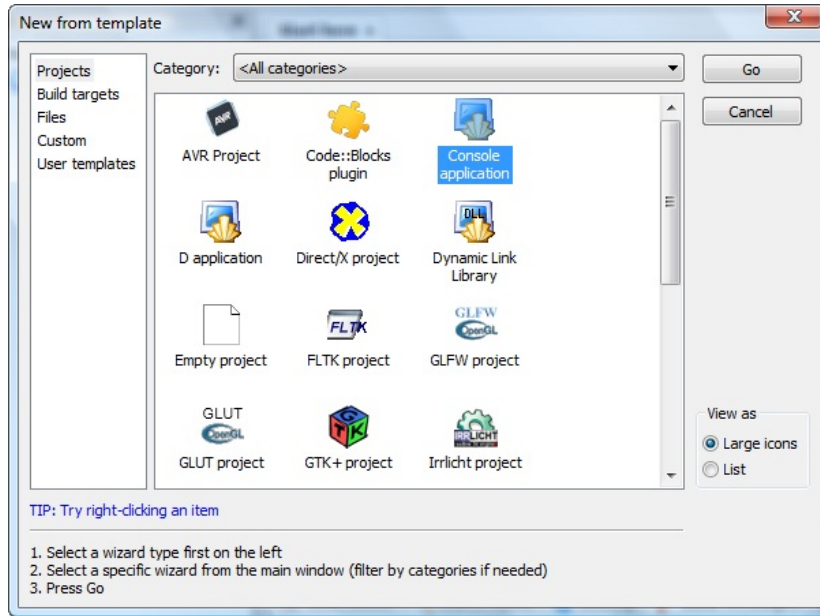
fonctionnement de la compilation dans un chapitre futur. Pour l'instant, on se contente de savoir le minimum nécessaire pour ne pas tout mélanger. Ce bouton ne nous sera donc pas utile de suite.



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

Créer un nouveau projet

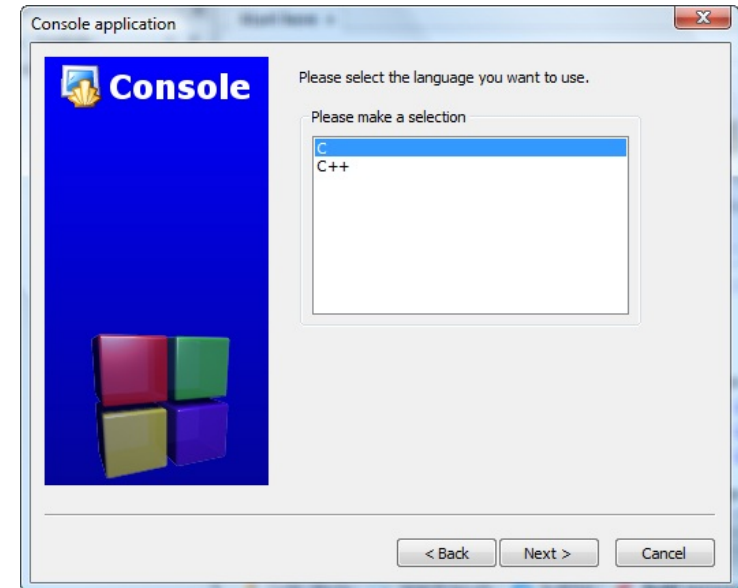
Pour créer un nouveau projet, c'est très simple : allez dans le menu File / New / Project. Dans la fenêtre qui s'ouvre (fig. suivante), choisissez Console application.



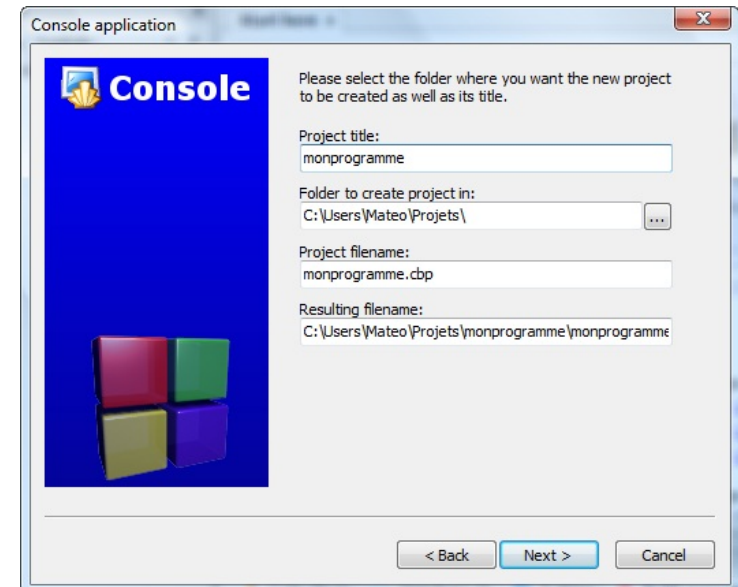
Comme vous pouvez le voir, Code::Blocks propose de réaliser pas mal de types de programmes différents qui utilisent des bibliothèques connues comme la SDL (2D), OpenGL (3D), Qt et wxWidgets (fenêtres), etc. Pour l'instant, ces icônes servent plutôt à faire joli car les bibliothèques ne sont pas installées 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, cette première page ne servant à rien.

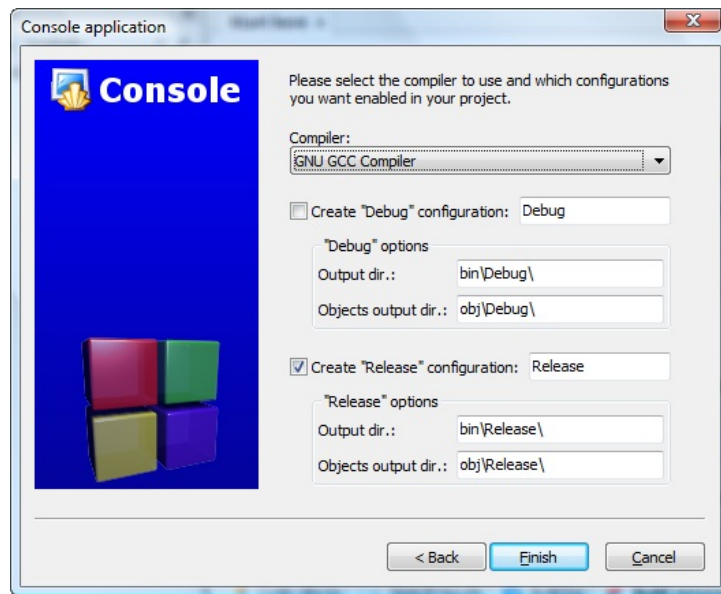
On vous demande ensuite si vous allez faire du C ou du C++ (fig. suivante) : répondez « C ».



On vous demande le nom de votre projet (fig. suivante) et dans quel dossier les fichiers source seront enregistrés.



Enfin, la dernière page (fig. suivante) vous permet de choisir de quelle façon le programme doit être compilé. Vous pouvez laisser les options par défaut, ça n'aura pas d'incidence sur ce que nous allons faire dans l'immédiat (veillez à ce que la case Debug ou Release au moins soit cochée).



Cliquez sur **Finish**, c'est bon !

Code::Blocks vous créera un premier projet avec déjà un tout petit peu de code source dedans.

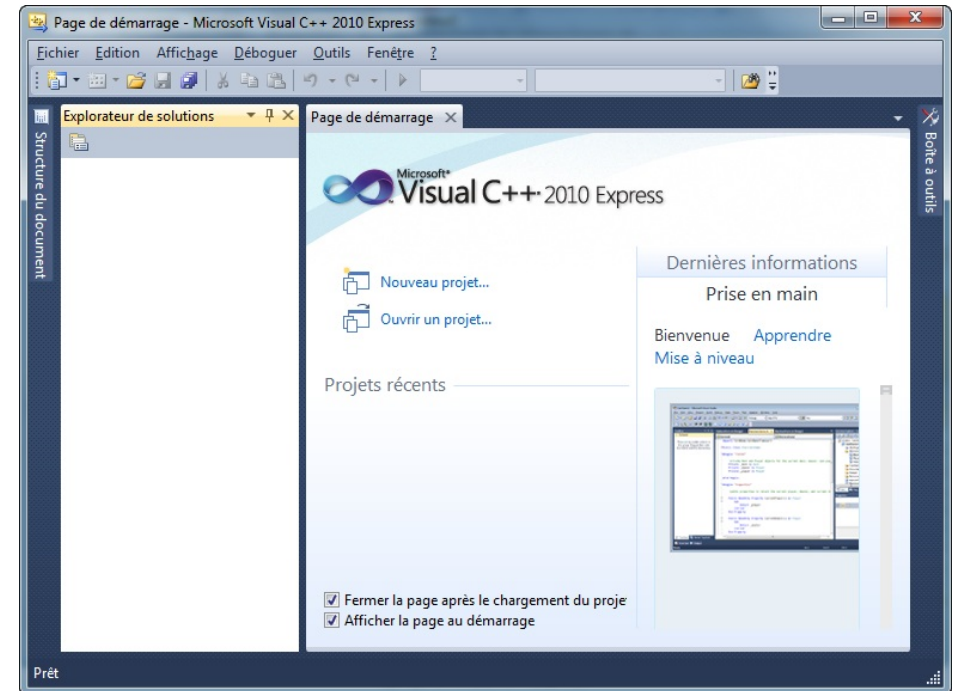
Dans le cadre de gauche « Projects », développez l'arborescence en cliquant sur le petit « + » pour afficher la liste des fichiers du projet. Vous devriez avoir au moins un `main.c` que vous pourrez ouvrir en double-cliquant dessus. Vous voilà parés !

Visual C++ (Windows seulement)

Quelques petits rappels sur Visual C++ :

- c'est l'IDE de Microsoft ;
- il est à la base payant, mais Microsoft a sorti une version gratuite intitulée Visual C++ Express ;
- il permet de programmer en C et en C++ (et non pas seulement en C++ comme son nom le laisse entendre).

Nous allons bien entendu voir ici la version gratuite, Visual C++ Express (attention, il n'est compatible avec Windows 7 qu'à partir de la version 2010) :



Quelles sont les différences avec le « vrai » Visual ?

Il n'y a pas l'éditeur de ressources qui vous permet de dessiner des images, des icônes, ou des fenêtres. Mais bon, ça, entre nous, on s'en moque bien parce qu'on n'aura pas besoin de s'en servir dans ce cours. Ce ne sont pas des fonctionnalités indispensables, bien au contraire.

Pour télécharger Visual C++ Express, rendez-vous sur le [site web de Visual C++](#). Sélectionnez ensuite Visual C++ Express Français un peu plus bas sur la page.

Visual C++ Express est en français et totalement gratuit. Ce n'est donc pas une version d'essai limitée dans le temps. C'est une chance d'avoir un IDE aussi puissant que celui de Microsoft disponible gratuitement, ne la laissez donc pas passer.

Installation

L'installation devrait normalement se passer sans encombre. Le programme d'installation va télécharger la dernière version de Visual sur Internet.

Je vous conseille de laisser les options par défaut.

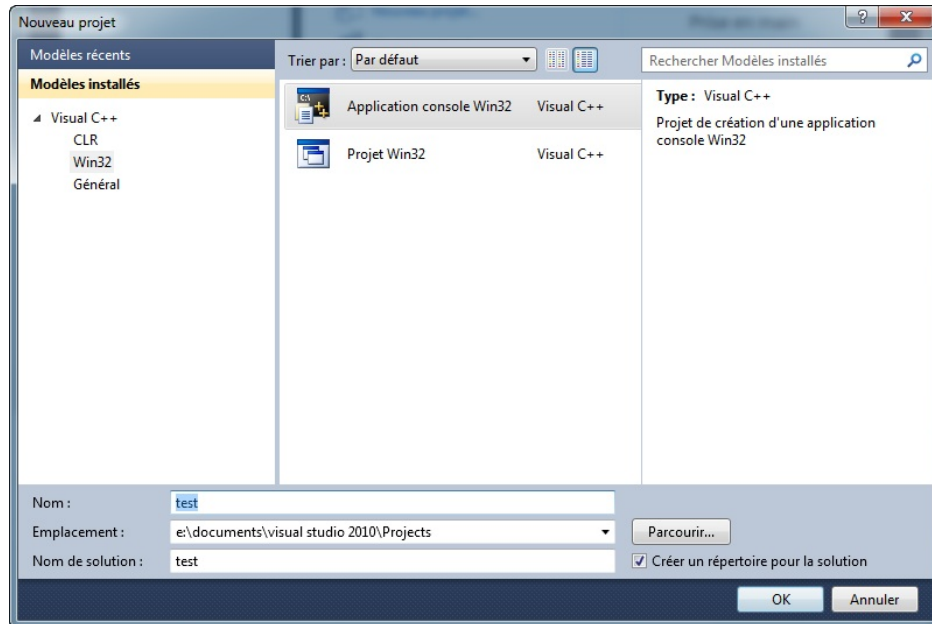
Il faut vous enregistrer dans les 30 jours. Pas de panique, c'est gratuit et rapide ; mais il faut le faire.

Cliquez sur le lien qui vous est donné : vous arrivez sur le site de Microsoft. Connectez-vous avec votre compte Windows Live ID (équivalent du compte Hotmail ou MSN) ou créez-en un si vous n'en avez pas, puis répondez au petit questionnaire.

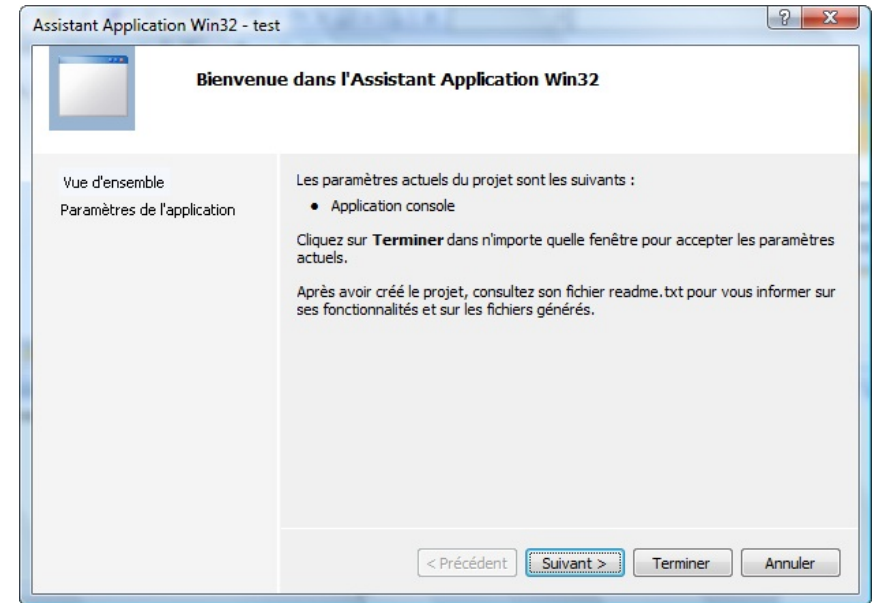
On vous donnera à la fin une clé d'enregistrement. Vous devrez recopier cette clé dans le menu ? / Inscrire le produit.

Créer un nouveau projet

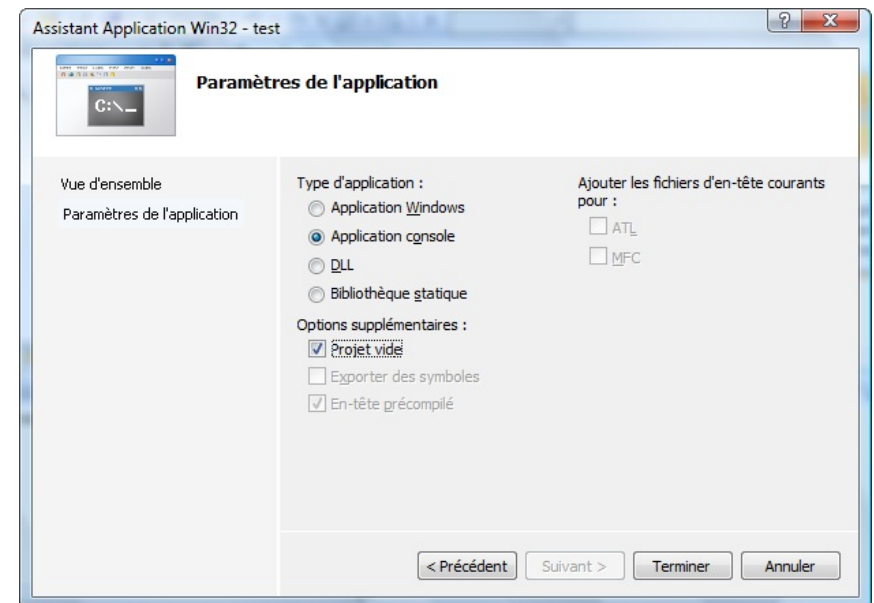
Pour créer un nouveau projet sous Visual, allez dans le menu **Fichier / Nouveau / Projet**. Sélectionnez **Win32** dans la colonne de gauche, puis **Application console Win32** à droite (fig. suivante). Entrez un nom pour votre projet, par exemple **test**.



Validez. Une nouvelle fenêtre s'ouvre :



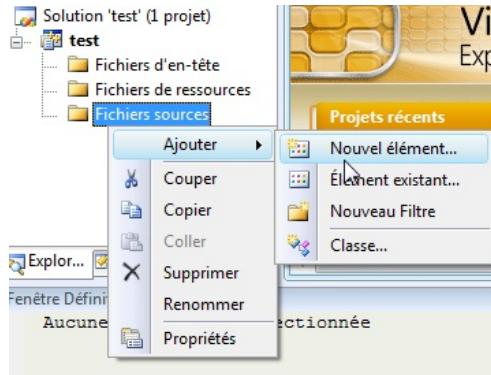
Cette fenêtre ne sert pas à grand-chose. Par contre, cliquez sur **Paramètres de l'application** de l'application dans la colonne de gauche.



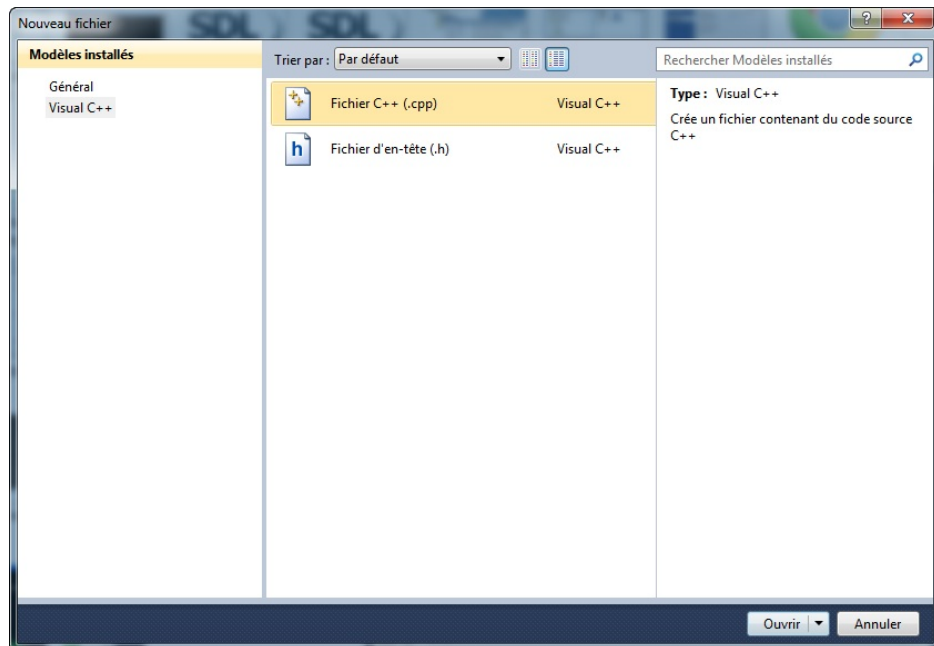
Veillez à ce que `Projet vide` soit coché comme sur la fig. suivante. Cliquez enfin sur `Terminer`.

Ajouter un nouveau fichier source

Votre projet est pour l'instant bien vide. Faites un clic droit sur le dossier `Fichiers sources` situé sur votre gauche, puis allez dans `Ajouter / Nouvel élément` (fig. suivante).



Une fenêtre s'ouvre. Sélectionnez `Visual C++` à gauche puis `Fichier C++ (.cpp)` (je sais, on ne fait pas de C++ mais ça n'a pas d'importance ici). Entrez un nom pour votre fichier : `main.c`, comme sur la figure suivante.

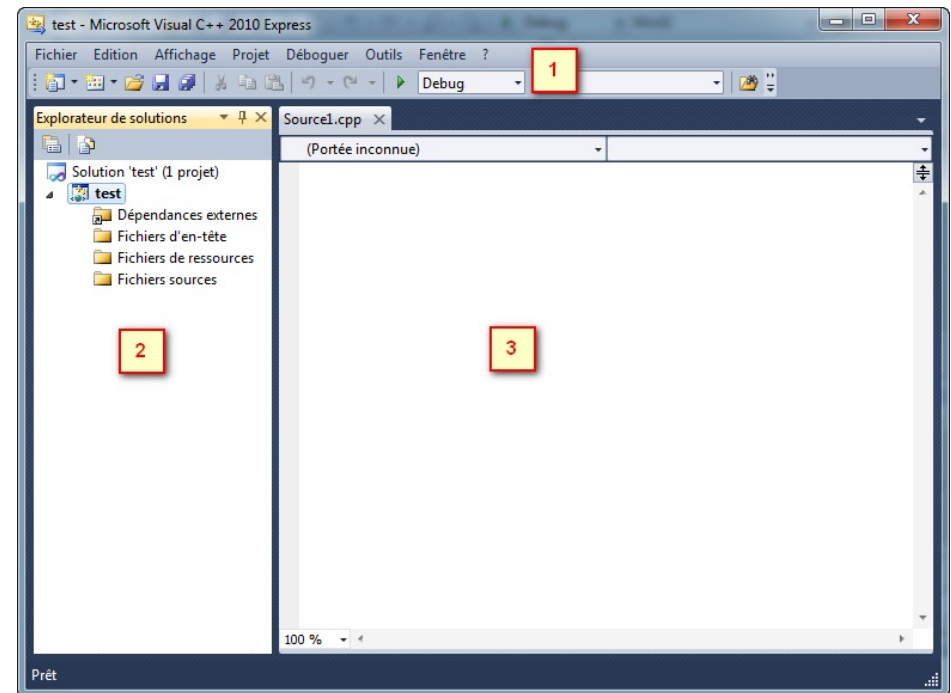


Cliquez sur `Ajouter`. Un fichier vide est créé, je vous invite à l'enregistrer rapidement sous le nom de `main.c`.

C'est bon, vous allez pouvoir commencer à écrire du code !

La fenêtre principale de Visual

Voyons ensemble le contenu de la fenêtre principale de Visual C++ Express (fig. suivante).



Cette fenêtre ressemble en tous points à celle de Code::Blocks. On va rapidement (re)voir quand même ce que signifient chacune des parties.

1. La barre d'outils : tout ce qu'il y a de plus standard. Ouvrir, enregistrer, enregistrer tout, couper, copier, coller, etc. Par défaut, il semble qu'il n'y ait pas de bouton de barre d'outils pour compiler. Vous pouvez les rajouter en faisant un clic droit sur la barre d'outils, puis en choisissant `Déboguer` et `Générer` dans la liste. Toutes ces icônes de compilation ont leur équivalent dans les menus `Générer` et `Déboguer`. Si vous faites `Générer`, cela créera l'exécutable (ça signifie « compiler » pour Visual). Si vous faites `Déboguer / Exécuter`, on devrait vous proposer de compiler avant d'exécuter le programme. F7 permet de générer le projet, et F5 de l'exécuter.
2. Dans cette zone très importante vous voyez normalement la liste des fichiers de votre projet. Cliquez sur l'onglet `Explorateur de solutions` en bas, si ce n'est déjà fait. Vous devriez voir que Visual crée déjà des dossiers pour séparer les différents types de fichiers de votre projet (sources, en-tête et ressources). Nous verrons un peu plus tard quels sont les différents types de fichiers qui constituent un projet.
3. La partie principale : c'est là qu'on modifie les fichiers source.

Voilà, on a fait le tour de Visual C++. Vous pouvez aller jeter un œil dans les options (Outils / Options) si ça vous chante, mais n'y passez pas trois heures. Il faut dire qu'il y a tellement de cases à cocher de partout qu'on ne sait plus trop où donner de la tête.

Xcode (Mac OS seulement)

Il existe plusieurs IDE compatibles Mac. Il y a Code::Blocks bien sûr, mais ce n'est pas le seul. Je vais vous présenter ici l'IDE le plus célèbre sous Mac : Xcode.

Cette section dédiée à Xcode est inspirée d'un [tutoriel](#) paru sur [LogicielMac.com](#), avec l'aimable autorisation de son auteur PsychoH13. Merci à Flohw pour la mise à jour des captures.

Xcode, où es-tu ?



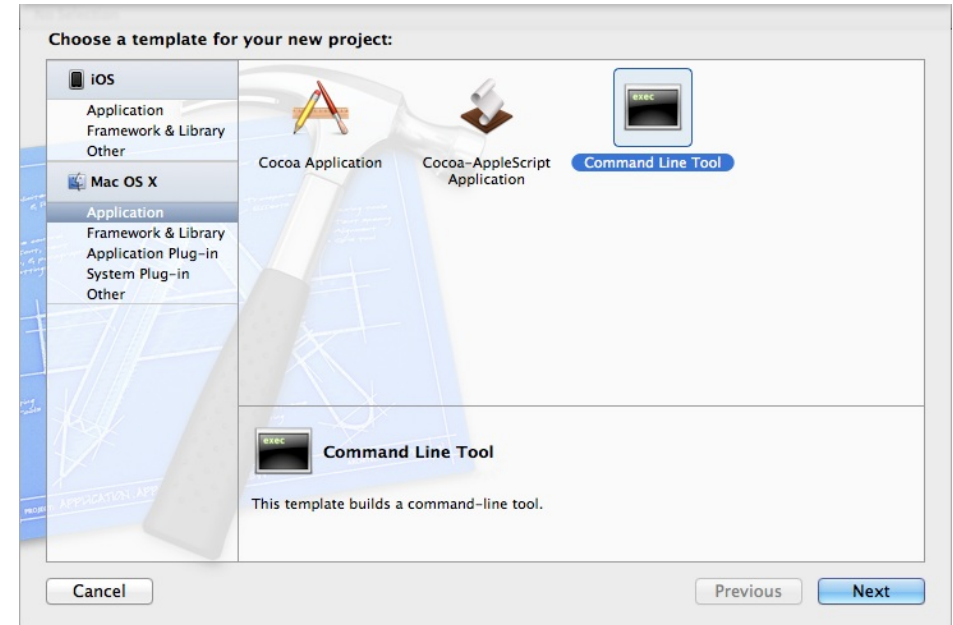
Tous les utilisateurs de Mac OS ne sont pas des programmeurs. Apple l'a bien compris et n'installe pas par défaut d'IDE avec Mac OS. Heureusement, pour ceux qui voudraient programmer, tout est prévu. En effet, Xcode (logo en fig. suivante) est disponible sur le MacAppStore. Commencez donc par le récupérer là-bas.

Par ailleurs, je vous conseille de mettre en favoris [la page dédiée aux développeurs sur le site d'Apple](#). Vous y trouverez une foule d'informations utiles pour le développement sous Mac. Vous pourrez notamment y télécharger plusieurs logiciels pour développer. N'hésitez pas à vous inscrire à l'ADC (« Apple Development Connection »), c'est gratuit et vous serez ainsi tenus au courant des nouveautés.

Lancement de Xcode

Xcode est l'IDE le plus utilisé sous Mac, créé par Apple lui-même. Les plus grands logiciels, comme iPhoto et Keynote, ont été codés à l'aide de Xcode. C'est réellement l'outil de développement de choix quand on a un Mac !

La première chose à faire est de créer un nouveau projet, alors commençons par ça. Allez dans le menu **File / New Project**. Une fenêtre de sélection de projet s'ouvre (fig. suivante).



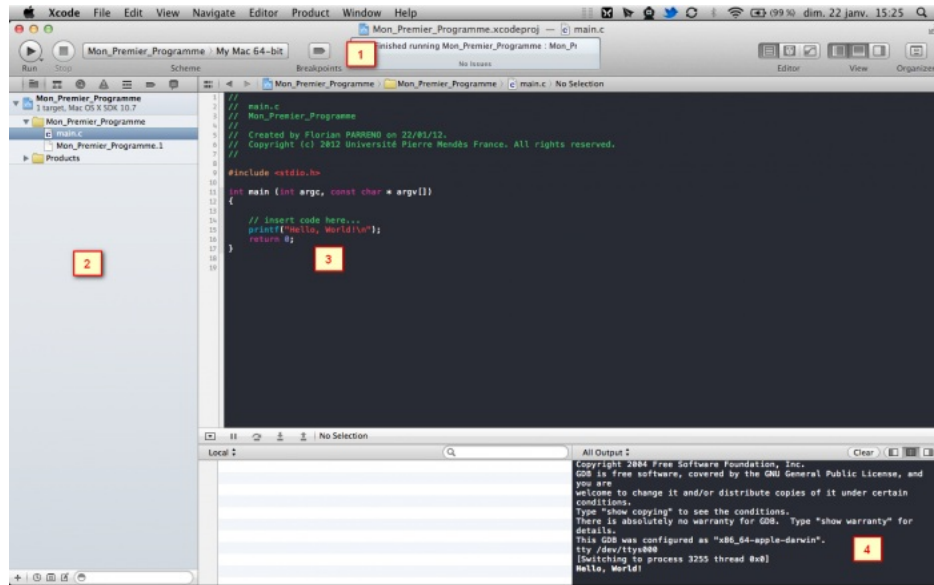
Allez dans la section **Application** et sélectionnez **Command Line Tool**. Si vous avez une version plus ancienne du logiciel, il vous faudra probablement aller dans la section **Command line utility** et sélectionner **Standard tool**.

Cliquez ensuite sur **Next**. On vous demandera où vous voulez enregistrer votre projet (un projet doit toujours être enregistré dès le début) ainsi que son nom. Placez-le dans le dossier que vous voulez.

Une fois créé, votre projet se présentera sous la forme d'un dossier contenant de multiples fichiers dans le **Finder**. Le fichier à l'extension `.xcodeproj` correspond au fichier du projet. C'est lui que vous devrez sélectionner la prochaine fois si vous souhaitez réouvrir votre projet.

La fenêtre de développement

Dans Xcode, si vous sélectionnez `main.c` à gauche, vous devriez avoir une fenêtre similaire à la fig. suivante.



La fenêtre est découpée en quatre parties, ici numérotées de 1 à 4.

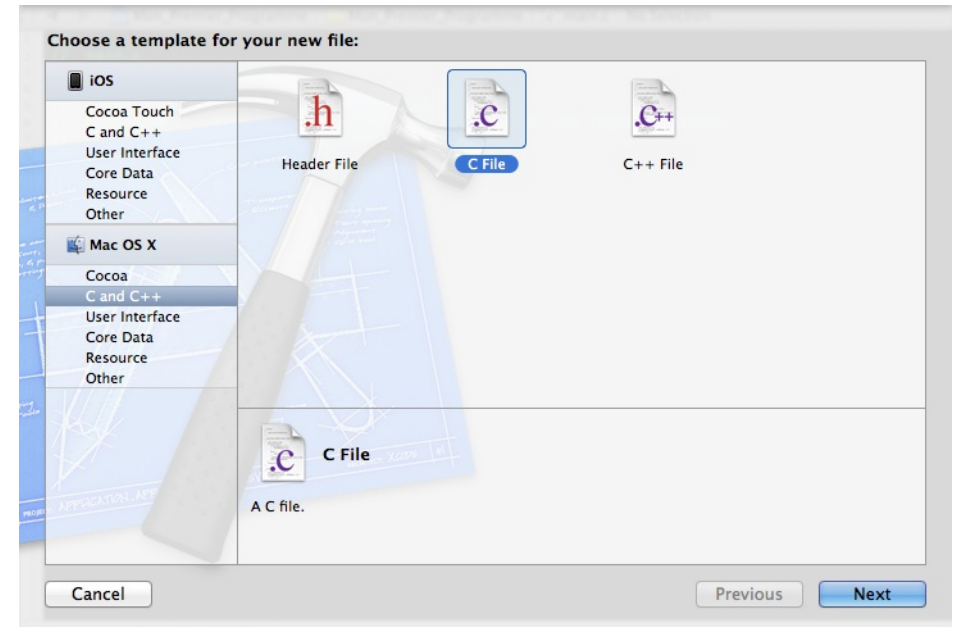
1. La première partie est la barre de boutons tout en haut. Le plus important d'entre eux, Run, vous permettra d'exécuter votre programme.
2. La partie de gauche correspond à l'arborescence de votre projet. Certaines sections regroupent les erreurs, les avertissements, etc. Xcode vous place automatiquement dans la section la plus utile, celle qui porte le nom de votre projet.
3. La troisième partie change en fonction de ce que vous avez sélectionné dans la partie de gauche. Ici, on a le contenu de notre fichier `main.c`.
4. Enfin, la quatrième partie affiche le résultat de l'exécution du programme dans la console, lorsque vous avez cliqué sur Run.

Ajouter un nouveau fichier

Au début, vous n'aurez qu'un seul fichier source (`main.c`). Cependant, plus loin dans le cours, je vous demanderai de créer de nouveaux fichiers source lorsque nos programmes deviendront plus gros.

Pour créer un nouveau fichier source sous Xcode, rendez-vous dans le menu **File / New File**.

Un assistant vous demande quel type de fichier vous voulez créer. Rendez-vous dans la section **Mac OS X/C and C++** et sélectionnez **C File** (Fichier C). Vous devriez avoir sous les yeux la fig. suivante.



Vous devrez donner un nom à votre nouveau fichier (ce que vous voulez). L'extension, elle, doit rester `.c`. Parfois - nous le verrons plus loin -, il faudra aussi créer des fichiers `.h` (mais on en reparlera). La case à cocher **Also create fichier.h** est là pour ça. Pour le moment, elle ne nous intéresse pas.

Cliquez ensuite sur **Finish**. C'est fait ! Votre fichier est créé et ajouté à votre projet, en plus de `main.c`.

Vous êtes maintenant prêts à programmer sous Mac !

En résumé

- Les programmeurs ont besoin de trois outils : un éditeur de texte, un compilateur et un débogueur.
- Il est possible d'installer ces outils séparément, mais il est courant aujourd'hui d'avoir un package trois-en-un que l'on appelle **IDE**, l'environnement de développement.
- Code::Blocks, Visual C++ et Xcode comptent parmi les IDE les plus célèbres.

Votre premier programme

On a préparé le terrain jusqu'ici, maintenant il serait bien de commencer à programmer un peu, qu'en dites-vous ? C'est justement l'objectif de ce chapitre ! À la fin de celui-ci, vous aurez réussi à créer votre premier programme !

Bon d'accord, ce programme sera en noir et blanc et ne saura que vous dire bonjour, il semblera donc complètement inutile mais ce sera votre premier ; je peux vous assurer que vous en serez fiers.

Console ou fenêtre ?

Nous avons rapidement parlé de la notion de « programme console » et de « programme fenêtre » dans le chapitre précédent. Notre IDE nous demandait quel type de programme nous voulions créer et je vous avais dit de répondre `console`.

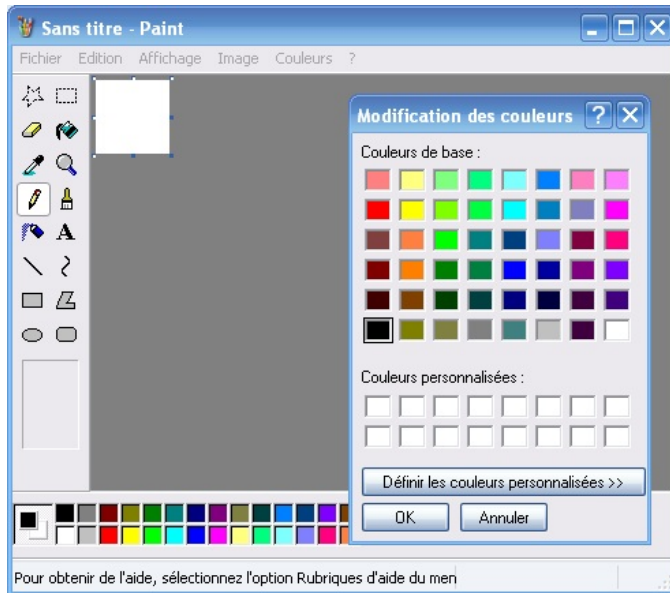
Il faut savoir qu'en fait il existe deux types de programmes, pas plus :

- les programmes avec fenêtres ;
- les programmes en console.

Les programmes en fenêtres

Ce sont les programmes que vous connaissez.

La fig. suivante est un exemple de programme en fenêtres que vous connaissez sûrement.



Ça donc, c'est un programme avec des fenêtres. Je suppose que vous aimeriez bien créer ce type de programmes, hmm ? Eh bien... vous n'allez pas pouvoir de suite !

En effet, créer des programmes avec des fenêtres en C c'est possible, mais... quand on débute, c'est bien trop compliqué ! Pour débiter, 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. À 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 beaucoup de 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 !

J'ai une grande nouvelle ! **La console n'est pas morte** ! En effet, Linux a remis au goût du jour l'utilisation de la console. La fig. suivante est une capture d'écran d'une console sous Linux.

```
2.2.5_appli.html 3.2.7.css 3.6.8.html
2.2.5.css 3.2.8.css 3.6.9.html
2.2.6_appli.html 3.2.9_appli.html ancras.html
2.2.6.css 3.2.9.css base.php
2.3.10_appli.html 3.3.10.html cible_formulaire.php
2.3.10.css 3.3.11.css cible.html
2.3.11_appli.html 3.3.12.css designl.css
2.3.11.css 3.3.13_appli.html erreur_paragraphe.html
2.3.12.css 3.3.13.css essai2.css
2.3.13.html 3.3.14_appli.html essai.css
2.3.14.css 3.3.14.css images
2.3.15.html 3.3.15.css tests_design.html
2.3.16.css 3.3.1.css traitement.php
2.3.17.css 3.3.2.html
2.3.18.css 3.3.3.css

[root@ns1 exemples]# cd ..
[root@ns1 xhtml-css]# ls
anim css.php images pseudoformats.php
annexes design.php images.php qcm.php
autres index.php tableaux.php
boites_partiel.php formatage_partiel.php intro.php texte.php
boites_partie2.php formatage_partie2.php liens.php xhtml.php
conclusion.php formulaires.php listes.php

[root@ns1 xhtml-css]#
```

Brr... Terrifiant, hein ? Voilà, vous avez maintenant une petite idée de ce à quoi ressemble une console.

Ceci dit, plusieurs remarques :

- aujourd'hui on sait afficher de la couleur, tout n'est donc pas en noir et blanc comme on pourrait le croire ;
- 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 débiter (ce qui n'est pas le cas des programmes en mode « fenêtre »).

Notez que la console a évolué : elle peut afficher des couleurs, et rien ne vous empêche de mettre une image de fond.



Et sous Windows ? Il n'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`.

La fig. suivante représente la maaagnifique 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 pouvoir ensuite 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.

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 si vous avez oublié comment faire.

Pour rappel, sous Code::Blocks (qui est l'IDE que je vais utiliser tout au long de ce cours), il faut aller dans le menu **File / New / Project**, puis choisir **Console Application** et sélectionner le langage C.

Code::Blocks a donc généré le minimum de code en langage C dont on a besoin. Le voici :

Code : C

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

int main()
{
    printf("Hello world!\n");
    return 0;
}
```



Notez qu'il y a une ligne vide à la fin de ce code. Il est nécessaire de taper sur la touche « Entrée » après la dernière accolade. Chaque fichier en C devrait normalement se terminer par une ligne vide. Si vous ne le faites pas, ce n'est pas grave, mais le compilateur risque de vous afficher un avertissement (*warning*).

Notez que la ligne :

Code : C

```
int main()
```

... peut aussi s'écrire :

Code : C

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

Les deux écritures sont possibles, mais la seconde (la compliquée) est la plus courante. J'aurai donc tendance à utiliser plutôt cette dernière dans les prochains chapitres. En ce qui nous concerne, que l'on utilise l'une ou l'autre des écritures, ça ne changera rien pour nous. Inutile donc de s'y attarder, surtout que nous n'avons pas encore le niveau pour analyser ce que ça signifie.

Si vous êtes sous un autre IDE, copiez ce code source dans votre fichier `main.c` pour que nous ayons le même code vous et moi.

Enregistrez le tout. Oui je sais, on n'a encore rien fait, mais enregistrez quand même, c'est une bonne habitude à prendre. Normalement, vous n'avez qu'un seul fichier source appelé `main.c` (le reste, ce sont des fichiers de projet générés par votre IDE).

Analysons le code minimal

Ce code minimal qu'on vient de voir n'est rien d'autre que du chinois pour vous, j'imagine. Et pourtant, moi je vois là un programme console qui affiche un message à l'écran. Il va falloir apprendre à lire tout ça !

Commençons par les deux premières lignes qui se ressemblent beaucoup :

Code : C

```
#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 ?). 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 d'insérer ces lignes en haut de chacun de vos fichiers.



Où 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 deux lignes, donc deux fichiers inclus. Ces fichiers s'appellent `stdio.h` et `stdlib.h`. Ces fichiers existent déjà, des fichiers source tout prêts. On verra plus tard qu'on les appelle des **bibliothèques** (certains parlent aussi de **librairies** mais c'est un anglicisme). En gros, ces fichiers contiennent du code tout prêt qui permet d'afficher du texte à l'écran.

Sans ces fichiers, écrire du texte à l'écran aurait été mission impossible. L'ordinateur à la base ne sait rien faire, il faut tout lui dire. Vous voyez la galère dans laquelle on est !

Bref, en résumé les deux premières lignes incluent les bibliothèques 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

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

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 `{ }`. Toute la fonction `main` se trouve donc entre ces accolades. Si vous avez bien suivi, notre fonction `main` contient deux lignes :

Code : C

```
printf("Hello world!\n");
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 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.



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

Passons à l'instruction suivante justement :

```
return 0;
```

Eh bien ça, en gros, ça veut dire que c'est fini (eh oui, déjà). Cette ligne indique qu'on arrive à la fin de notre fonction `main` et demande de renvoyer la valeur 0.



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. En pratique, 0 signifie « tout s'est bien passé » et n'importe quelle autre valeur signifie « 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, cela ferait trop de choses à assimiler.

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 : comme je suis de bonne humeur, je vous fais un schéma qui récapitule le vocabulaire qu'on vient d'apprendre (fig. suivante).

```
#include <stdio.h>
#include <stdlib.h> } Directives de préprocesseur

int main()
{
    printf("Hello world!\n");
    return 0;
} } Instructions } Fonction
```

Testons notre programme

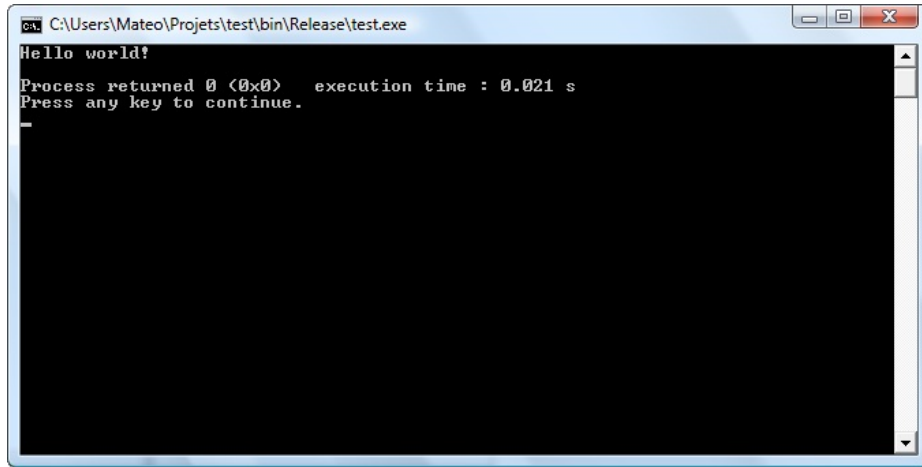
Tester devrait aller vite. Tout ce que vous avez à faire c'est compiler le projet, puis l'exécuter (cliquez sur l'icône Build & Run sous Code::Blocks).

Si vous ne l'avez pas encore fait, on vous demandera d'enregistrer les fichiers. Faites-le.



Si la compilation ne fonctionne pas et que vous avez une erreur de ce type : "My-program - Release" uses an invalid compiler. Skipping... Nothing to be done... Cela signifie que vous avez téléchargé la version de Code::Blocks sans mingw (le compilateur). Retournez sur le site de Code::Blocks pour télécharger la version avec mingw.

Après un temps d'attente insupportable (la compilation), votre premier programme va apparaître sous vos yeux totalement envahis de bonheur (fig. suivante).



Le programme affiche « Hello world! » (sur la première ligne).

Les lignes en dessous ont été générées par Code::Blocks et indiquent que le programme s'est bien exécuté et combien de temps s'est écoulé depuis le lancement.

On vous invite à appuyer sur n'importe quelle touche du clavier pour fermer la fenêtre. Votre programme s'arrête alors. Oui je sais, ce n'est pas transcendant. Mais bon, quand même ! C'est un premier programme, un instant dont vous vous souviendrez toute votre vie ! ... Non ?

Écrire un message à l'écran

À partir de maintenant, on va modifier nous-mêmes le code de 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 choisir le texte qui s'affiche à l'écran ?

Ce sera en fait assez simple. Si vous partez du code qui a été donné plus haut, il vous suffit simplement de remplacer « Hello world! » par « Bonjour » dans la ligne qui fait appel à `printf`.

Comme je vous le disais plus tôt, `printf` est une **instruction**. Elle commande à l'ordinateur : « *Affiche-moi ce message à l'écran* ». Il faut savoir que `printf` est en fait une fonction qui a déjà été 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 deux lignes ?

Code : C

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

Je vous avais dit qu'elles permettaient d'ajouter des bibliothèques dans votre programme.

Les bibliothèques sont en fait des fichiers avec des tonnes 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 fait donc 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 : une fonction contient des instructions qui appellent d'autres fonctions, et ainsi de suite.

Donc, pour faire appel à une fonction, c'est simple : il suffit d'écrire son nom, suivi de deux parenthèses, puis un point-virgule.

Code : C

```
printf();
```

C'est bien, mais ce n'est pas suffisant. Il faut indiquer quoi écrire à l'écran. Pour faire ça, il faut donner à la fonction `printf` le texte à afficher. Pour ce faire, ouvrez des guillemets à l'intérieur des parenthèses et tapez le texte à afficher entre ces guillemets, comme cela avait déjà été fait sur le code minimal.

Dans notre cas, on va donc taper très exactement :

Code : C

```
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

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

int main()
{
    printf("Bonjour");
    return 0;
}
```

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

1. affiche « Bonjour » à l'écran ;
2. la fonction `main` est terminée, renvoie 0. Le programme s'arrête alors.

La fig. suivante vous montre ce que donne ce programme à l'écran.

```

C:\Users\Mateo\Projets\test\bin\Release\test.exe
Bonjour
Process returned 0 (0x0)   execution time : 0.046 s
Press any key to continue.

```

Comme vous pouvez le voir, la ligne du « Bonjour » est un peu collée avec le reste du texte, contrairement à tout à l'heure. 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 »).

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. Ils sont faciles à reconnaître : c'est un ensemble de deux caractères. Le premier d'entre eux est toujours un anti-slash (\), et le second un nombre ou une lettre. Voici deux 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 devrais taper :

Code : C

```
printf("Bonjour\n");
```

Votre ordinateur comprend qu'il doit afficher « Bonjour » suivi d'un retour à la ligne (fig. suivante).

```

C:\Users\Mateo\Projets\test\bin\Release\test.exe
Bonjour
Process returned 0 (0x0)   execution time : 0.096 s
Press any key to continue.

```

C'est déjà un peu mieux, non ?



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.

À partir de là vous avez deux solutions.

- **Passer à Linux.** C'est une solution un peu radicale et il me faudrait un cours entier pour vous expliquer comment vous servir de Linux. Si vous n'avez pas le niveau, oubliez cette possibilité pour le moment.
- **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 rencontrerez plus ce problème-là. Je vous recommande donc de ne pas utiliser d'accents temporairement, pendant votre apprentissage dans la console. Vos futurs programmes « professionnels » n'auront pas ce problème, rassurez-vous.

Pour ne pas être gêné, vous devriez donc écrire sans accent :

Code : C

```
printf("Bonjour Gerard\n");
```

On remercie notre ami Gérard pour nous avoir soulevé ce problème !

Si d'aventure vous vous appelez Gérard, sachez que je n'ai rien contre ce prénom. C'est simplement le premier prénom avec un accent qui m'est passé par la tête... Et puis bon, il faut toujours que quelqu'un prenne pour les autres, que voulez-vous !

Les commentaires, c'est très utile !

Avant de terminer ce premier chapitre de « véritable » programmation, je dois absolument vous faire découvrir 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 source plus tard. On ne dirait pas comme ça, mais on oublie vite comment fonctionnent les programmes qu'on a écrits. Si vous faites une pause ne serait-ce que 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 a priori pas votre code source), cela lui permettra de se familiariser avec bien plus rapidement ;
- enfin, ça va me permettre à moi d'ajouter des annotations dans les codes source de ce cours. Et de mieux vous expliquer à quoi peut servir telle ou telle ligne de code.

Il y a plusieurs manières d'insérer 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

```
// 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

```
printf("Bonjour"); // Cette instruction affiche Bonjour à l'écran
```

- Votre commentaire est **long** : vous avez beaucoup à dire, 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 écririez donc par exemple :

Code : C

```
/* 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

```
/*
```

```
Ci-dessous, ce sont des directives de préprocesseur.
Ces lignes permettent d'ajouter des fichiers au projet,
fichiers que l'on appelle bibliothèques.
Grâce à ces bibliothèques, 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.
*/
```

```
int main()
{
    printf("Bonjour"); // Cette instruction affiche Bonjour à l'écran
    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. À 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

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

En résumé

- Les programmes peuvent communiquer avec l'utilisateur via une console ou une fenêtre.
- Il est beaucoup plus facile pour nos premiers programmes de travailler avec la **console**, bien que celle-ci soit moins attirante pour un débutant. Cela ne nous empêchera pas par la suite de travailler avec des fenêtres dans la partie III. Tout vient à point à qui sait attendre. 😊
- Un programme est constitué d'**instructions** qui se terminent toutes par un point-virgule.
- Les instructions sont contenues dans des **fonctions** qui permettent de les classer, comme dans des boîtes.
- La fonction `main` (qui signifie « principale ») est la fonction par laquelle démarre votre programme. C'est la seule qui soit obligatoire, aucun programme ne peut être compilé sans elle.
- `printf` est une fonction toute prête qui permet d'afficher un message à l'écran dans une console.
- `printf` se trouve dans une **bibliothèque** où l'on retrouve de nombreuses autres fonctions prêtes à l'emploi.

Un monde de variables

Vous savez afficher un texte à l'écran. Très bien. Ça ne vole peut-être pas très haut pour le moment, mais c'est justement parce que vous ne connaissez pas encore ce qu'on appelle les **variables** en programmation.

Le principe dans les grandes lignes, c'est de faire retenir des nombres à l'ordinateur. On va apprendre à stocker des nombres dans la mémoire.

Je souhaite que nous commençons par quelques explications sur la mémoire de votre ordinateur. Comment fonctionne une mémoire ? Combien un ordinateur possède-t-il de mémoires différentes ? Cela pourra paraître un peu simpliste à certains d'entre vous, mais je pense aussi à ceux qui ne savent pas bien ce qu'est une mémoire.

Une affaire de mémoire

Ce que je vais vous apprendre dans ce chapitre a 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 suffi, 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 données). Or, vous allez rire, mais jusqu'ici nous avons été incapables de créer une mémoire qui soit à la fois très rapide et importante. Plus exactement, la mémoire rapide coûte cher, on n'en fait donc 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 ultra-rapide 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, je souhaite 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 (abrégié « 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.

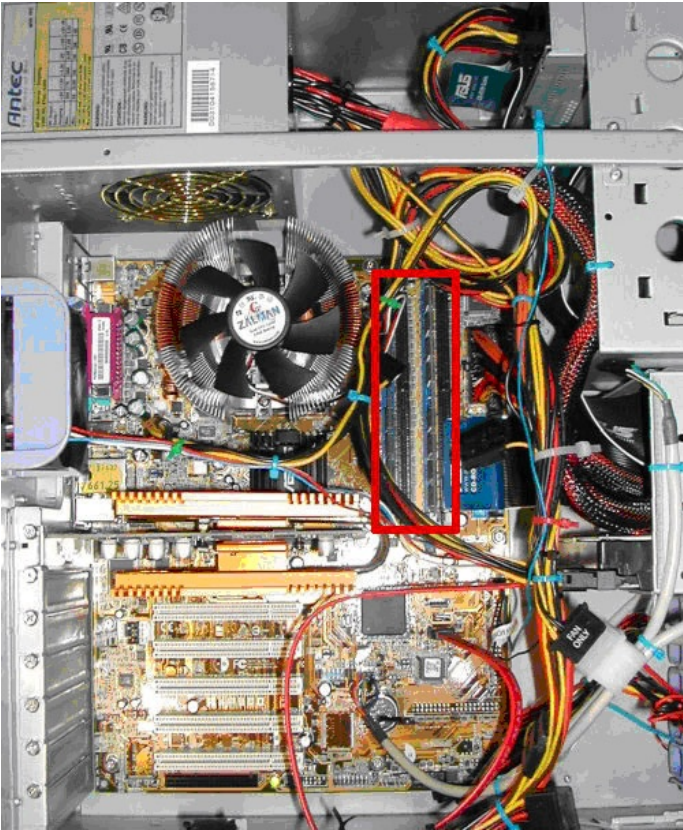
On va y aller par zooms successifs. Commençons par votre ordinateur (fig. suivante).



Vous reconnaissez le clavier, la souris, l'écran et l'unité centrale (la tour). Intéressons-nous maintenant à l'unité centrale (fig. suivante), 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. Ouvrons-la (fig. suivante).



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. 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'œil nu.

Voilà à quoi ressemble une barrette de mémoire vive de plus près (fig. suivante).



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.

Je vous propose un schéma du fonctionnement de la mémoire vive (fig. suivante). Il est très simplifié (comme mes schémas de


compilation !), mais c'est parce que nous n'avons pas besoin de trop de détails. Si vous retenez ce schéma, ce sera déjà très bien !

Adresse	Valeur
0	145
1	3.8028322
2	0.827551
3	3901930
...	...
3 448 765 900 126 (et des poussières)	940.5118

Comme vous le voyez, il faut en gros distinguer deux 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.
- À 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 personnage d'un jeu), 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.

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 deux choses :

- une **valeur** : c'est le nombre qu'elle stocke, par exemple 5 ;
- un **nom** : c'est ce qui permet de la reconnaître. En programmant en C, on n'aura pas à retenir l'adresse mémoire (ouf !) : à la place, on va juste indiquer des noms de variables. C'est le compilateur qui fera la conversion entre le nom et l'adresse. Voilà déjà un souci de 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 minuscules, majuscules et des chiffres (abcABC012) ;
- votre nom de variable doit commencer par une lettre ;
- les espaces sont interdits. À la place, on peut utiliser le caractère « underscore » _ (qui ressemble à un trait de soulignement). C'est le seul caractère différent des lettres et chiffres autorisé ;
- vous n'avez pas le droit d'utiliser des accents (éâê etc.).


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

Voici quelques exemples de noms de variables corrects : `nombreDeVies`, `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 abrégé `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 virgule :

- 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 ... ça 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 :

Nom du type	Minimum	Maximum
<code>signed char</code>	-127	127
<code>int</code>	-32 767	32 767
<code>long</code>	-2 147 483 647	2 147 483 647
<code>float</code>	-1 x10 ³⁷	1 x10 ³⁷
<code>double</code>	-1 x10 ³⁷	1 x10 ³⁷



Les valeurs présentées ci-dessus sont les minimums garantis par le langage. En réalité, il est fort probable que vous puissiez stocker des valeurs plus élevées que celles-ci. Cependant, veuillez à garder ces valeurs en tête lorsque vous choisissez un type, c'est important.



Notez que tous les types n'ont pas été présentés, seul les principaux ont été conservés.

Les trois premiers types (`signed char`, `int`, `long`) permettent de stocker des nombres entiers : 1, 2, 3, 4...
Les deux derniers (`float`, `double`) permettent de stocker des nombres décimaux (appelés nombres **flottants**) : 13.8, 16.911...

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 entiers (`signed 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 :

<code>unsigned char</code>	0 à 255
<code>unsigned int</code>	0 à 65 535

<code>unsigned long</code>	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 nombres négatifs, mais l'avantage de pouvoir stocker des nombres deux fois plus grands (`signed char` s'arrête à 127, tandis que `unsigned char` s'arrête à 255 par exemple).



Vous remarquerez que le type `char` a été présenté soit avec le mot-clé `signed`, soit avec le mot-clé `unsigned`, mais jamais seul. La raison en est toute simple : le type peut-être signé ou non signé suivant les machines. Veillez donc à spécifier lequel des deux vous souhaitez utiliser suivant le type de valeur que vous désirez stocker.



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

Oui, mais on a créé à l'origine plusieurs types 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 `int`.

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 `int` (ou `double` pour un flottant).

En résumé, on fera surtout la distinction entre nombres entiers et flottants :

- pour un nombre **entier**, on utilisera le plus souvent `int` ;
- pour un nombre **flottant**, 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 **demandeur à l'ordinateur la permission d'utiliser un peu de mémoire**.

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

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

Par exemple, si je veux créer ma variable `nombreDeVies` de type `int`, je dois taper la ligne suivante :

Code : C

```
int nombreDeVies;
```

Et c'est tout ! Quelques autres exemples stupides pour la forme :

Code : C

```
int noteDeMaths;
double sommeArgentRecue;
unsigned int nombreDeLecteursEnTrainDeLireUnNomDeVariableUnPeuLong;
```

Bon bref, vous avez compris le principe je pense !

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

Code : C

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

int main(int argc, char *argv[]) // Équivalent de int main()
{
    int nombreDeVies;

    return 0;
}
```

Si vous lancez le programme ci-dessus, vous constaterez avec stupeur... qu'il ne fait rien.

Quelques explications

Alors, avant que vous ne métrangiez en croyant que je vous mène en bateau depuis tout à l'heure, laissez-moi juste dire une chose pour ma défense.

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 souci qu'il pourrait y avoir, c'est qu'il n'y ait plus de place en mémoire... Mais heureusement cela arrive rarement, car pour remplir toute la mémoire rien qu'avec des `int` il faut vraiment le vouloir !

Soyez donc sans crainte, 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 variables par des virgules sur la même ligne : `int nombreDeVies, niveau, ageDuJoueur;`. Cela créera trois variables `int` 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

```
nombreDeVies = 5;
```

Rien de plus à faire. Vous indiquez le nom de la variable, un signe égal, puis la valeur que vous voulez y mettre. Ici, on vient de donner la valeur 5 à la variable `nombreDeVies`.

Notre programme complet ressemble donc à ceci :

Code : C

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

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

    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. N'est-ce pas magnifique ?

On peut s'amuser si on veut à changer la valeur par la suite :

Code : C

```
int 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.

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

```
int 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

```
int nombreDeVies = 5;
```

Ici, la variable `nombreDeVies` est déclarée et 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

```
const int 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.

À 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 : « Triple idiot, pourquoi tu essaies de modifier la valeur d'une constante ? »).

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 l'on veut afficher la valeur de la variable. Par exemple :

Code : C

```
printf("Il vous reste %d vies");
```

Ce « symbole spécial » dont je viens de vous parler est en fait un `%` suivi d'une lettre (dans mon exemple, la lettre `'d'`). Cette lettre permet d'indiquer ce que l'on doit afficher. `'d'` signifie que l'on souhaite afficher un `int`.

Il existe plusieurs autres possibilités, mais pour des raisons de simplicité on va se contenter de retenir celles-ci :

Format	Type attendu
<code>"%d"</code>	<code>int</code>

"%ld"	long
"%f"	float
"%f"	double



Remarquez que le format utilisé pour afficher un **float** et un **double** est identique.

Je vous parlerai des autres symboles en temps voulu. Pour le moment, retenir uniquement ceux-ci 😊

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

```
printf("Il vous reste %d vies", nombreDeVies);
```

Le `%d` sera remplacé par la variable indiquée après la virgule, à savoir `nombreDeVies`. On se teste ça dans un programme ?

Code : C

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

int main(int argc, char *argv[])
{
    int nombreDeVies = 5; // Au départ, le joueur a 5 vies

    printf("Vous avez %d 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 %d vies maintenant
    !\n\n", nombreDeVies);

    return 0;
}
```

Ça 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 !
```

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 `%d` ou des `%f` là où vous voulez, puis d'indiquer les variables correspondantes dans le même ordre, séparées par des virgules.

Par exemple :

Code : C

```
printf("Vous avez %d vies et vous etes au niveau n° %d",
    nombreDeVies, niveau);
```



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

```
int main(int argc, char *argv[])
{
    int nombreDeVies = 5, niveau = 1;

    printf("Vous avez %d vies et vous etes au niveau n° %d\n",
    nombreDeVies, niveau);

    return 0;
}
```

Ce qui affichera :

Code : Console

```
Vous avez 5 vies et vous etes au niveau n° 1
```

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 d'entrer quelque chose dans la console, on va utiliser une autre fonction toute prête : `scanf`. Cette fonction ressemble beaucoup à `printf`. Vous devez mettre un format pour indiquer ce que l'utilisateur doit entrer (un `int`, un `float`, ...). Puis vous devez ensuite indiquer le nom de la variable qui va recevoir le nombre.

Voici comment faire par exemple :

Code : C

```
int age = 0;
scanf("%d", &age);
```

On doit mettre le %d entre 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 ?

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 !



Attention, il y a une petite divergence de format entre printf et scanf ! Pour récupérer un float, c'est le format "%f" qu'il faut utiliser, mais pour le type double c'est le format "%lf".

Code : C

```
double poids = 0;
scanf("%lf", &poids);
```

Revenons à notre programme. Lorsque celui-ci arrive à un scanf, il se met en pause et attend que l'utilisateur entre 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

```
int main(int argc, char *argv[])
{
    int age = 0; // On initialise la variable à 0

    printf("Quel age avez-vous ? ");
    scanf("%d", &age); // On demande d'entrer l'âge avec scanf
    printf("Ah ! Vous avez donc %d ans !\n\n", age);

    return 0;
}
```

Code : Console

```
Quel age avez-vous ? 20
Ah ! Vous avez donc 20 ans !
```

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.

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 (« ééydf »), la variable ne changera pas de valeur. Ce qui est bien ici, c'est qu'on avait initialisé notre variable à 0 au début. De ce fait, 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 !

En résumé

- Nos ordinateurs possèdent plusieurs types de mémoire. De la plus rapide à la plus lente : les registres, la mémoire cache, la mémoire vive et le disque dur.
- Pour « retenir » des informations, notre programme a besoin de stocker des données dans la mémoire. Il utilise pour cela la **mémoire vive**. Les registres et la mémoire cache sont aussi utilisés pour augmenter les performances, mais cela fonctionne automatiquement, nous n'avons pas à nous en préoccuper.
- Dans notre code source, les **variables** sont des données stockées temporairement en mémoire vive. La valeur de ces données peut changer au cours du programme.
- À l'opposé, on parle de **constantes** pour des données stockées en mémoire vive. La valeur de ces données ne peut pas changer.
- Il existe plusieurs types de variables, qui occupent plus ou moins d'espace en mémoire. Certains types comme **int** sont prévus pour stocker des nombres entiers, tandis que d'autres comme **double** stockent des nombres décimaux.
- La fonction scanf permet de demander à l'utilisateur de saisir un nombre.

Une bête de calcul

Je vous l'ai dit dans le chapitre précédent : votre ordinateur n'est en fait qu'une grosse machine à calculer. Que vous soyez en train d'écouter de la musique, regarder un film ou jouer à un jeu vidéo, votre ordinateur ne fait que des calculs.

Ce chapitre va vous apprendre à réaliser la plupart des calculs qu'un ordinateur sait faire. Nous réutiliserons ce que nous venons tout juste d'apprendre, à savoir les variables. L'idée, c'est justement de faire des calculs avec vos variables : ajouter des variables entre elles, les multiplier, enregistrer le résultat dans une autre variable, etc.

Même si vous n'êtes pas fan des mathématiques, ce chapitre vous sera absolument indispensable.

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, pas de panique).

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 les faire**. Fort heureusement, nous verrons plus loin dans ce chapitre qu'il existe une bibliothèque mathématique livrée avec le langage C qui contient des fonctions mathématiques toutes prêtes. Vous n'aurez donc pas à les réécrire, à moins que vous souhaitiez volontairement passer un sale quart d'heure (ou que vous soyez prof de maths).

Voyons donc l'addition pour commencer.
Pour faire une addition, on utilise le signe + (sans blague !).
Vous devez mettre le résultat de votre calcul dans une variable. On va donc par exemple créer une variable `resultat` de type `int` et faire un calcul :

Code : C

```
int 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

```
printf("5 + 3 = %d", resultat);
```

À l'écran, cela donnera :

Code : Console

```
5 + 3 = 8
```

Voilà pour l'addition.
Pour les autres opérations, c'est la même chose, seul le signe utilisé change (voir tab. suivante).

Signes des opérateurs	
Opération	Signe

Addition	+
Soustraction	-
Multiplication	*
Division	/
Modulo	%

Si vous avez déjà utilisé la calculatrice sur votre ordinateur, vous devriez connaître ces signes. 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, $6 / 3$ font 2, votre ordinateur vous donnera la réponse juste. Jusque-là pas de souci.

Mais prenons maintenant une division avec reste comme $5 / 2$... Le résultat devrait être 2.5. Et pourtant ! Regardez ce que fait ce code :

Code : C

```
int resultat = 0;


resultat = 5 / 2;
printf ("5 / 2 = %d", 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 (aussi appelée « division euclidienne »). Cela veut dire qu'il tronque le résultat, il ne garde que la partie entière (le 2).

 Hé mais je sais pourquoi ! C'est parce que `resultat` est un `int` ! Si ça avait été un `double`, il aurait pu stocker un nombre décimal à l'intérieur !

Eh non, ce n'est pas la raison ! Essayez le même code en transformant juste `resultat` en `double`, et vous verrez qu'on vous affiche quand même 2. Parce que les nombres de l'opération sont des nombres entiers, l'ordinateur répond par un nombre entier.

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 l'ordinateur considère que ce sont des nombres décimaux, donc il fait une division de nombres décimaux) :

Code : C

```
double resultat = 0;

resultat = 5.0 / 2.0;
printf ("5 / 2 = %f", resultat);
```

Code : Console

```
5 / 2 = 2.500000
```

Là, le nombre est correct. Bon : il affiche des tonnes de zéros derrière si ça lui chante, mais le résultat reste quand même correct.

Cette propriété de la division de nombres entiers est très importante. Il faut que vous reteniez que pour un ordinateur :

- $5 / 2 = 2$;
- $10 / 3 = 3$;
- $4 / 5 = 0$.

C'est un peu surprenant, mais c'est sa façon de calculer avec des entiers.

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 deux fois. De même, « combien de fois y a-t-il le nombre 3 dans 10 ? » Trois 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 quatre autres, mais pour votre ordinateur ça reste une opération de base... probablement pour justement 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 modules :

- $5 \% 2 = 1$;
- $14 \% 3 = 2$;
- $4 \% 2 = 0$.

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, il n'y a rien à ajouter au sujet des modules. Je tenais juste à l'expliquer à 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 cinq 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

```
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 !

Imaginez un programme qui demande deux 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-mêmes ce petit programme, c'est facile et ça vous entraînera !

La réponse est ci-dessous :

Code : C

```
int main(int argc, char *argv[])
{
    int resultat = 0, nombre1 = 0, nombre2 = 0;

    // On demande les nombres 1 et 2 à l'utilisateur :

    printf("Entrez le nombre 1 : ");
    scanf("%d", &nombre1);
    printf("Entrez le nombre 2 : ");
    scanf("%d", &nombre2);

    // On fait le calcul :

    resultat = nombre1 + nombre2;

    // Et on affiche l'addition à l'écran :

    printf ("%d + %d = %d\n", nombre1, nombre2, resultat);

    return 0;
}
```

Code : Console

```
Entrez le nombre 1 : 30
Entrez le nombre 2 : 25
30 + 25 = 55
```

Sans en avoir l'air, on vient de faire là notre premier programme ayant un intérêt. Notre programme est capable d'additionner deux nombres et d'afficher le résultat de l'opération !

Vous pouvez essayer avec n'importe quel nombre (du moment que vous ne dépassez pas les limites d'un type `int`), votre ordinateur effectuera le calcul en un éclair. Encore heureux parce que des opérations comme ça, il doit en faire des milliards en une seule seconde !

Je vous conseille de faire la même chose avec les autres opérations pour vous entraîner (soustraction, multiplication...). 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 trois variables à la fois, ça fonctionne sans problème :

Code : C

```
resultat = nombre1 + nombre2 + nombre3;
```

Les raccourcis

Comme promis, nous n'avons pas de nouvelles opérations à voir. Et pour cause ! Nous les connaissons déjà toutes. 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.

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'incrémementation.

L'incrémementation

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

```
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 deux 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'incrémementation**. Cette instruction produit exactement le même résultat que le code qu'on vient de voir :

Code : C

```
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 deux signes `+`, et bien entendu, de ne pas oublier le point-virgule.

Même 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'œil des programmeurs, et vous êtes maintenant capables de le comprendre ! C++ signifie qu'il s'agit du langage C « incrémenté », c'est-à-dire si on veut « du langage C à un niveau supérieur ». En pratique, le C++ permet surtout de programmer différemment mais il n'est pas « meilleur » que le C : juste différent.

La décrémementation

C'est tout bêtement l'inverse de l'incrémementation : on enlève 1 à une variable.

Même si on fait plus souvent des incrémentations que des décrémementations, cela reste une opération pratique que vous utiliserez de temps en temps.

La décrémementation, si on l'écrit en forme « longue » :

Code : C

```
nombre = nombre - 1;
```

Et maintenant en forme « raccourcie » :

Code : C

```
nombre--;
```

On l'aurait presque deviné tout seul ! Au lieu de mettre un `++`, vous mettez un `--` : si votre variable vaut 6, elle vaudra 5 après l'instruction de décrémementation.

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 deux une variable :

Code : C

```
nombre = nombre * 2;
```

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

Code : C

```
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

```
int 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)
```

(Ne boudez pas, un peu de calcul mental n'a jamais tué personne !)

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 un jour des lignes répétitives à taper dans un programme.

Retenez quand même que l'incrémementation reste de loin le raccourci le plus utilisé.

La bibliothèque mathématique

En langage C, il existe ce qu'on appelle des bibliothèques « standard », c'est-à-dire des bibliothèques toujours utilisables. Ce

sont en quelque sorte des bibliothèques « de base » qu'on utilise très souvent.

Les bibliothèques sont, je vous le rappelle, des ensembles de fonctions toutes prêtes. Ces fonctions ont été écrites par des programmeurs avant vous, elles vous évitent en quelque sorte d'avoir à réinventer la roue à chaque nouveau programme.

Vous avez déjà utilisé les fonctions `printf` et `scanf` de la bibliothèque `stdio.h`. Il faut savoir qu'il existe une autre bibliothèque, appelée `math.h`, qui contient de nombreuses fonctions mathématiques toutes prêtes.

En effet, les cinq opérations de base que l'on a vues 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 jeunes ou que vous n'avez pas assez fait de maths dans votre vie. Toutefois, la bibliothèque mathématique contient de nombreuses autres fonctions dont vous aurez très probablement besoin.

Tenez par exemple, on ne peut pas faire de puissances en C ! Comment calculer un simple carré ? Vous pouvez toujours essayer de taper `5^2` dans votre programme, mais votre ordinateur ne le comprendra jamais car il ne sait pas ce que c'est... À moins que vous le lui expliquiez en lui indiquant la bibliothèque mathématique !

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

Code : C

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

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

J'ai justement l'intention de vous les présenter. Bon : comme il y a beaucoup de fonctions, je ne peux pas en 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 fonctions principales, 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 ne dirait pas comme ça, mais en fait ça finit par servir !

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

```
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 cette fois dans `stdlib.h`.



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

```
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 deux valeurs : le nombre et la puissance à laquelle vous voulez l'élever. Voici le schéma de la fonction :

Code : C

```
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

```
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

```
double resultat = 0, nombre = 100;
resultat = sqrt(nombre); // resultat vaudra 10
```

sin, cos, tan

Ce sont les trois 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.

En résumé

- Un ordinateur n'est en fait qu'une **calculatrice géante** : tout ce qu'il sait faire, ce sont des opérations.
- Les opérations connues par votre ordinateur sont très **basiques** : l'addition, la soustraction, la multiplication, la division et le modulo (il s'agit du reste de la division).
- Il est possible d'**effectuer des calculs entre des variables**. C'est d'ailleurs ce qu'un ordinateur sait faire de mieux : il le fait bien et vite.
- L'**incrément**ation est l'opération qui consiste à ajouter 1 à une variable. On écrit `variable++`.
- La **décrément**ation est l'opération inverse : on retire 1 à une variable. On écrit donc `variable--`.
- Pour augmenter le nombre d'opérations connues par votre ordinateur, il faut charger la **bibliothèque mathématique** (c'est-à-dire `#include <math.h>`).
- Cette bibliothèque contient des **fonctions mathématiques plus avancées**, telles que la puissance, la racine carrée, l'arrondi, l'exponentielle, le logarithme, etc.

Les conditions

Nous avons vu dans le premier chapitre qu'il existait de nombreux langages de programmation. Certains se ressemblent d'ailleurs : un grand nombre d'entre eux sont inspirés du langage C.

En fait le langage C a été créé il y a assez longtemps, ce qui fait qu'il a servi de modèle à de nombreux autres plus récents. La plupart des langages de programmation ont finalement des ressemblances, ils reprennent les principes de base de leurs aînés.

En parlant de principes de base : nous sommes en plein dedans. Nous avons vu comment créer des variables, faire des calculs avec (concept commun à tous les langages de programmation !), nous allons maintenant nous intéresser aux **conditions**. Sans conditions, nos programmes informatiques feraient toujours la même chose !

La condition if... else

Les conditions permettent de 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 !

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 deux ou trois 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
<code>==</code>	est égal à
<code>></code>	est supérieur à
<code><</code>	est inférieur à
<code>>=</code>	est supérieur ou égal à
<code><=</code>	est inférieur ou égal à
<code>!=</code>	est différent de

 Faites très attention, il y a bien deux 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

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.

Écrivez 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

```
if (/* Votre condition */)
{
    // Instructions à exécuter si la condition est vraie
}
```

À 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

```
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 (comme c'est le cas ici), alors celles-ci deviennent facultatives. Je recommande néanmoins de toujours mettre des accolades pour des raisons de clarté.

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.

Cela peut paraître évident pour certains, mais plusieurs lecteurs visiblement perdus m'ont encouragé à ajouter cette explication. Voici donc un code complet que vous pouvez tester :

Code : C

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

int main(int argc, char *argv[])
{
    int age = 20;

    if (age >= 18)
    {
        printf ("Vous etes majeur !\n");
    }

    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.

Utilisez 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 fonctionnera aussi bien si vous écrivez tout sur une même ligne. Par exemple :

Code : C

```
if (age >= 18) { printf ("Vous etes majeur !"); }
```

Pourtant, même s'il est possible d'écrire comme ça, c'est **absolument déconseillé**.

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. Ça ne change rien au fonctionnement du programme final, mais c'est une question de « style informatique » si vous voulez. Si vous voyez le code de quelqu'un d'autre présenté un peu différemment, c'est qu'il code avec un style différent. Le principal dans tous les cas étant que le 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

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

```
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 place entre le **if** et le **else**.

On dit dans ce cas à l'ordinateur :

Citation

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

```
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"); // Langage bébé, vous pouvez pas comprendre
}
```

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, seul un **if** est nécessaire (logique me direz-vous, sinon il n'y a pas de condition !).

Notez qu'on peut mettre autant de **else if** que l'on veut. On peut donc écrire :

Citation

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 :

Symbole	Signification
&&	ET
	OU
!	NON

Test ET

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

Code : C

```
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 deux 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 pour l'exemple un programme stupide 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 beaucoup d'argent (parce que là, même à 10 ans on vous acceptera à bras ouverts !). Notre test pour savoir si le client a le droit d'ouvrir un compte en banque pourrait être :

Code : C

```
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

```
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 deux signes ==

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

Code : C

```
if (age == 18)
{
    printf ("Vous venez de devenir majeur !");
}
```

N'oubliez pas de mettre deux 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, non 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 fonctionne pas comme ils voudraient !

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

```
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 entrer plus en détails 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.

Quelques petits tests pour bien comprendre

Nous allons commencer par faire quelques petites expériences avant d'introduire cette nouvelle notion. Voici un code source très simple que je vous propose de tester :

Code : C

```
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. Faites un autre test en remplaçant 1 par 0 :

Code : C

```
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

```
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 fautive, il remplace `age >= 18` par le nombre 0, et du coup la condition est fautive : 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

```
int age = 20;
int majeur = 0;

majeur = age >= 18;
printf("Majeur vaut : %d\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 cela grâce à 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 nombres 0 et 1, pour dire si « quelque chose est faux ou vrai ».

En langage C, il n'existe pas de type de variable « booléen ».

Du coup, on est obligé d'utiliser un type entier comme `int` pour gérer les booléens.

Les booléens dans les conditions

Souvent, on fera un test `if` sur une variable booléenne :

Code : C

```
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 peut être lue facilement par un être humain. On voit `if (majeur)`, ce qui peut se 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 dès le début.

Tenez, voici un autre test imaginaire :

Code : C

```
if (majeur && garçon)
```

Ce test signifie « si tu es majeur ET que tu es un garçon ». `garçon` 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 !

Les booléens permettent donc de dire si quelque chose est vrai ou faux.

C'est vraiment 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 (comme un âge), 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

```
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

```
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 entrer 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 ?
```

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**, et enfin faites un **switch** pour dire à l'utilisateur « tu as choisi le menu Royal Cheese » par exemple.

Allez, au travail !

Correction

Voici la solution (j'espère que vous l'avez trouvée !) :

Code : C

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

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

    printf("=== Menu ===\n");
    printf("1. Royal Cheese\n");
    printf("2. Mc Deluxe\n");
    printf("3. Mc Bacon\n");
    printf("4. Big Mac\n");
    printf("\nVotre choix ? ");
    scanf("%d", &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, ça 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");

    return 0;
}
```

Et voilà le travail !

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 entrer 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 deux fois la même condition : la première avec un **if... else**, et la seconde, identique, mais sous forme d'une expression 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 stupide je suis d'accord, mais ça me permet de vous montrer comment on peut se servir des expressions ternaires.

Voici comment faire cela avec un **if... else** :

Code : C

```
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 ternaire :

Code : C

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

Les ternaires permettent, sur une seule ligne, de changer la valeur d'une variable en fonction d'une condition. Ici la condition est tout simplement `majeur`, mais ça pourrait être n'importe quelle condition plus longue bien entendu. Un autre exemple ?
`autorisation = (age >= 18) ? 1 : 0;`

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

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

En résumé

- Les **conditions** sont à la base de tous les programmes. C'est un moyen pour l'ordinateur de **prendre une décision** en fonction de la valeur d'une variable.
- Les mots-clés **if**, **else if**, **else** signifient respectivement « si », « sinon si », « sinon ». On peut écrire autant de **else if** que l'on désire.
- Un **booléen** est une variable qui peut avoir deux états : vrai (1) ou faux (0) (toute valeur différente de 0 est en fait considérée comme « vraie »). On utilise des **int** pour stocker des booléens car ce ne sont en fait rien d'autre que des nombres.
- Le **switch** est une alternative au **if** quand il s'agit d'analyser la valeur d'une variable. Il permet de rendre un code source plus clair si vous vous apprêtez à tester de nombreux cas. Si vous utilisez de nombreux **else if** c'est en général le signe qu'un **switch** serait plus adapté pour rendre le code source plus lisible.
- Les **ternaires** sont des conditions très concises qui permettent d'affecter rapidement une valeur à une variable en fonction du résultat d'un test. On les utilise avec parcimonie car le code source a tendance à devenir moins lisible avec elles.

Les boucles

Après avoir vu comment réaliser des conditions en C, nous allons découvrir les boucles. Qu'est-ce qu'une boucle ? C'est une technique permettant de répéter les mêmes instructions plusieurs fois. Cela nous sera bien utile par la suite, notamment pour le premier TP qui vous attend après ce chapitre.

Relaxez-vous : ce chapitre sera simple. Nous avons vu ce qu'étaient les conditions et les booléens dans le chapitre précédent, c'était un gros morceau à avaler. Maintenant ça va couler de source et le TP ne devrait pas vous poser trop de problèmes.

Enfin profitez-en, parce qu'ensuite nous ne tarderons pas à entrer dans la partie II du cours, et là vous aurez intérêt à être bien réveillés !

Qu'est-ce qu'une boucle ?

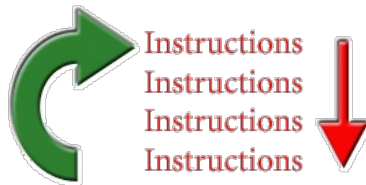
Je me répète : une boucle est une structure qui permet de répéter les mêmes instructions plusieurs fois.

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 trois types de boucles courantes en C :

- `while`
- `do... while`
- `for`

Dans tous les cas, le schéma est le même (fig. suivante).



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 n'est pas du genre à se plaindre, vous savez : il fait ce qu'on lui dit de faire... Il pourrait très bien se bloquer dans une boucle infinie, c'est d'ailleurs une des nombreuses craintes des programmeurs.

Et c'est là qu'on retrouve... les conditions ! Quand on crée une boucle, on indique toujours une condition. Cette condition signifiera « Répète la boucle tant que cette condition est vraie ».

Comme je vous l'ai dit, il y a plusieurs manières de s'y prendre. 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

```
while (/* Condition */)
{
    // Instructions à répéter
}
```

C'est aussi simple que cela. `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 lui redemande 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

```
int nombreEntre = 0;

while (nombreEntre != 47)
{
    printf("Tapez le nombre 47 ! ");
    scanf("%d", &nombreEntre);
}
```

Voici maintenant le test que j'ai fait. Notez que j'ai fait exprès de me tromper 2-3 fois avant de taper le bon nombre.

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 ? Ça consiste à ajouter 1 à la variable en faisant `variable++` ;.

Regardez attentivement ce bout de code et, surtout, essayez de le comprendre :

Code : C

```
int 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 !
```

```
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 `compteur++`. `compteur` valait 0, elle 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 inférieure à 10 ? ». Ben oui, `compteur` vaut 1 ! 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 d'ailleurs voir que la variable `compteur` augmente au fur et à mesure dans la boucle, en l'affichant dans le `printf` :

Code : C

```
int compteur = 0;

while (compteur < 10)
{
    printf("La variable compteur vaut %d\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 utile 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

```
while (1)
{
    printf("Boucle infinie\n");
}
```

Souvenez-vous des booléens : 1 = vrai, 0 = faux. Ici, la condition est toujours vraie, ce programme affichera donc « 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

```
int 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.



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**, sinon votre programme plantera à la compilation !

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 utile 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**, si ce n'est plus, il vous faudra donc 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

```
int compteur = 0;

while (compteur < 10)
{
```

```
printf("Salut les Zeros !\n");
compteur++;
}
```

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

Code : C

```
int compteur;

for (compteur = 0 ; compteur < 10 ; compteur++)
{
    printf("Salut les Zeros !\n");
}
```

Quelles différences ?

- Vous noterez que l'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 trois instructions condensées, chacune séparée 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 non. Tant que la condition est vraie, la boucle **for** continue.
- Enfin, il y a l'**incrément** : 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 !

En résumé

- Les **boucles** sont des structures qui nous permettent de répéter une série d'instructions plusieurs fois.
- Il existe plusieurs types de boucles : **while**, **do... while** et **for**. Certaines sont plus adaptées que d'autres selon les cas.
- La boucle **for** est probablement celle qu'on utilise le plus dans la pratique. On y fait très souvent des incrémentations ou des décréments de variables.



TP : Plus ou Moins, votre premier jeu

Nous arrivons maintenant dans le premier TP. Le but est de vous montrer que vous savez faire des choses avec ce que je vous ai appris. Car en effet, la théorie c'est bien, mais si on ne sait pas mettre tout cela en pratique de manière concrète... ça ne sert à rien d'avoir passé tout ce temps à apprendre.

Croyez-le ou non, vous avez déjà le niveau pour réaliser un premier programme amusant. C'est un petit jeu en mode console (les programmes en fenêtres arriveront plus tard je vous le rappelle). Le principe du jeu est simple et le jeu est facile à programmer. C'est pour cela que j'ai choisi d'en faire le premier TP du cours.

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.

1. L'ordinateur tire au sort un nombre entre 1 et 100.
2. Il vous demande de deviner le nombre. Vous entrez donc un nombre entre 1 et 100.
3. L'ordinateur compare le nombre que vous avez entré 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.
4. Puis l'ordinateur vous redemande le nombre.
5. ... Et il vous indique si le nombre mystère est supérieur ou inférieur.
6. Et ainsi de suite, jusqu'à ce que vous trouviez 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 ne 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 !

Bon, on a donc deux solutions.

- Soit on demande à l'utilisateur d'entrer le nombre mystère via un `scanf` d'abord. Ça implique qu'il y ait deux joueurs : l'un entre 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 ensuite si vous voulez.

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).

Pour ce faire, on va utiliser la formule suivante (je ne pouvais pas trop vous demander de la deviner !) :

Code : C

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

La première ligne (avec `srand`) permet d'initialiser le générateur de nombres aléatoires. Oui, c'est un peu compliqué, je vous avais prévenus. `nombreMystere` est une variable qui contiendra le nombre tiré 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` deux 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

```
const int MAX = 100, MIN = 1;
```

Les bibliothèques à inclure

Pour que votre programme fonctionne correctement, vous aurez besoin d'inclure trois bibliothèques : `stdlib`, `stdio` et `time` (la dernière sert pour les nombres aléatoires).

Votre programme devra donc commencer par :

Code : C

```
#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 !



Pour vous faire générer des nombres aléatoires, j'ai été obligé de vous donner des codes « tout prêts », sans vous expliquer totalement comment ils fonctionnent. 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 notions 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.

À vous de jouer ! Bonne chance !

Correction !

Stop ! À partir d'ici je ramasse les copies.

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, sachez que c'est probablement aussi bien.

La correction de « Plus ou Moins »

Voici la correction que je vous propose :

Code : C

```
#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.

Télécharger l'exécutable et les sources de
"Plus ou Moins" (7 Ko)



L'exécutable (.exe) est compilé pour Windows, donc si vous êtes sous un autre système d'exploitation il faudra



obligatoirement recompiler le programme pour qu'il marche chez vous.

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, etc.) et créez un nouveau projet de type console, vide. Une fois que c'est fait, demandez 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 avez réussi à faire une erreur là, vous êtes très forts.

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 pour 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 !).

À chaque passage dans la boucle, on redemande à l'utilisateur d'entrer un nombre. On stocke le nombre qu'il propose dans `nombreEntre`.

Puis, on compare ce `nombreEntre` au `nombreMystere`. Il y a trois 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 ? Eh bien... 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**.

Quand ces deux nombres sont égaux (c'est-à-dire quand on a trouvé), la boucle s'arrête. Le programme est alors terminé.

Idées d'amélioration

Vous ne croyiez tout de même pas qu'on allait s'arrêter là ? Je veux vous inciter à continuer à améliorer ce programme, pour vous entraîner. N'oubliez pas que c'est en vous entraînant comme cela 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-là je ne manque pas d'idées pour l'améliorer !

Attention : cette fois je ne vous fournis pas de correction, il faudra vous débrouiller tout seuls ! Si vous avez vraiment des problèmes, n'hésitez pas à aller faire un tour sur les [forums du Site du Zéro](#), section langage C. 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 a 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 mettez 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 qui l'intéresse. 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`... Eh 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. 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 sont à votre disposition si vous avez des questions.

Les fonctions

Nous terminerons la partie I du cours (« Les bases ») par cette notion fondamentale que sont les fonctions en langage C. Tous les programmes en C se basent sur le principe que je vais vous expliquer dans ce chapitre.

Nous allons apprendre à structurer nos programmes en petits bouts... un peu comme si on jouait aux Legos. Tous les gros programmes en C sont en fait des assemblages de petits bouts de code, et ces petits bouts de code sont justement ce qu'on appelle... des fonctions !

Créer et appeler une fonction

Nous avons vu dans les tout premiers chapitres qu'un programme en C commençait par une fonction appelée `main`. Je vous avais même fait un schéma récapitulatif, pour vous rappeler quelques mots de vocabulaire (fig. suivante).

```
#include <stdio.h>
#include <stdlib.h> } Directives de préprocesseur

int main()
{
    printf("Hello world!\n");
    return 0; } Instructions } Fonction
```

En haut, on y trouve 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`. Je vous rassure, c'est toujours vrai ! Seulement, jusqu'ici nous sommes restés à l'intérieur de la fonction `main`. Nous n'en sommes jamais sortis. Revoyez vos codes sources et vous verrez : nous sommes toujours restés à l'intérieur des accolades de la fonction `main`.



Eh bien, c'est mal d'avoir fait comme 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 bazar...

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.

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. La fig. suivante représente une fonction schématiquement.



Lorsqu'on appelle une fonction, il y a trois 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 (fig. suivante). Bien entendu, les fonctions seront en général plus compliquées.



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

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

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 « Hé ! 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.

Schéma d'une fonction


Vous avez déjà eu un aperçu de la façon dont 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.

Le code suivant représente une fonction schématiquement. C'est un modèle à connaître :

Code : C

```
type nomFonction(parametres)
{
    // Insérez vos instructions ici
}
```

Vous reconnaissez 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 deux sortes de fonctions :
 - les fonctions qui renvoient une valeur : on leur met un des types que l'on connaît (`char`, `int`, `double`, etc.);
 - 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.
-  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.

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.

- Ensuite vous avez les **accolades** qui indiquent le début et la fin de la fonction. À l'intérieur de ces accolades vous mettrez les instructions que vous voulez. 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 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 `int` et qu'elle renvoie un nombre entier aussi de type `int`. Cette fonction calcule le triple du nombre qu'on lui donne :

Code : C

```
int triple(int nombre)
{
    int resultat = 0;

    resultat = 3 * nombre; // On multiplie le nombre fourni par 3
    return resultat;       // On retourne la variable resultat qui
    // vaut le triple de nombre
}
```

Voilà notre première fonction ! Une première chose importante : comme vous le voyez, la fonction est de type `int`. Elle doit donc renvoyer une valeur de type `int`.

Entre les parenthèses, vous avez les variables que la fonction reçoit. Ici, notre fonction `triple` reçoit une variable de type `int` appelée `nombre`.

La ligne qui donne pour consigne de « renvoyer une valeur » est celle qui contient le `return`. Cette ligne se trouve généralement à la fin de la fonction, après les calculs.

Code : C

```
return resultat;
```

Ce code signifie pour la fonction : « Arrête-toi là et renvoie le nombre `resultat` ». Cette variable `resultat` DOIT être de type `int`, car la fonction renvoie un `int` 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 d'écrire notre fonction `triple` ? Non, on peut faire tout cela en une ligne en fait :

Code : C

```
int triple(int 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 acceptant plusieurs paramètres. Par exemple, une fonction `addition` qui additionne deux nombres `a` et `b` :

Code : C

```
int addition(int a, int 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 forcément toujours le même texte puisque la fonction ne reçoit aucun paramètre susceptible de modifier son comportement !

Imaginons une fonction `bonjour` qui affiche juste « Bonjour » à l'écran :

Code : C

```
void bonjour()
{
    printf("Bonjour");
}
```

Je n'ai rien mis entre parenthèses car la fonction ne prend aucun paramètre. De plus, j'ai utilisé 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 s'entraîner un peu avec ce qu'on vient d'apprendre. Nous allons utiliser notre fonction `triple` (décidément 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 placez après, ça ne marchera pas. Je vous expliquerai pourquoi par la suite.

Voici un code à tester et à comprendre :

Code : C

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

int triple(int nombre)
{
    return 3 * nombre;
}

int main(int argc, char *argv[])
{
    int nombreEntree = 0, nombreTriple = 0;

    printf("Entrez un nombre... ");
    scanf("%d", &nombreEntree);
```

```

nombreTriple = triple(nombreEntre);
printf("Le triple de ce nombre est %d\n", nombreTriple);

return 0;
}

```

Notre programme commence par la fonction `main` comme vous le savez. On demande à l'utilisateur d'entrer un nombre. On envoie ce nombre qu'il a entré 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

```
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 ordonne 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 code particulièrement commenté vous indique 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 !) :

Code : C

```

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

int triple(int nombre) // 6
{
    return 3 * nombre; // 7
}

int main(int argc, char *argv[]) // 1
{
    int nombreEntre = 0, nombreTriple = 0; // 2

    printf("Entrez un nombre... "); // 3
    scanf("%d", &nombreEntre); // 4

    nombreTriple = triple(nombreEntre); // 5
    printf("Le triple de ce nombre est %d\n", nombreTriple); // 8

    return 0; // 9
}

```

Voici ce qui se passe, ligne par ligne.

1. Le programme commence par la fonction `main`.
2. Il lit les instructions dans la fonction une par une dans l'ordre.
3. Il lit l'instruction suivante et fait ce qui est demandé (`printf`).
4. De même, il lit l'instruction et fait ce qui est demandé (`scanf`).
5. Il lit l'instruction... Ah ! On appelle la fonction `triple`, on doit donc sauter à la ligne de la fonction `triple` plus haut.
6. On saute à la fonction `triple` et on récupère un paramètre (`nombre`).

7. On fait des calculs sur le nombre et on termine la fonction. **return** signifie la fin de la fonction et permet d'indiquer le résultat à renvoyer.
8. On retourne dans le `main` à l'instruction suivante.
9. Un **return** ! La fonction `main` se termine et donc le programme est terminé.

Si vous avez compris dans quel ordre l'ordinateur lit 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 (fig. suivante).

2) La fonction `triple` retourne (return) une valeur. Cette valeur, c'est 3x le nombre qu'on lui a envoyé.

Cette valeur de retour est stockée dans la variable `nombreTriple` de la fonction `main`. Le signe « = » permet donc de dire « Envoie le résultat de la fonction dans cette variable ».

```

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

```

```

int triple(int nombre)
{
    return 3 * nombre;
}

```

```

int main(int argc, char *argv[])
{
    int nombreEntre = 0, nombreTriple = 0;

```

```

    printf("Entrez un nombre... ");
    scanf("%d", &nombreEntre);

```

```

    nombreTriple = triple(nombreEntre);
    printf("Le triple de ce nombre est %d\n", nombreTriple);

    return 0;
}

```

1) La variable `nombreEntre` est envoyée en paramètre à la fonction `triple`. Celle-ci récupère cette variable dans une autre variable qui s'appelle « nombre ».

Note : on aurait aussi pu mettre le même nom de variable dans les 2 fonctions. Il n'y aurait pas eu de conflit, car une variable appartient à sa fonction.

Note : ce n'est pas le cas de 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 :

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 il y a un changement au niveau du dernier `printf`. De plus, on n'a pas déclaré de variable `nombreTriple` car on ne s'en sert plus :

Code : C

```

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

```

```

int triple(int nombre)
{
    return 3 * nombre;
}

```

```

int main(int argc, char *argv[])
{
    int nombreEntre = 0;

```

```
printf("Entrez un nombre... ");
scanf("%d", &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 %d\n", triple(nombreEntre));

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 plus fin dans tout ça, 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 Lego.

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. Ça va permettre de simplifier considérablement l'écriture de nos programmes et ça croyez-moi on en aura bien besoin !

Des 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...

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 rien de nouveau, mais ce 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 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` car on va forcément manipuler des nombres décimaux. Lisez-la attentivement :

Code : C

```
double conversion(double euros)
{
    double francs = 0;

    francs = 6.55957 * euros;
    return francs;
}

int main(int argc, char *argv[])
{
    printf("10 euros = %f\n", conversion(10));
    printf("50 euros = %f\n", conversion(50));
}
```

```
printf("100 euros = %f\n", conversion(100));
printf("200 euros = %f\n", conversion(200));

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 prévenus. 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 écrire `conversion(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).

À vous de jouer ! Écrivez une seconde fonction (toujours avant la fonction `main`) qui fera elle la conversion inverse : Francs => Euros. Ce ne sera pas bien difficile, il 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

```
void punition(int nombreDeLignes)
{
    int i;

    for (i = 0 ; i < nombreDeLignes ; i++)
    {
        printf("Je ne dois pas recopier mon voisin\n");
    }
}

int main(int argc, char *argv[])
{
    punition(10);

    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
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`. À part ça, il n'y a rien de bien différent.

Il aurait été bien plus intéressant de créer une fonction `puniton` qui s'adapte à n'importe quelle sanction. On lui aurait envoyé deux 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 n'auriez pas vu, je vous rappelle qu'on n'a fait que manipuler des variables contenant des nombres depuis le début du cours !). 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 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 nommée `aireRectangle` va prendre deux paramètres : la largeur et la hauteur. Elle renverra l'aire.

Code : C

```
double aireRectangle(double largeur, double hauteur)
{
    return largeur * hauteur;
}

int main(int argc, char *argv[])
{
    printf("Rectangle de largeur 5 et hauteur 10. Aire = %f\n",
    aireRectangle(5, 10));
    printf("Rectangle de largeur 2.5 et hauteur 3.5. Aire = %f\n",
    aireRectangle(2.5, 3.5));
    printf("Rectangle de largeur 4.2 et hauteur 9.7. Aire = %f\n",
    aireRectangle(4.2, 9.7));

    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

```
void aireRectangle(double largeur, double hauteur)
```

```
{
    double aire = 0;

    aire = largeur * hauteur;
    printf("Rectangle de largeur %f et hauteur %f. Aire = %f\n",
    largeur, hauteur, aire);
}

int main(int argc, char *argv[])
{
    aireRectangle(5, 10);
    aireRectangle(2.5, 3.5);
    aireRectangle(4.2, 9.7);

    return 0;
}
```

Comme vous le voyez, le `printf` est à l'intérieur de la fonction `aireRectangle` et produit 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

```
int menu()
{
    int 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 sucrés a la sauce de myrtilles
        enrobée de chocolat\n");
        printf("3 : Escalope de kangourou saignante et sa gelée aux
        fraises poivrées\n");
        printf("4 : La surprise du Chef (j'en salive
        d'avance...) \n");
        printf("Votre choix ? ");
        scanf("%d", &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 êtes un
```

```
sacre aventurier dites donc !\n");
    break;
}

return 0;
}
```

J'en ai profité pour améliorer le menu (par rapport à ce qu'on faisait habituellement) : la fonction `menu` affiche à nouveau le menu tant que l'utilisateur n'a pas entré 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 une méthode rapide et pratique.

À vous de jouer ! Le code est encore améliorable : on pourrait afficher un message d'erreur si l'utilisateur entre un mauvais nombre plutôt que de simplement afficher une nouvelle fois le menu.

En résumé

- Les fonctions s'appellent entre elles. Ainsi, le `main` peut appeler des fonctions toutes prêtes telles que `printf` ou `scanf`, mais aussi des fonctions que nous avons créées.
- Une fonction récupère en entrée des variables qu'on appelle **paramètres**.
- Elle effectue certaines opérations avec ces paramètres puis retourne en général une valeur à l'aide de l'instruction `return`.