



**ECOLE NATIONALE
DE L'AVIATION CIVILE**

Langage C

Partie 1

Emmanuel Romagnoli

- Présentation du langage C
- L'environnement de développement Code::Blocks
- Variables
- Constantes
- Expressions et opérateurs
- Instructions d'affichage et de saisie
- Structure d'un programme en langage C
- Structures de contrôle conditionnelles
- Structures de contrôle itératives
- Exercices

Présentation du langage C



Un partenariat MIT – AT&T – Honeywell

Dans le cadre du projet MAC (Mathematics and Computation), lancé le 1^{er} juillet 1963, le MIT a mis en place un partenariat avec :

- les laboratoires Bell d'AT&T (le géant américain des télécoms) ;
- la société Honeywell qui fournit l'ordinateur GE 645.



Voir http://en.wikipedia.org/wiki/MIT_Computer_Science_and_Artificial_Intelligence_Laboratory

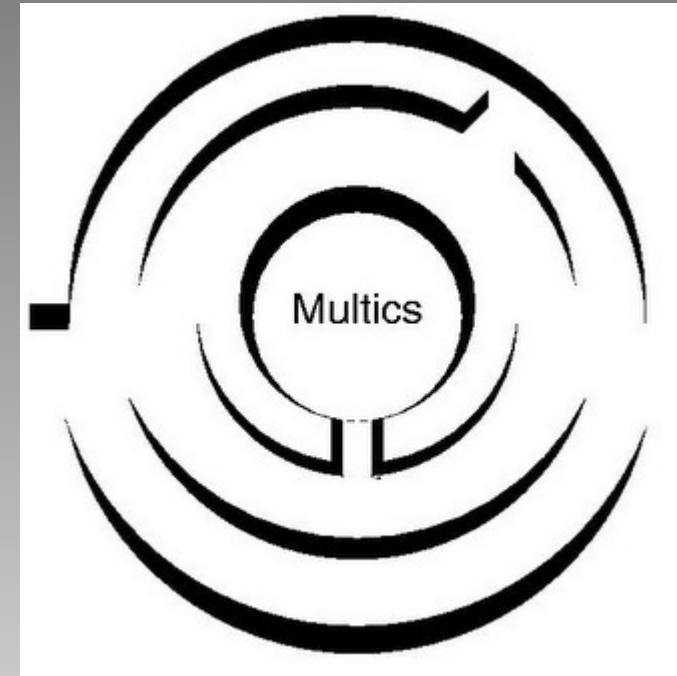
Un nouvel OS multi-utilisateur

Ce nouveau système d'exploitation, appelé MULTiplexed Information and Computing Service (MULTICS), devait permettre à une centaine de personnes de partager ce GE 645.

Ce système était développé en PL/1 sous forme d'anneaux concentriques afin de mieux gérer les droits d'accès des utilisateurs aux ressources.

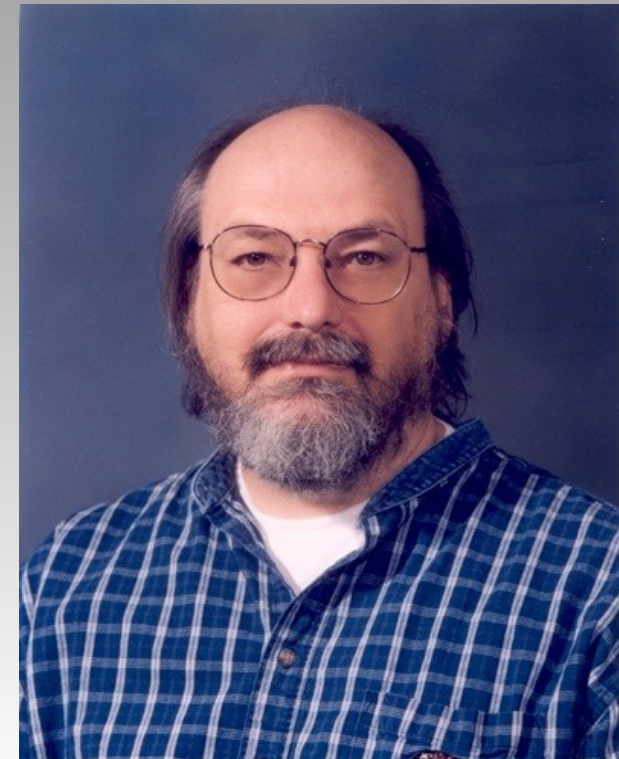
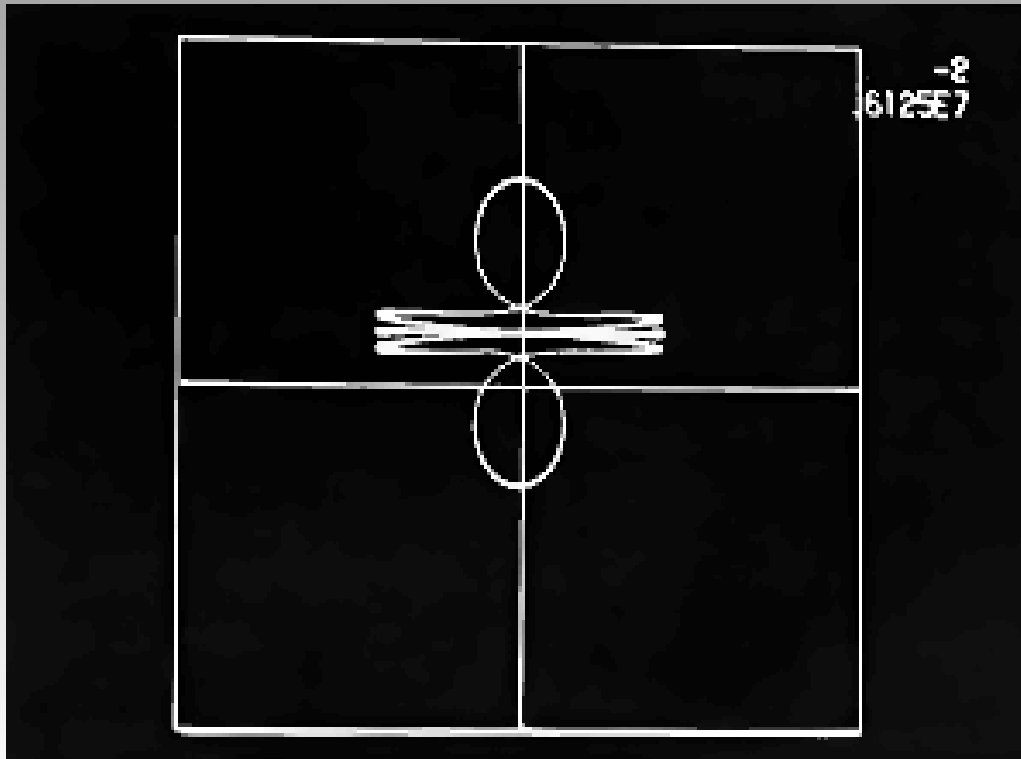
Multics fut mis en service en 1969 après de nombreuses difficultés et le retrait des partenaires industriels.

Voir <http://www.multicians.org/>



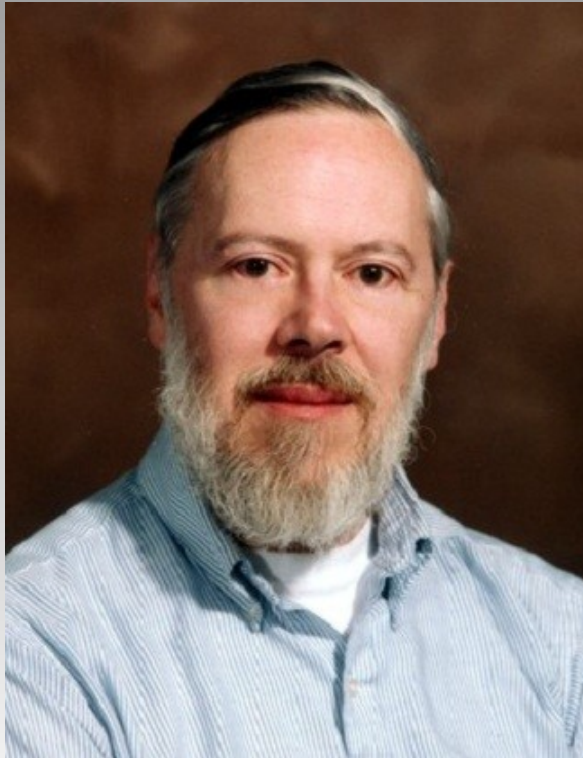
Space Travel de Ken Thompson

Ken Thompson a développé un jeu, appelé Space Travel, sur MULTICS en 1969. Quand les Bell Labs (auxquels il appartient) décident de quitter le projet, il entreprend de porter le jeu sur PDP-7.



Dennis Ritchie

Pour cela, il développe, avec l'aide de Dennis Ritchie, un système d'exploitation, en assembleur, sur un PDP-7, en s'inspirant des concepts de MULTICS.



Voir <http://www.faqs.org/docs/artu/ch02s01.html>
<http://www.uvlist.net/game-164857-Space+Travel>

L'arrivée d'UNICS

Le système, opérationnel en **1970**, fut d'abord baptisé « UNIplicated Information and Computing Service » (UNICS) par Brian Kernighan, en opposition à MULTICS, puis **Unix**.



Voir <http://www.cs.princeton.edu/~bwk/>
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.100.7314&rank=1>

Un besoin de ré-écriture d'Unix

Pour faciliter le portage d'Unix, celui-ci doit être ré-écrit dans un langage de haut-niveau (il ne reste alors plus qu'à le compiler sur la machine cible).

Thompson créé d'abord, en 1970, une version simplifiée d'un langage appelé BCPL pour le PDP-7 : ce langage s'est appelé le langage B.

Lorsque les Bell Labs ont affecté un PDP-11 au projet Unix, le langage B a montré ses limites (il ne savait manipuler que des mots de 16 bits et non des octets).

Voir <http://fr.wikipedia.org/wiki/BCPL>
<http://c.developpez.com/cours/historique-langage-c/>
<http://www.corp.att.com/history/milestones.html>

Des octets et des pointeurs

Denis Ritchie écrit un langage « new B » qui permettait de manipuler les **octets**. Il a également ajouté d'autres éléments comme les **pointeurs** ce qui en a fait un nouveau langage qui fut finalement baptisé « langage C ».

En **1973**, **Unix version 2** est donc implémenté en **C**.

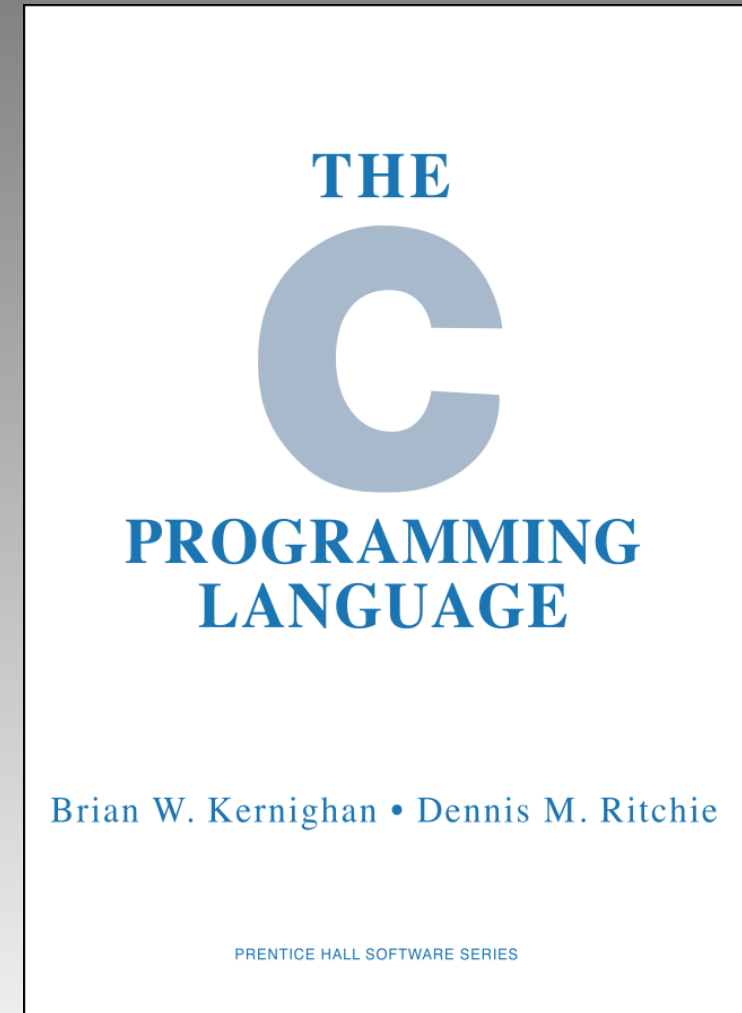
Comme un décret de 1956, interdit à AT&T de commercialiser autre chose que des équipements téléphoniques et télégraphiques, l'entreprise met le code source d'Unix des différentes versions à disposition des universités.

Voir <http://fr.wikipedia.org/wiki/BCPL>
<http://c.developpez.com/cours/historique-langage-c/>
<http://www.corp.att.com/history/milestones.html>

Le premier ouvrage de référence

Brian Kernighan et Denis Ritchie publient, en 1978, le premier ouvrage de description de ce nouveau langage afin d'en faciliter la diffusion (en même temps que l'OS Unix).

« The C Programming Language » (encore appelé « K&R ») fait donc référence au langage C « originel » qui est aussi appelé « **K&R C** ».



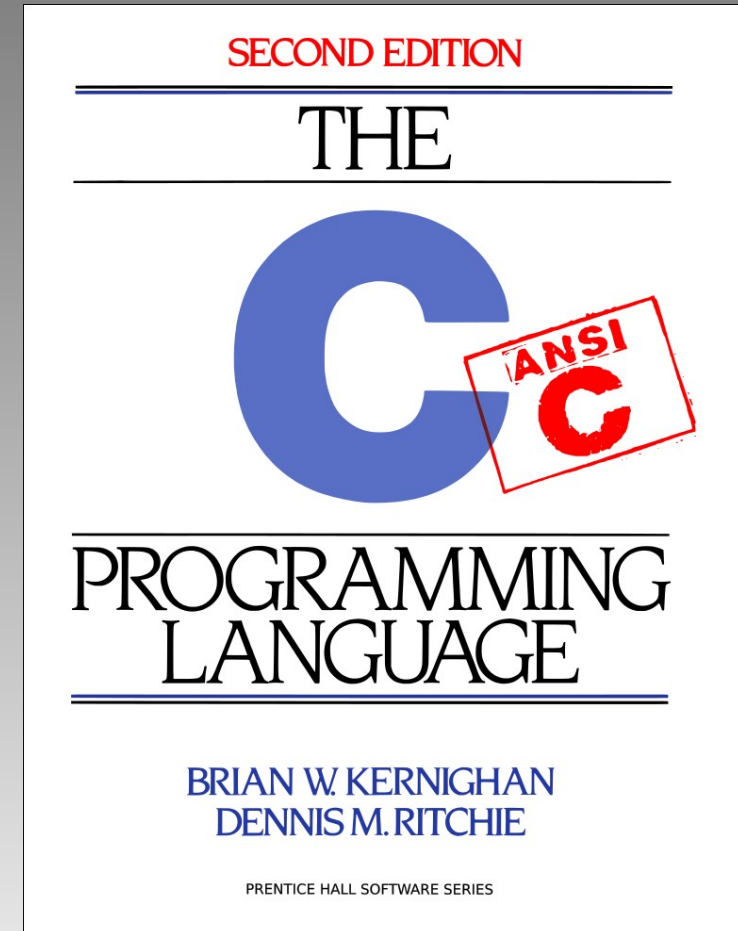
Voir http://en.wikipedia.org/wiki/The_C_Programming_Language

Une première normalisation par l'ANSI

En 1983, l'ANSI (l'équivalent de l'AFNOR aux États-Unis) crée le comité X3J11 afin de normaliser le langage C.

La norme ANSI X3,15-1989, ratifiée en 1989, est aussi connue sous le nom « **C89** ».

Une seconde édition de l'ouvrage K&R est alors publiée pour tenir compte de cette normalisation.



Voir http://en.wikipedia.org/wiki/ANSI_C

Une normalisation internationale par l'ISO

L'ANSI propose cette nouvelle norme à l'ISO (International Organization for Standardization), un organisme lié à l'ONU, dans lequel siègent des représentants d'organismes nationaux de normalisation (ANSI, AFNOR, DIN...)

La norme ISO/IEC 9899:1990, connue également sous le nom « **C90** », a été ratifiée, en 1990, après quelques petits changements mineurs.

Voir http://en.wikipedia.org/wiki/ANSI_C

L'ISO a fait évoluer la norme :

- ISO/IEC 9899/AMD1:1995 ou « **C95** » :
 - quelques corrections ;
 - introduction de nouvelles macro...
- ISO/IEC 9899:1999 ou « **C99** » :
 - nouveaux types de données ;
 - fonctions inlines ;
 - utilisation des unicodes pour le nommage...
- ISO/IEC 9899:2011 ou « **C11** » :
 - gestion du multi-threading

Voir http://www.iso-9899.info/wiki/The_Standard

- ISO/IEC 9899:2018 ou « **C18** » :
 - quelques corrections par rapport à la C11
- « **C2x** » devrait être voté en 2021

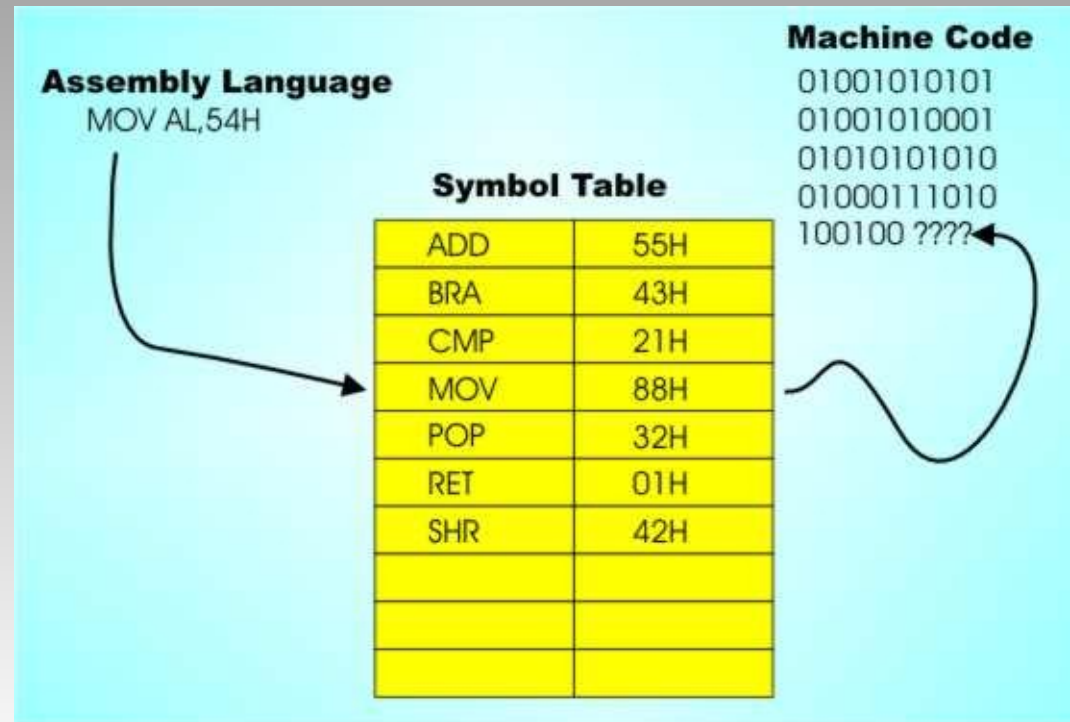
Langage de première génération

Il est d'usage de classer les langages de programmation par génération. Les langages de première génération (**1GL**) consistaient à bouger des interrupteurs pour programmer l'ordinateur en **langage machine** (concrètement à introduire des 0 et des 1 dans la mémoire de l'ordinateur).



Langage de deuxième génération

Par la suite, on a créé le langage assembleur composés de mnémoniques qui étaient directement convertis en langage machine. Le premier assembleur était « Initial Order » pour l'EDSAC (1949) avec des mnémonique à une lettre.



Voir <http://www.i-programmer.info/babbages-bag/301-assemblers-compilers-and-interpreters.html>
<http://www.cl.cam.ac.uk/~mr10/edsacposter.pdf>

L'arrivée des langages de haut-niveau

En **1953**, John Backus crée le premier langage de **haut niveau** (Speedcoding) pour les machines IBM 701. Il servira de base à la création, en 1954, du **FORTRAN** (FORMula TRANSlator) sur l'IBM 704.

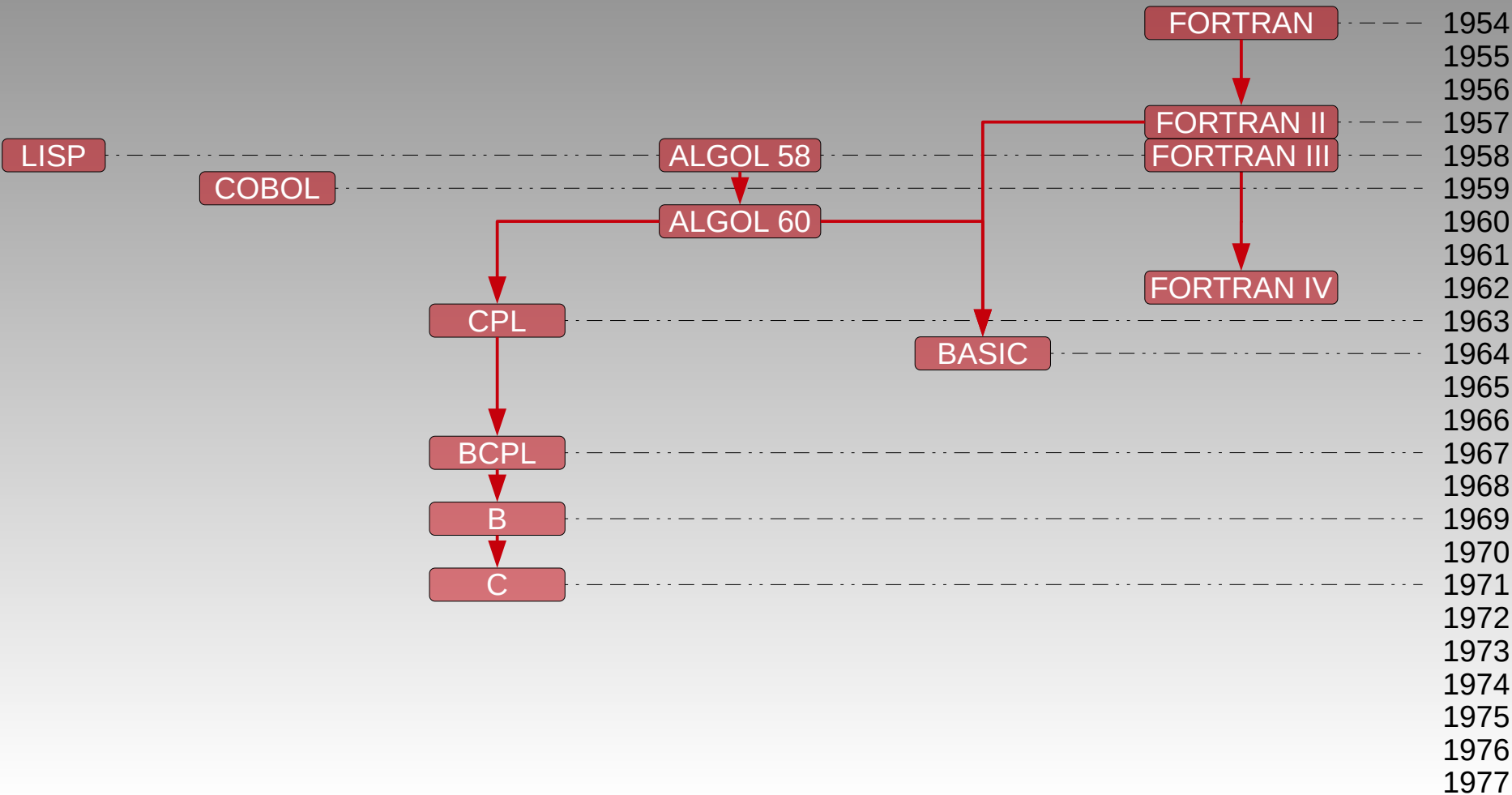


Le compilateur générerait un code assembleur ayant des performances comparables à celui écrit à la main. Son utilisation permet de **réduire la taille du code d'un facteur 20**.

Voir <http://dirkgerrits.com/wp-content/uploads/john-backus.pdf>
http://www-03.ibm.com/ibm/history/exhibits/builders/builders_backus.html
<http://community.computerhistory.org/scc/projects/FORTRAN/paper/p4-backus.pdf>
<http://www.softwarepreservation.org/projects/FORTRAN/BackusEtAl-Preliminary Report-1954.pdf>
<http://scarl.sewanee.edu/cs101/Lecture/6-History Of Computers/next.html>
<http://en.wikipedia.org/wiki/Fortran>

Un arbre généalogique complexe

A la suite de FORTRAN, de nombreux autres langages, dont le C, ont été créés par améliorations successives.



Voir <http://www.levenez.com/lang/>

3^{ème}, 4^{ème} et 5^{ème} générations de langages

Les langages de **3^{ème} génération**, comme **le C**, le Pascal, le Visual BASIC, le C++, le C#, le Java..., permettent d'implémenter des **algorithmes** qui sont ensuite transformés en langage assembleur et enfin en langage machine.

D'autres langages, plus proches du langage naturel comme le SQL, Scilab utilisent des **requêtes** qui sont ensuite transformés en algorithmes : ils forment la **4^{ème} génération**.

Les langages de **5^{ème} génération**, comme Prolog, n'utilisent que **des contraintes et des objectifs** afin de déterminer une solution (programmation par contrainte).

Voir http://en.wikipedia.org/wiki/Third-generation_programming_language
http://en.wikipedia.org/wiki/Fourth-generation_programming_language
http://en.wikipedia.org/wiki/Fifth-generation_programming_language

Du code source au code machine

Le langage C est un langage de haut-niveau avec des mots-clés en langue Anglaise.

On est donc très éloignés des « 0 » et des « 1 », seuls éléments que comprennent le microprocesseur.

Il est donc nécessaire de mettre en place un mécanisme de traduction.

Code source

```
void main ()  
{  
    printf ("Hello");  
}
```



```
0110100010110101001  
0001001010100010011  
1010100100101010010  
0010001010001001010  
0010111010101010111  
0110101100110011000  
1110101001010101101  
0101011010011101011
```

L'interprétation

Un premier mécanisme consiste à traduire les instructions **pendant l'exécution**, par un **interpréteur**.

Il en résulte :

- un mécanisme **lent** (on traduit et on exécute à chaque fois qu'on lance le programme) ;
- une exécution directe sans fichier intermédiaires ;
- un langage de programmation **très portable** (un seul code source peut fonctionner sous Windows, Mac OS X, Linux..., il faut cependant posséder l'interpréteur adapté).

Exemples : BASIC, Javascript, Python, Perl, PHP, Ruby

La compilation

Un deuxième mécanisme consiste à traduire les instructions **avant l'exécution**, par un **compilateur** qui génère un autre fichier appelé un **exécutable** (on dit aussi que le compilateur produit du **code « natif »**).

Il en résulte :

- un mécanisme **rapide** (on traduit une seule fois et on exécute la version compilée du programme) ;
- besoin d'un fichier intermédiaire ;
- un langage de programmation **un peu moins portable** (souvent besoin d'adapter le code source au système).

Exemples : Pascal, **C**, C++

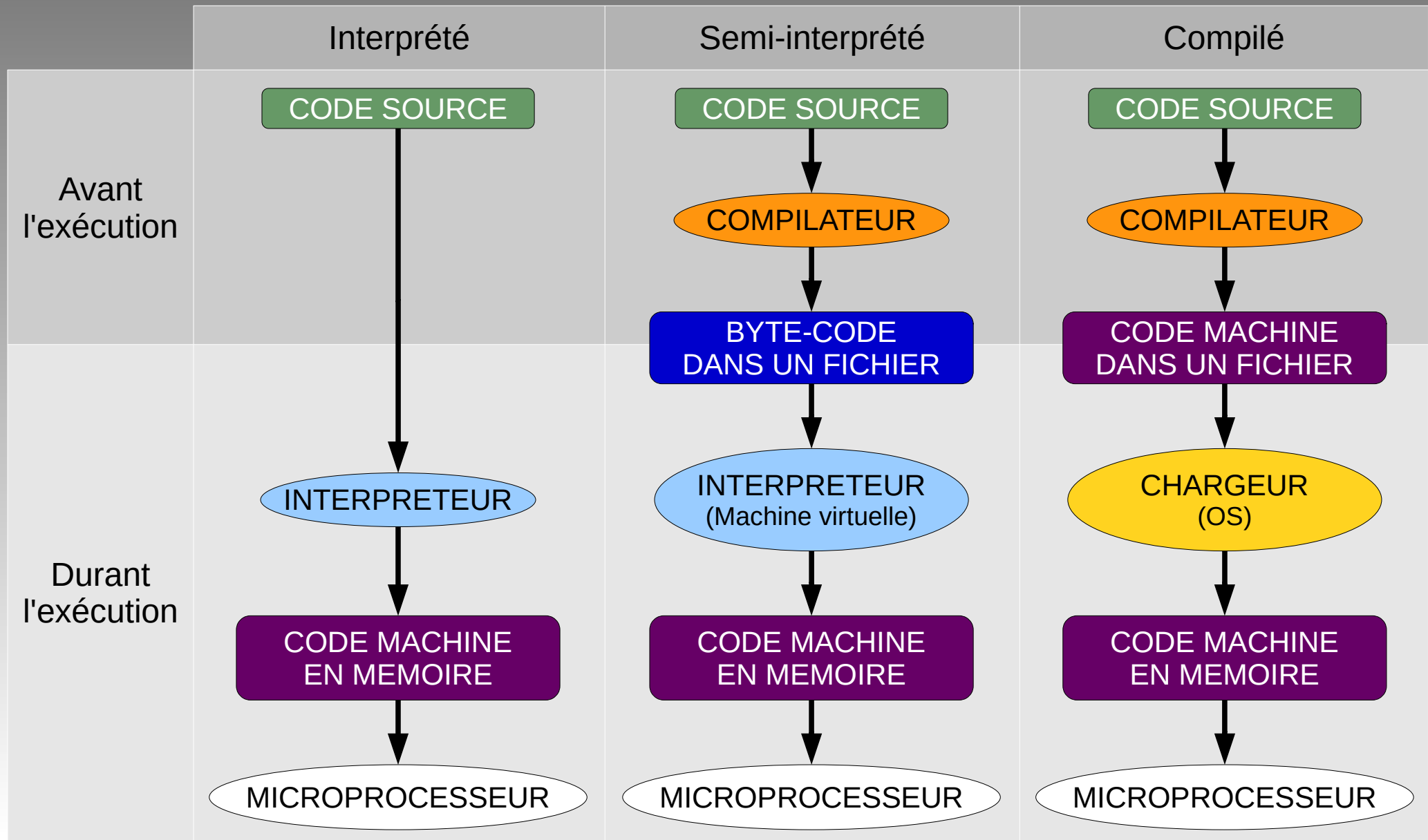
Le semi-interprété

Un troisième mécanisme a été mis en œuvre, notamment avec l'arrivée du langage **Java**. Il consiste à compiler le code source en un autre code, appelée « **byte-code** » qui est ensuite interprété par une **machine virtuelle**.

Le byte-code est organisé (optimisé) de manière à fortement faciliter la phase d'interprétation.

Il en résulte un code **très portable** (il faut disposer de la machine virtuelle adaptée au système) avec un fonctionnement **plutôt rapide** (un peu moins que les langages purement compilés).

Synthèse des 3 mécanismes



Différents paradigmes de programmation

Il existe différents paradigmes de programmation :

- La programmation **impérative** décrite juste après.
- La programmation orientée objets qui consiste à organiser le code en éléments logiciels qui interagissent entre eux.
- La programmation déclarative qui s'appuie sur l'utilisation de fonctions mathématiques, de prédicats...

La programmation impérative

La programmation impérative est basée sur des instructions qui modifient l'état du programme en cours d'exécution (affectation, opération arithmétique...).

C'est le type de programmation le plus courant qu'on retrouve même au niveau du micro-processeur :

- l'instruction MOV est l'équivalent d'une affectation
- l'instruction ADD permet de réaliser une addition...

Cette catégorie regroupe donc des langages aussi variés que le FORTRAN, l'Assembleur, le **C**, le BASIC, le Pascal...

Elle comporte des sous-catégories comme la programmation structurée et la programmation procédurale.

La programmation structurée

Les langages de programmation impérative comportent des instructions comme GOTO qui permettent de « sauter » à n'importe quel endroit du programme.

Mal utilisée, cette instruction peut conduire à concevoir des programmes illisibles : on parle de programmation spaghetti.

Cette situation a été dénoncée par des chercheurs comme Dijkstra : elle a conduit à la programmation **structurée** (utilisé en langage C) qui s'appuie sur :

- les structures de contrôle ;
- les blocs de codes

Voir www.ifi.unizh.ch/req/courses/kvse/uebungen/Dijkstra_Goto.pdf

La programmation procédurales

Tout programmer dans un seul bloc de code conduit à faire des copier-coller de blocs d'instructions ce qui allonge les programmes et conduit à des erreurs de recopie.

On « factorise » ces blocs de codes au sein de procédures, encore appelée routines ou fonctions.

Le code, organisée sous forme de modules, devient plus facile à concevoir, à implémenter et à maintenir : c'est l'application de la stratégie « diviser pour mieux régner ».

Cette programmation **procédurale** est utilisée en langage C.

Avantages

Le langage C a 50 ans, il est donc **très répandu** : il existe une documentation abondante à son sujet et des dépôts de codes.

Le langage est **bas niveau** ce qui permet de développer des programmes devant dialoguer avec des composants électroniques (programmation embarquée, développement de « drivers »)

L'ancienneté du langage et son caractère bas-niveau permet d'utiliser des compilateurs capables de bien **optimiser le code machine** (grande rapidité, utilisation en programmation parallèle).

Avantages

Le langage C est **normalisé** et **ouvert** ce qui permet aux éditeurs de logiciels de développer leurs propres compilateurs : le programmeur n'est pas enfermé dans une solution logicielle.

Le langage C a **influencé de nombreux autres langages** (Perl, PHP, C++, Java, C#, Objective-C, Python...) : connaître ce langage facilite l'apprentissage d'autres langages.

Inconvénients

La **gestion de la mémoire** est à la charge du programmeur (pas de « ramasse-miette » comme en Java).

Malgré son caractère « fortement typé », le langage C propose **peu de mécanismes de vérification du code** : dépassement possible de tableau... qui peuvent conduire à des plantages. Il est nécessaire d'utiliser des outils comme Lint ou Valgrind.

Pas de mécanismes tels que les espaces de nommages, les exceptions, la généricité propres aux langages plus modernes.

Popularité du langage

Selon Tiobe, le langage C est le langage le plus populaire (15,95% sur les moteurs de recherche comme Google).

Selon Redmonk, le langage C est classé 10^{ème} (requête sur le site Stackoverflow et son utilisation avec Git)

Selon PYPL, le langage C est classé 6^{ème} (recherche de tutoriel sur Google).

Voir <https://www.tiobe.com/tiobe-index/>
<https://redmonk.com/sogrady/2020/07/27/language-rankings-6-20/>
<http://pypl.github.io/PYPL.html>

**L'environnement
de développement
Code::Blocks**



Les outils de base

Il est possible de créer un programme C avec divers outils selon le système d'exploitation utilisé.

Les outils de bases sont (sous Linux) :

- **gcc** : un compilateur utilisé en ligne de commande sous des environnements ;
- **ld** : pour réaliser l'édition de liens ;
- **make** : pour automatiser le processus de compilation et d'édition de liens ;
- **gdb** : pour le débogage des programmes.

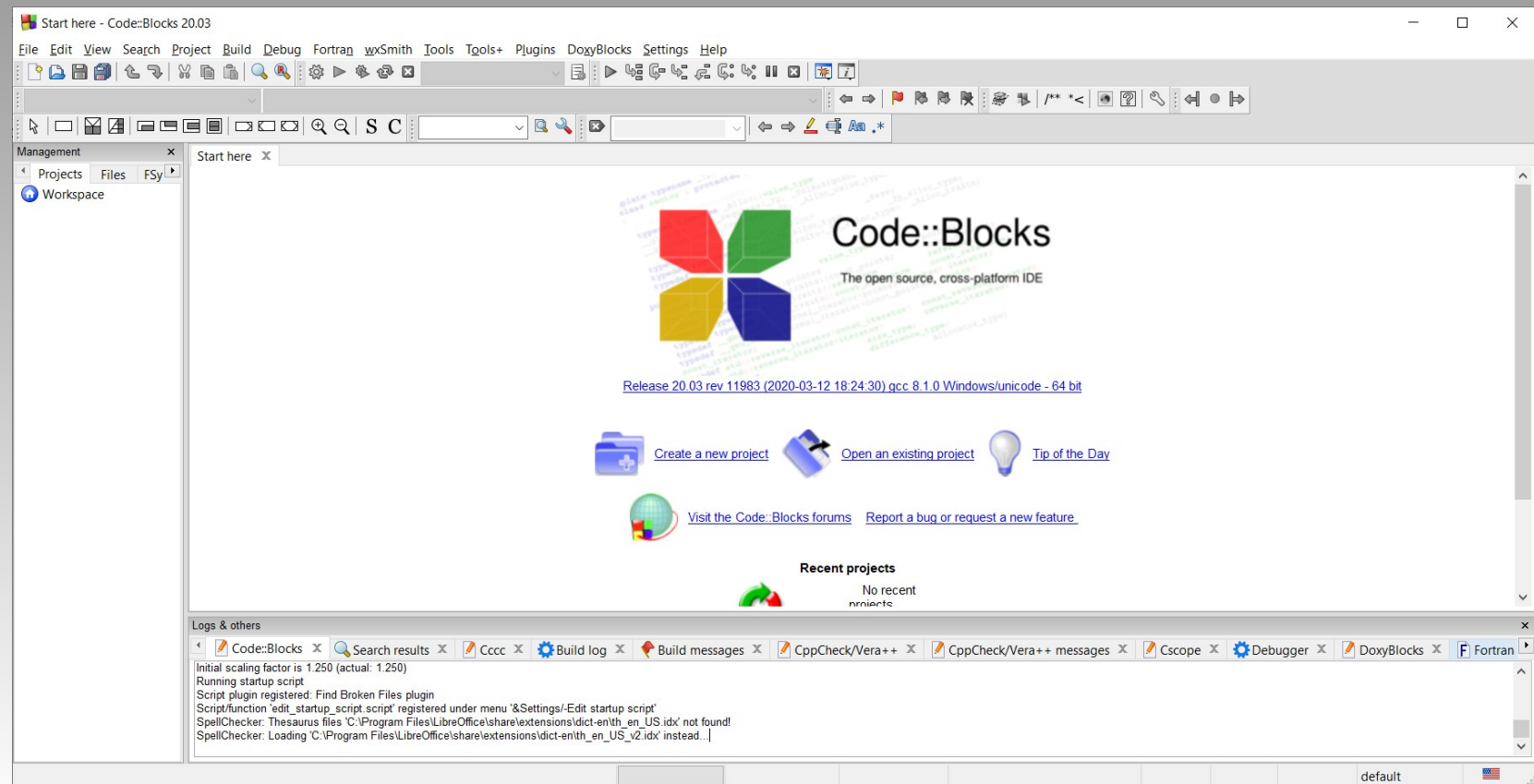
Il existe des portages de ces outils sous Windows (mingw, cmake...) et sous Mac OS X (clang...).

Ces outils de base sont utilisés par des IDE (environnements de développement intégrés) :

- Dev-C++ (<http://orwellddevcpp.blogspot.fr/>)
- Code::Blocks (<http://www.codeblocks.org/>)
- Qt Creator (<http://www.qt.io/>)
- Clion (<https://www.jetbrains.com/clion/>)
- Eclipse (<http://eclipse.org/downloads/>)
- Net Beans (<https://netbeans.org/>)
- Visual C++ (<https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx>)
- Xcode (<https://developer.apple.com/xcode/>)

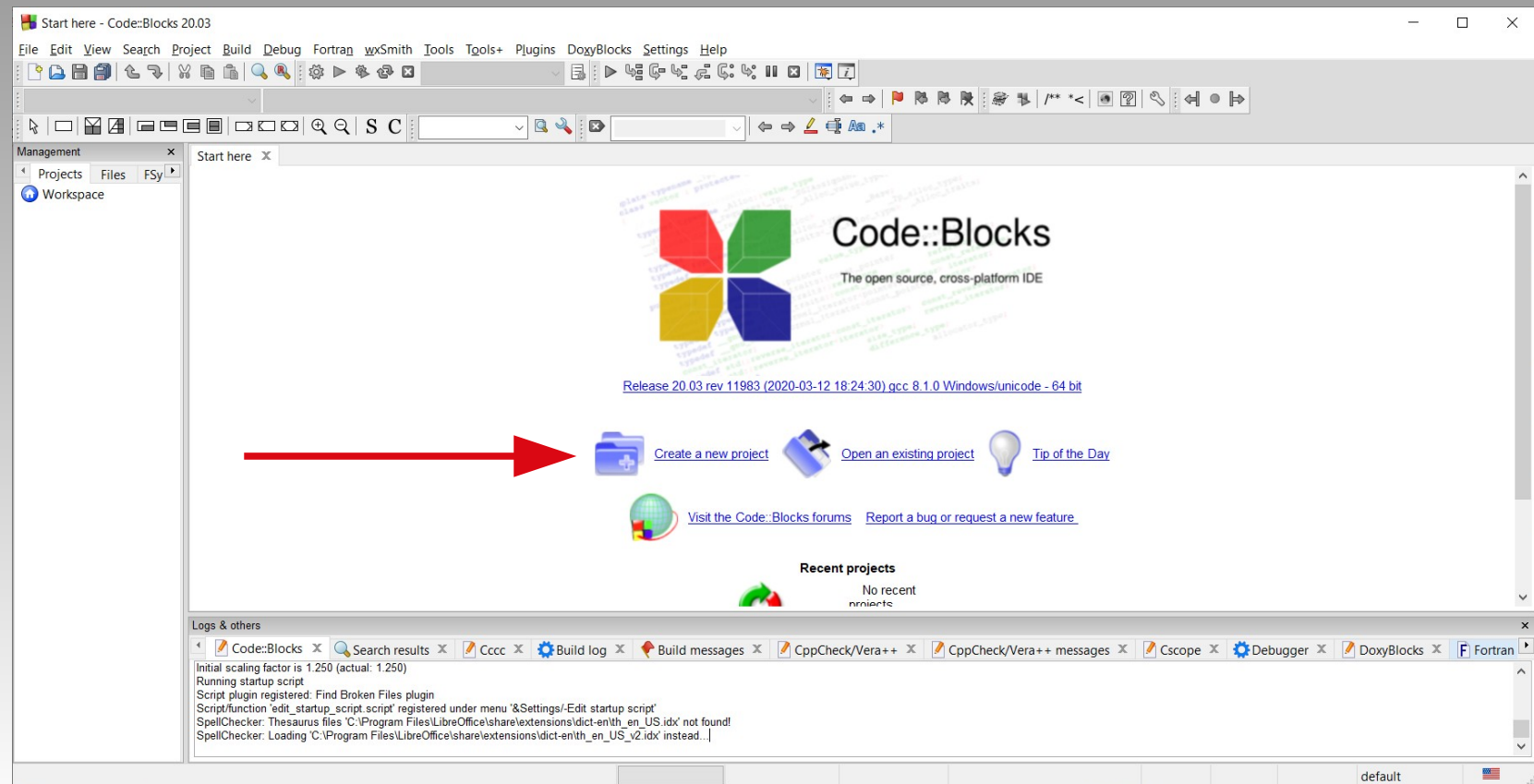
Démarrage de Code::Blocks

On démarre Code::Blocks et on obtient une fenêtre similaire à celle ci-dessous



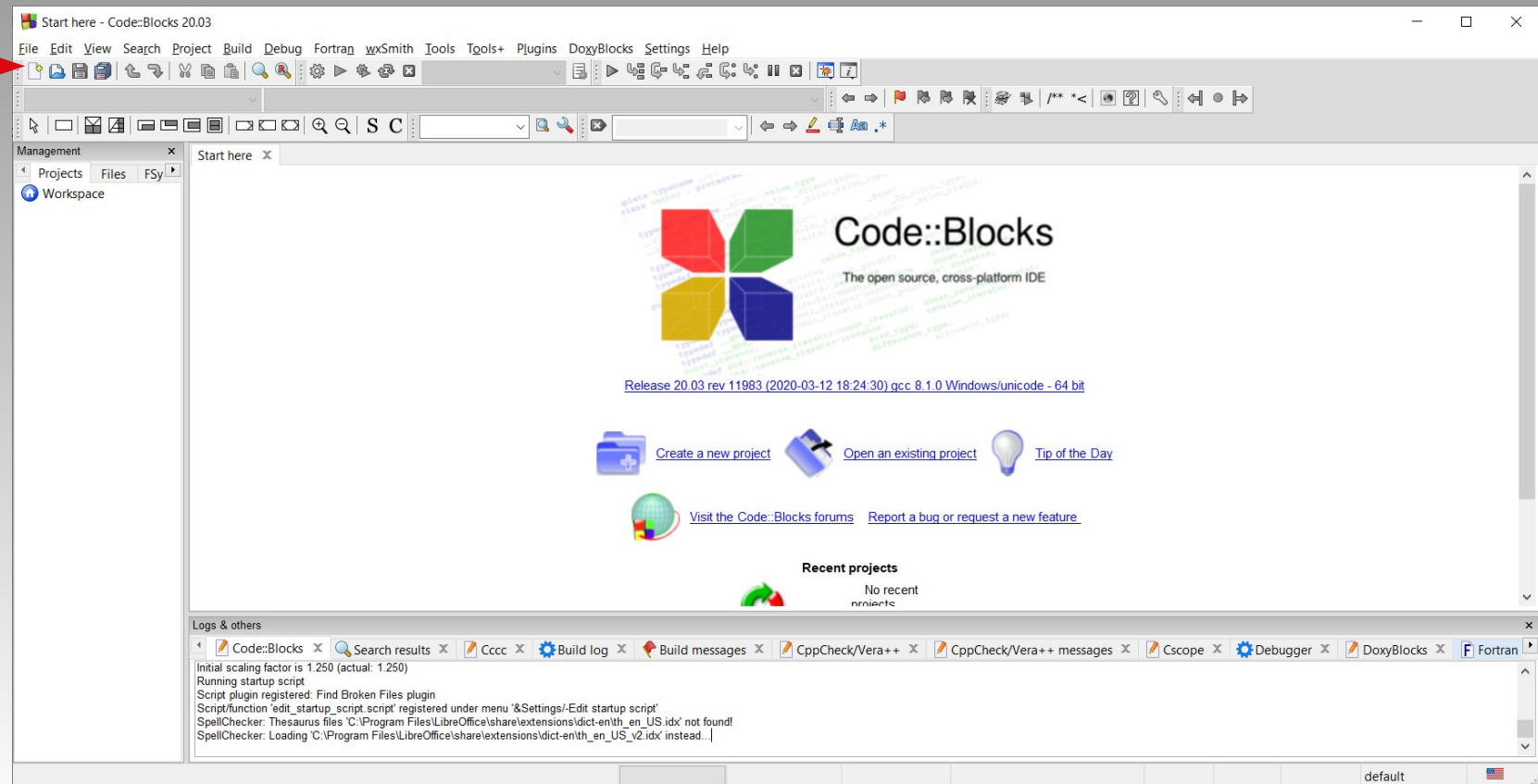
Initier la création d'un nouveau projet

On clique sur le lien « Create a new project ».



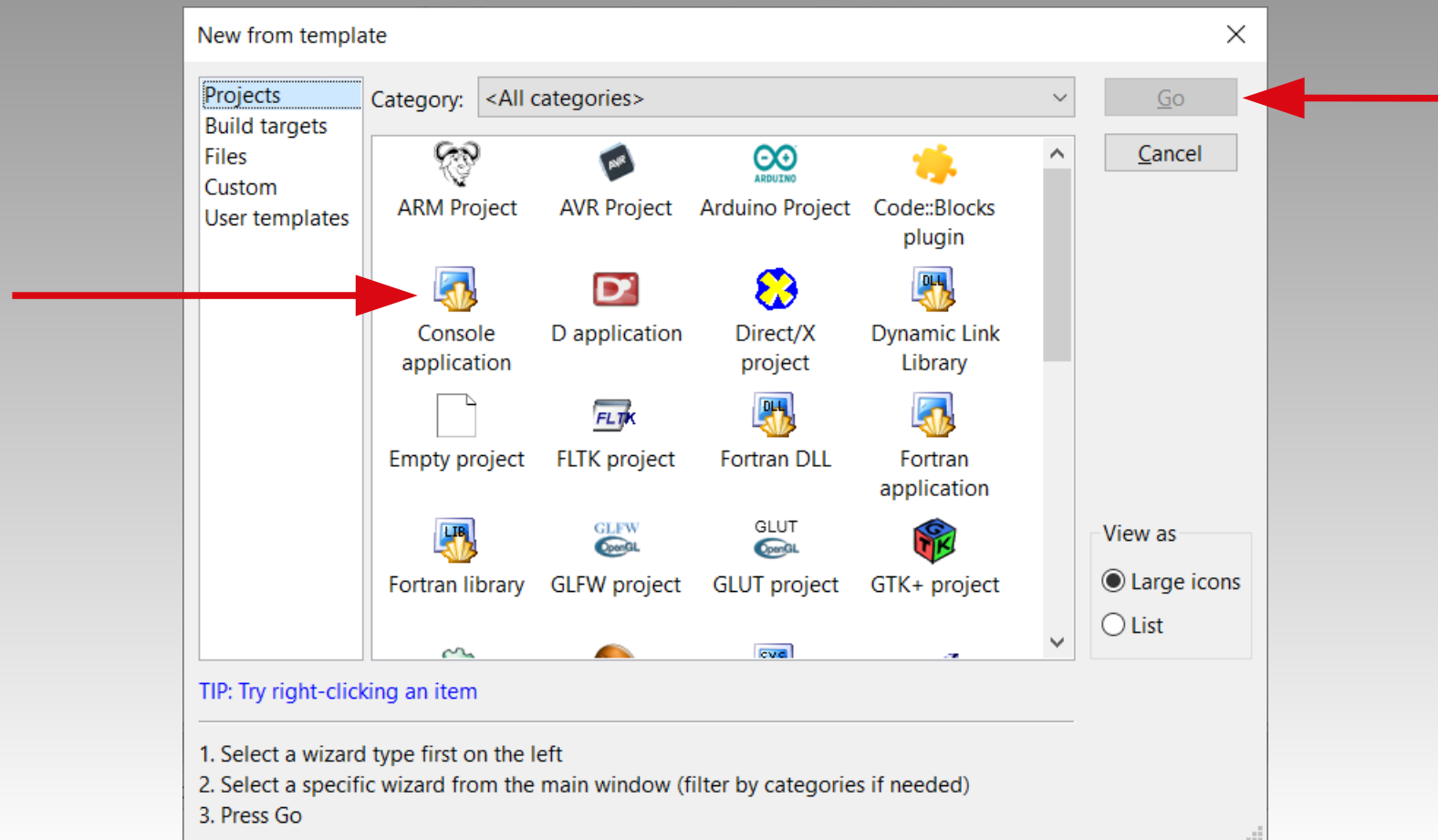
Initier la création d'un nouveau projet

On peut aussi le faire depuis le menu « File », puis « New » ou encore depuis l'icône « New » : on choisit alors « Project » dans le menu.



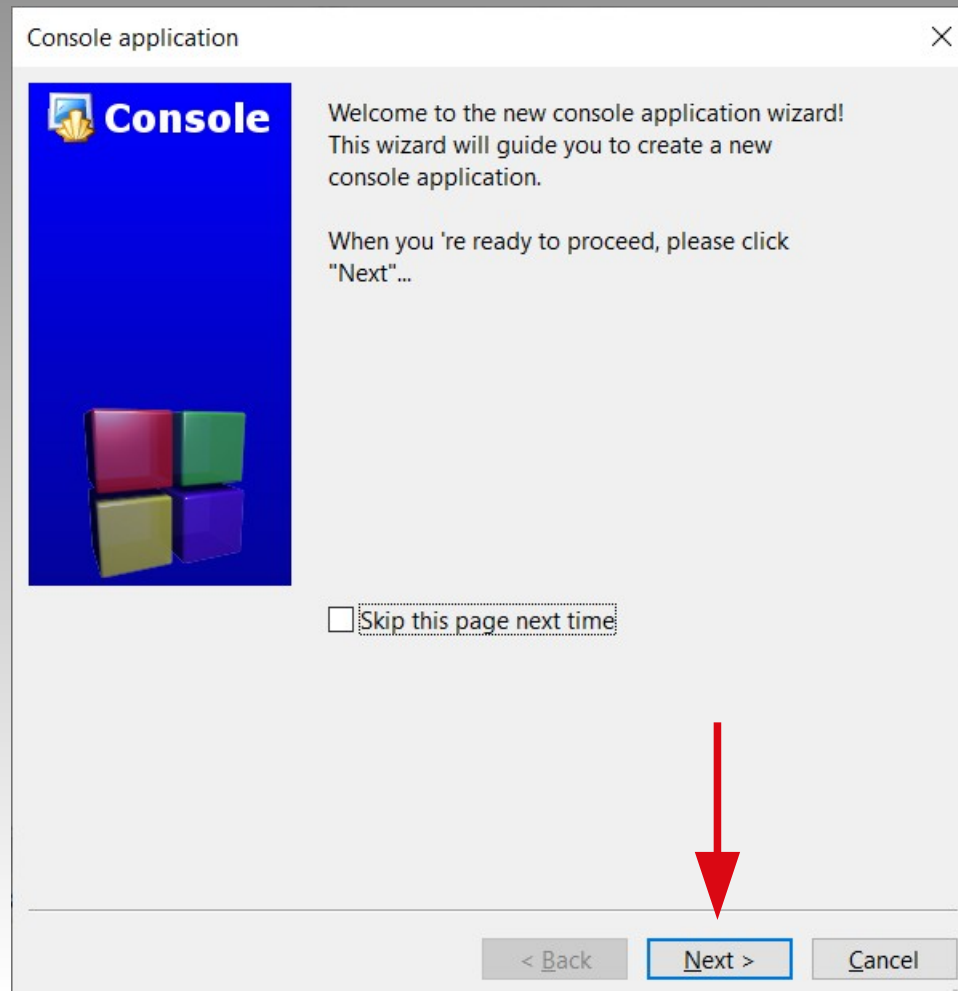
Choisir un template de projet

On clique sur l'icône « Console application » puis sur le bouton [Go].



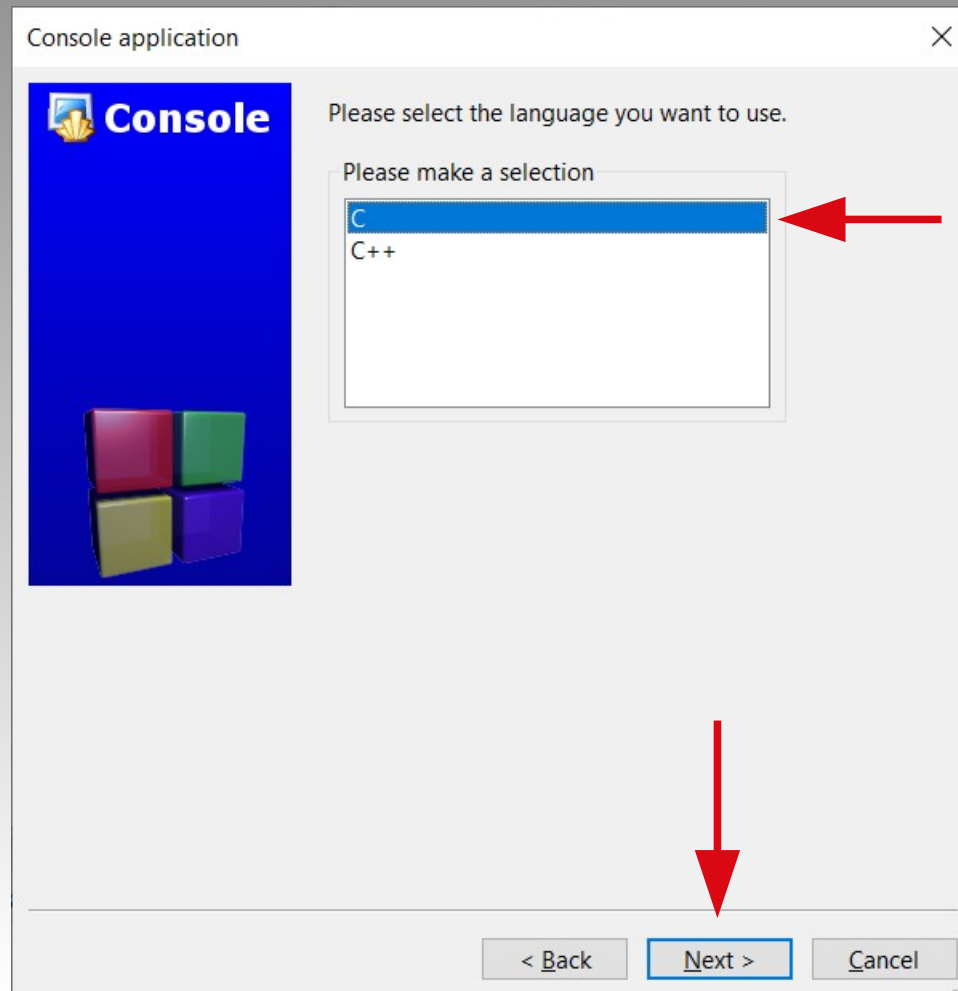
Une page de bienvenue

On clique simplement sur le bouton [Next].



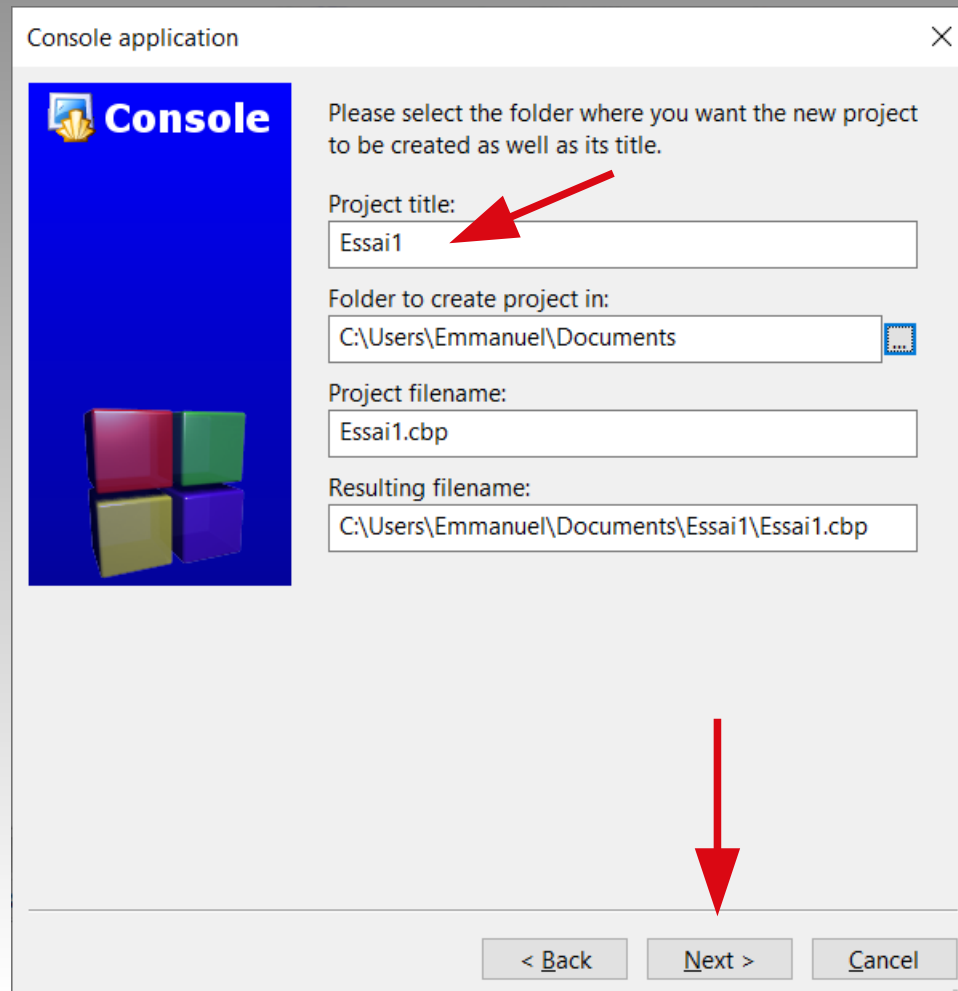
Choix du langage C

On choisit le langage C et on clique sur [Next]



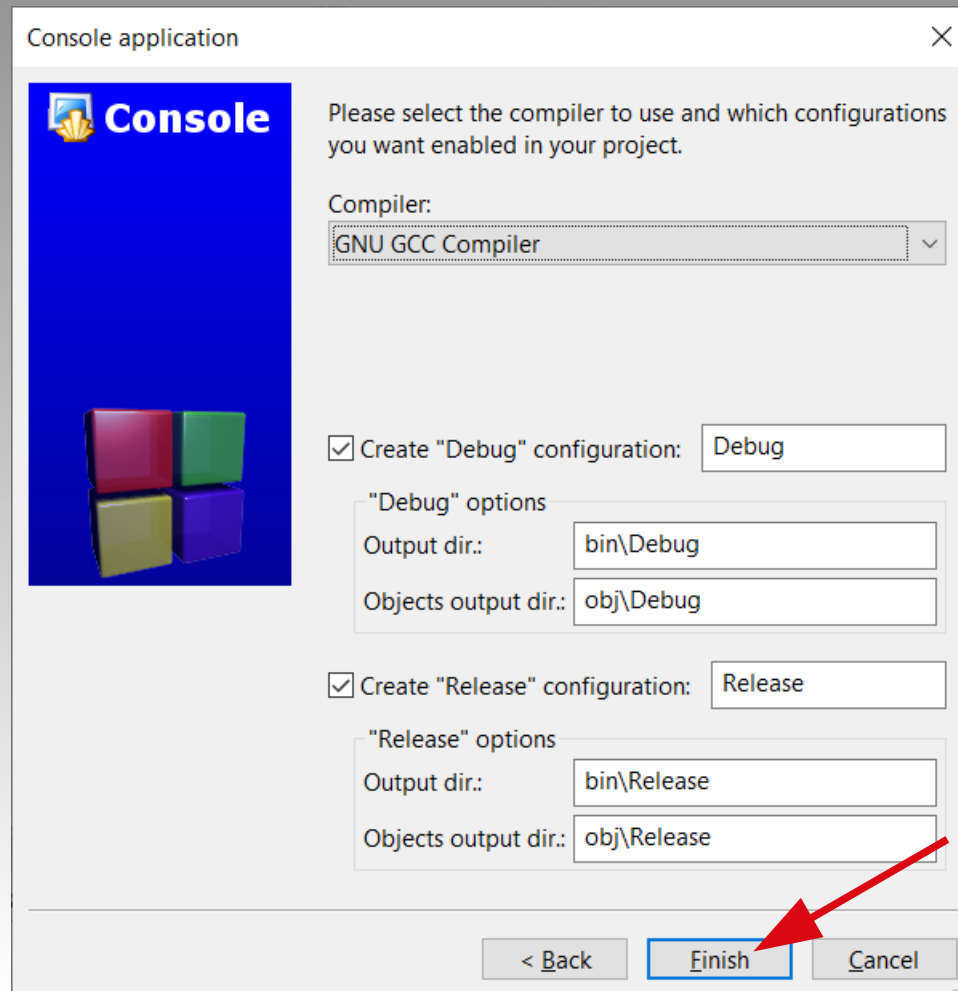
Nom du projet

On spécifie le nom du projet et son répertoire de sauvegarde puis on appuie sur [Next]



Choix du compilateur et des profils

On garde les choix par défaut puis on appuie sur [Finish]



Choix du compilateur et des profils

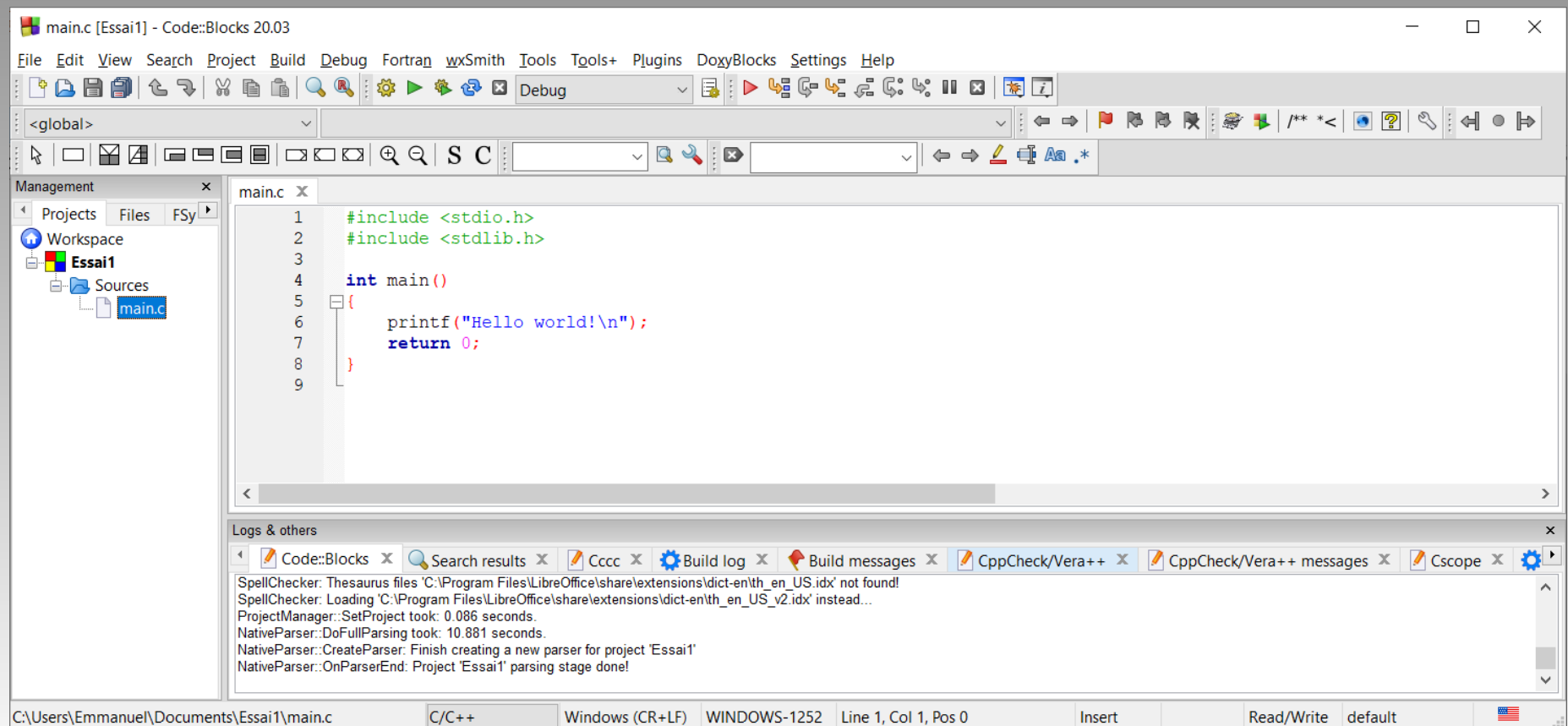
Généralement, on utilise le compilateur Mingw (<http://www.mingw.org/>) qui accompagne Code::Blocks mais il est possible d'en installer d'autres comme TDM (tdm-gcc.tdragon.net) ou le compilateur de Visual Studio.

Code::Blocks propose deux profils (avec des réglages de compilation légèrement différents) :

- un profil « debug » qui ajoute des informations dans le code machine pour le débogueur ;
- un profil « release » qui utilise de options pour optimiser le code machine généré par le compilateur.

Le programme par défaut

Une fois le projet créé, Code::Blocks propose un petit programme par défaut.



Quelques icônes

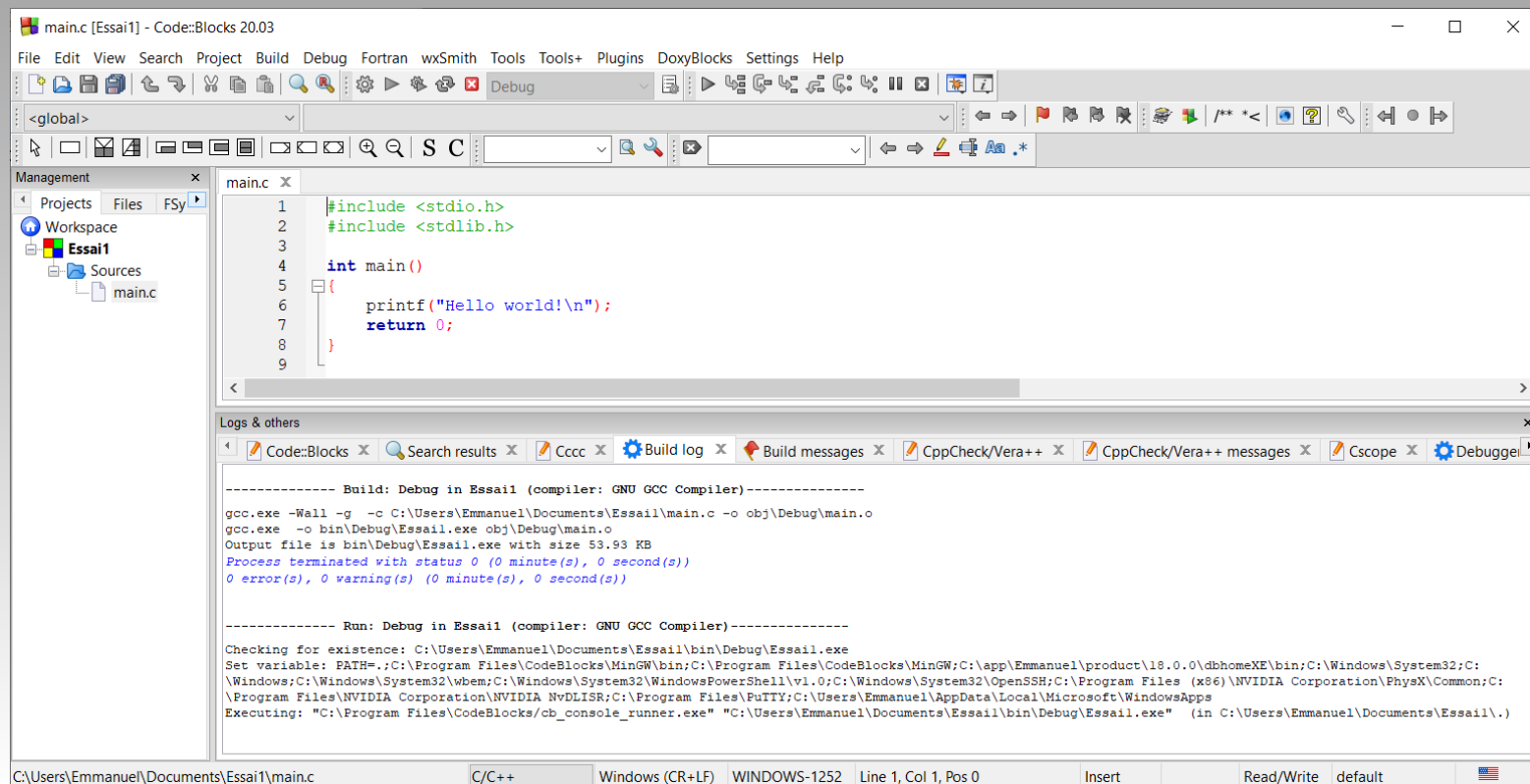
Voici la présentation de quelques widgets pour commencer à manipuler l'IDE.



Widgets	Description
1	Compiler les éléments modifiés sans exécuter le programme
2	Exécuter la dernière version du programme compilé
3	Compiler les éléments modifiés et exécuter le programme
4	Recompiler tous les éléments du programme sans l'exécuter
5	Arrêter l'exécution en cours (parfois quelques bugs qui nécessitent de redémarrer)
6	Menu déroulant permettant de choisir le profil (Debug ou Release)

Première exécution

On clique sur le widget 3 pour compiler et lancer l'exécution. On voit apparaître des messages dans une zone située sous le code source. On constate que l'IDE utilise le compilateur sous-jacent pour générer le code machine du programme.



The screenshot shows the Code::Blocks IDE interface. The main window displays a C program in `main.c`:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     printf("Hello world!\n");
7     return 0;
8 }
9
```

The "Logs & others" panel at the bottom shows the build and run output:

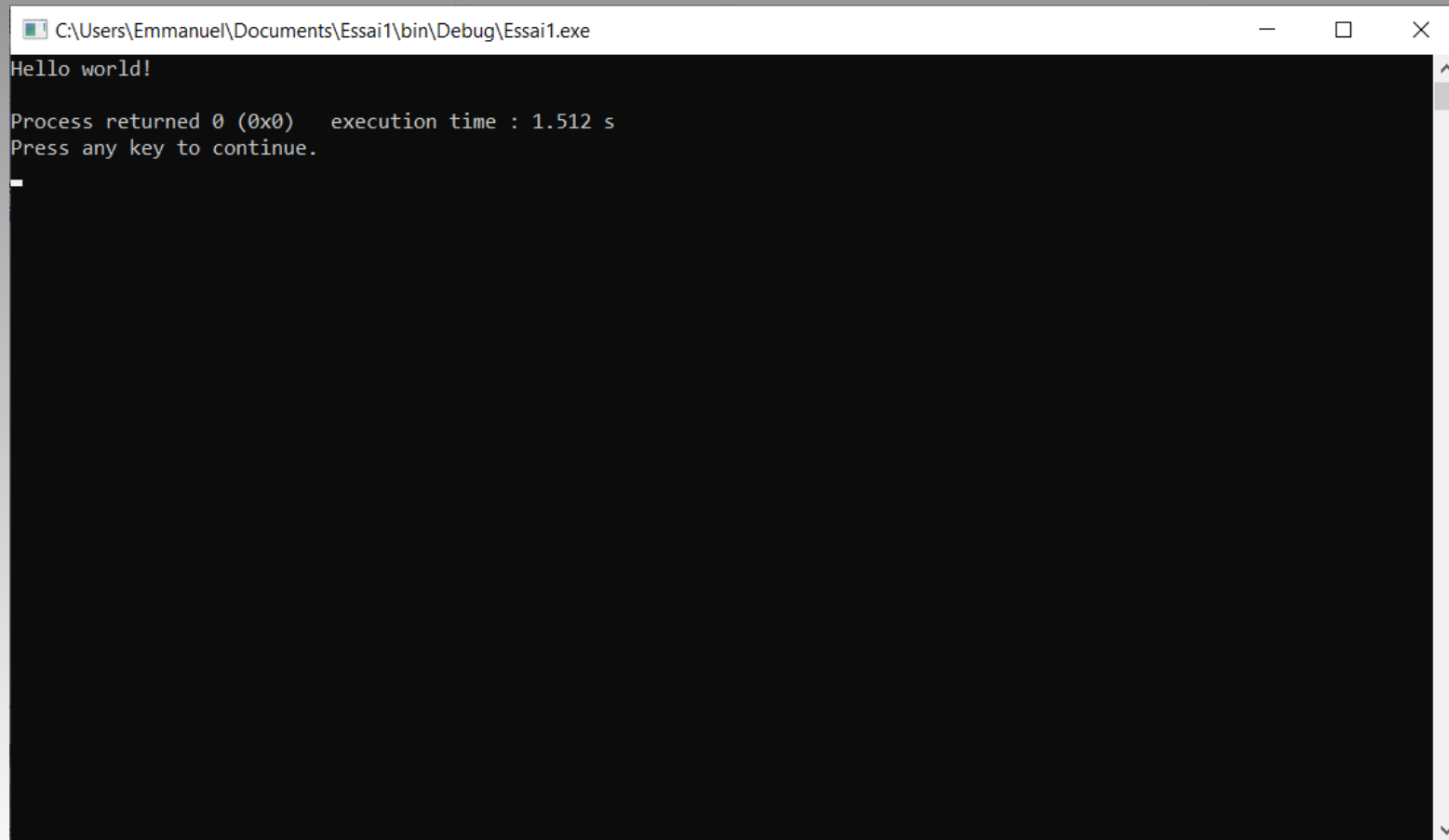
```
----- Build: Debug in Essai1 (compiler: GNU GCC Compiler)-----
gcc.exe -Wall -g -c C:\Users\Emmanuel\Documents\Essai1\main.c -o obj\Debug\main.o
gcc.exe -o bin\Debug\Essai1.exe obj\Debug\main.o
Output file is bin\Debug\Essai1.exe with size 53.93 KB
Process terminated with status 0 (0 minute(s), 0 second(s))
0 error(s), 0 warning(s) (0 minute(s), 0 second(s))

----- Run: Debug in Essai1 (compiler: GNU GCC Compiler)-----
Checking for existence: C:\Users\Emmanuel\Documents\Essai1\bin\Debug\Essai1.exe
Set variable: PATH=.;C:\Program Files\CodeBlocks\MinGW\bin;C:\Program Files\CodeBlocks\MinGW\C:\app\Emmanuel\product\18.0.0\dbhomeXE\bin;C:\Windows\System32;C:\Windows\System32\wbem;C:\Windows\System32\WindowsPowerShell\v1.0;C:\Windows\System32\OpenSSH;C:\Program Files (x86)\NVIDIA Corporation\PhysX\Common;C:\Program Files\NVIDIA Corporation\NVIDIA NvDLISR;C:\Program Files\PuTTY;C:\Users\Emmanuel\AppData\Local\Microsoft\WindowsApps
Executing: "C:\Program Files\CodeBlocks\cb_console_runner.exe" "C:\Users\Emmanuel\Documents\Essai1\bin\Debug\Essai1.exe" (in C:\Users\Emmanuel\Documents\Essai1.)
```

The status bar at the bottom indicates the file path `C:\Users\Emmanuel\Documents\Essai1\main.c`, the language `C/C++`, and the window state `Windows (CR+LF)`.

Première exécution

Une nouvelle fenêtre apparaît : il s'agit de la console dans laquelle s'exécute le programme qui affiche « Hello World ! ».



The screenshot shows a Windows console window titled "C:\Users\Emmanuel\Documents\Essai1\bin\Debug\Essai1.exe". The console output is as follows:

```
Hello world!  
  
Process returned 0 (0x0)   execution time : 1.512 s  
Press any key to continue.  
_
```

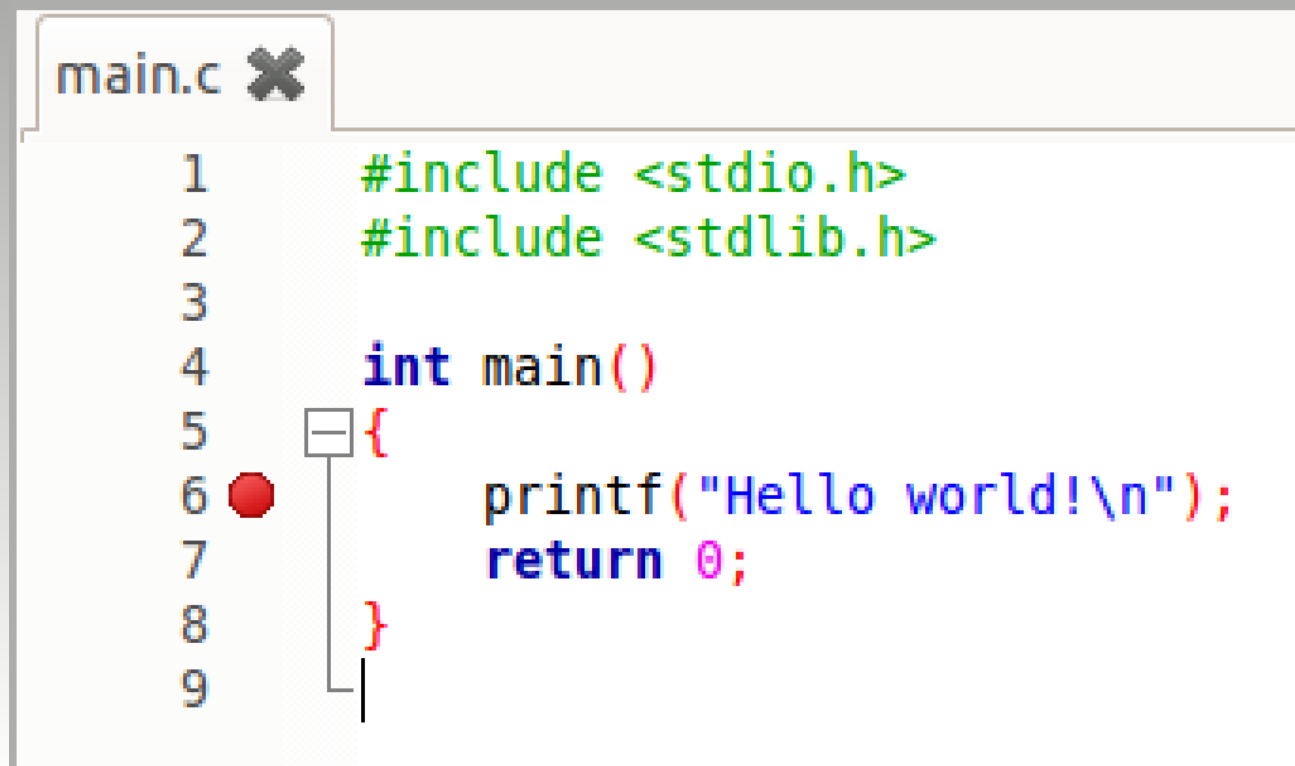
Le profil « debug »

Le profil « Debug » utilise une option de compilation (-g) qui demande au compilateur d'ajouter des informations qui sont utilisées par le débogueur « gdb ».



Les points d'arrêt

On peut forcer le programme à s'interrompre à une instruction donnée. Pour cela, il suffit de cliquer sur le numéro de ligne se trouvant en face de l'instruction : une boule rouge, matérialisant ce *breakpoint* apparaît



```
main.c ✕  
1      #include <stdio.h>  
2      #include <stdlib.h>  
3  
4      int main()  
5      {  
6  ●    printf("Hello world!\n");  
7      return 0;  
8      }  
9
```

The image shows a code editor window titled 'main.c ✕'. The code is a simple C program. Line 6 has a red dot (breakpoint) next to the 'printf' statement. A vertical line with a small square at the top and a horizontal line at the bottom is positioned to the left of the code, indicating the current execution point.

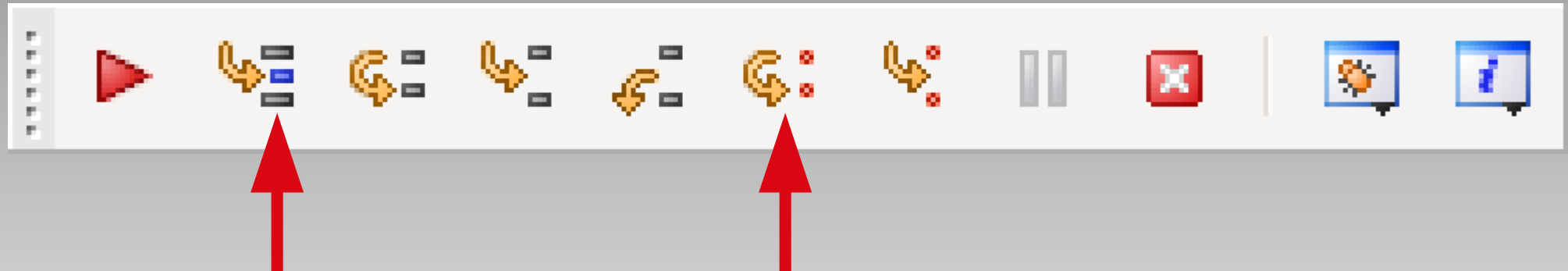
Le lancement du mode « débogage »

On lance l'exécution du programme en mode « débogage » en cliquant sur le triangle rouge. On constate que le programme s'arrête au point d'arrêt.



Avancer pas-à-pas

Certains boutons grisés deviennent accessibles. On peut alors utiliser le bouton qui permet d'avancer d'une ligne de code ou celui qui permet d'avancer d'une instruction.



Les autres boutons seront présentés plus tard, au fur et à mesure de l'avancé dans le cours

Variables



A l'instar de l'algorithmique, on retrouve la notion de variables qui permet de stocker temporairement des informations lors du fonctionnement du programme.

Ces variables correspondent à des cases mémoires qui ont :

- un nom ;
- un type ;
- une valeur ;
- une adresse (nous aborderons plus tard cette notion).

Quelques règles

Le nom des variables doit respecter quelques règles :

- Le nom des variables ne peut comporter que :
 - des lettres non accentuées (pas de « é », « ç »...) ;
 - des chiffres ;
 - le symbole « _ ».
- Le nom des variables ne peut pas commencer par un chiffre.
- On ne peut pas utiliser les mots-clés du langage C (le compilateur ne pourrait pas faire la différence entre le nom d'une variable et l'instruction éponyme).

Mots-clés du langage C

La liste des mots-clés varie selon la norme utilisée.

C90	auto	break	case	char	const
	continue	default	do	double	else
	enum	extern	float	for	goto
	if	int	long	register	return
	short	signed	sizeof	static	struct
	switch	typedef	union	unsigned	void
	volatile	while			
C99	inline	restrict	_Bool	_Complex	_Imaginary
C11	_Alignas	_Alignof	_Atomic	_Generic	_Noreturn
	_Static_assert	_Thread_local			

Voir http://fr.wikibooks.org/wiki/Programmation_C/Bases_du_langage#Mots_r.C3.A9serv.C3.A9s_du_langage

Quelques conventions

Il est aussi conseillé de respecter quelques conventions :

- Le nom des variables est écrit en minuscules.
- Si le nom d'une variable est composé de plusieurs termes, on peut les séparer en utilisant le « _ » ou en écrivant la première lettre de chaque terme avec une majuscule (à l'exception de la première lettre) :
 - `montant_des_depenses`
 - `montantDesDepenses`

Le langage C est **typé** :

- Le typage permet de définir la **plage de valeurs** pouvant être stockée dans une variable ce qui permet au compilateur de **contrôler l'exactitude des affectations** ;
- Le typage permet aussi déterminer la **taille d'une variable en mémoire** ce qui est indispensable pour la création des variables, l'arithmétique des pointeurs...

5 types

Le langage C propose 5 types de base pour stocker des nombres entiers :

- **char** ;
- `short int` ou **short** ;
- **int** ;
- `long int` ou **long** ;
- `long long int` ou **long long** (C99).

Premier tableau de synthèse

Le tableau ci-dessous énumère les 5 types permettant de stocker des entiers signés avec la plage de valeurs associées.

Type	Taille	Valeur minimale		Valeur maximale	
char	1 o.	-2^7	-128	$2^7 - 1$	127
short	2 o.	-2^{15}	-32768	$2^{15} - 1$	32767
int	2 o.	-2^{15}	-32768	$2^{15} - 1$	32767
	4 o.	-2^{31}	-2147483648	$2^{31} - 1$	2147483647
long	4 o.	-2^{31}	-2147483648	$2^{31} - 1$	2147483647
long long	8 o.	-2^{63}	-9223372036854775808	$2^{63} - 1$	9223372036854775807

On remarque le cas particulier du type `int` dont la taille varie selon les implémentations des compilateurs et selon l'architecture cible (processeurs 16 bits ou 32 bits).



5 types

Ces types permettent donc de manipuler des nombres signés qui peuvent être négatifs.

Si on souhaite utiliser des nombres non-signés, qui ne peuvent prendre que des valeurs positives ou nulles, il faut préfixer ces types par le mot-clé **unsigned**.

On a alors :

- `unsigned char ;`
- `unsigned short int` ou `unsigned short ;`
- `unsigned int ;`
- `unsigned long int` ou `unsigned long ;`
- `unsigned long long int` ou `unsigned long long .`

Second tableau de synthèse

Le tableau ci-dessous énumère les 5 types permettant de stocker les entiers non signés avec la plage de valeurs associées.

Type	Taille	Valeur Minimale	Valeur maximale	
unsigned char	1 o.	0	$2^8 - 1$	255
unsigned short	2 o.	0	$2^{16} - 1$	65535
unsigned int	2 o.	0	$2^{16} - 1$	65535
	4 o.	0	$2^{32} - 1$	4294967295
unsigned long	4 o.	0	$2^{32} - 1$	4294967295
unsigned long long	8 o.	0	$2^{64} - 1$	18446744073709551615

Objectifs

On va s'intéresser à un autre programme qui va permettre de retrouver les informations concernant les différents types utilisés pour manipuler des nombres entiers.

Cela permettra également de présenter :

- l'opérateur `sizeof` ;
- les macros du fichier d'en-tête `types.h`.

Le programme

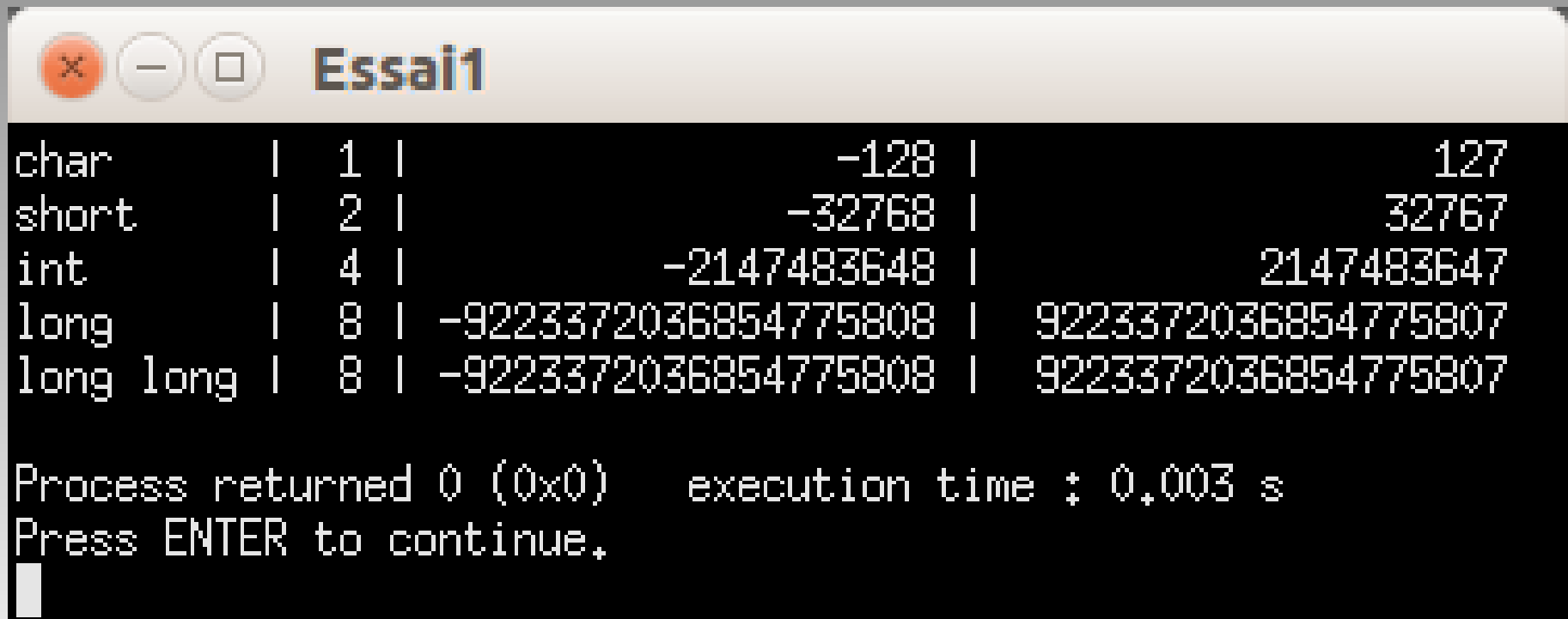
On considère le programme ci-dessous :

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

int main()
{
    printf ("char      | %2ld | %20d | %20d\n",
           sizeof(char),      CHAR_MIN,    CHAR_MAX);
    printf ("short     | %2ld | %20d | %20d\n",
           sizeof(short),     SHRT_MIN,   SHRT_MAX);
    printf ("int        | %2ld | %20d | %20d\n",
           sizeof(int),       INT_MIN,    INT_MAX);
    printf ("long        | %2ld | %20ld | %20ld\n",
           sizeof(long),      LONG_MIN,   LONG_MAX);
    printf ("long long   | %2ld | %20lld | %20lld\n",
           sizeof(long long), LLONG_MIN,  LLONG_MAX);
    return 0;
}
```

Exécution

Lorsqu'on l'exécute, on obtient un tableau de description des 5 types permettant de stocker des entiers.



```
Essai1
char      | 1 |          -128 |          127
short     | 2 |         -32768 |          32767
int       | 4 |    -2147483648 |    2147483647
long      | 8 | -9223372036854775808 | 9223372036854775807
long long | 8 | -9223372036854775808 | 9223372036854775807

Process returned 0 (0x0)    execution time : 0.003 s
Press ENTER to continue.
█
```

Explications

L'opérateur **sizeof**, fourni par le langage C permet de connaître la **taille d'un type ou d'une variable** (le nombre d'octets nécessaires pour stocker l'information en mémoire).

CHAR_MIN, CHAR_MAX... sont des macros définis dans le fichier d'en-tête `limits.h` qui permettent de connaître la valeur minimale et la valeur maximale pour un type signé.

Pour pouvoir les utiliser, il est nécessaire d'écrire l'instruction `#include <limits.h>`.

Explications

printf est une fonction, définie dans le fichier d'en-tête **stdio.h** (il faut donc écrire `#include <stdio.h>` afin de pouvoir l'utiliser).

`printf` affiche les informations écrites entre guillemets.

```
printf ("char          | %2ld | %20d | %20d\n",      sizeof(char),      CHAR_MIN,  CHAR_MAX);
printf ("short         | %2ld | %20d | %20d\n",      sizeof(short),     SHRT_MIN,  SHRT_MAX);
printf ("int           | %2ld | %20d | %20d\n",      sizeof(int),       INT_MIN,   INT_MAX);
printf ("long          | %2ld | %20ld | %20ld\n",      sizeof(long),      LONG_MIN,  LONG_MAX);
printf ("long long     | %2ld | %20lld | %20lld\n",    sizeof(long long), LLONG_MIN, LLONG_MAX);
```

Le « **\n** » est remplacé par un **passage à la ligne**.

Explications

Les blocs « %2d », « %20ld »... sont remplacés par des valeurs numériques générées par sizeof ou les macros.

```
printf ("char          | %2ld | %20d | %20d\n",      sizeof(char),      CHAR_MIN, CHAR_MAX);
printf ("short         | %2ld | %20d | %20d\n",    sizeof(short),     SHRT_MIN, SHRT_MAX);
printf ("int            | %2ld | %20d | %20d\n",    sizeof(int),       INT_MIN,  INT_MAX);
printf ("long           | %2ld | %20ld | %20ld\n",    sizeof(long),      LONG_MIN, LONG_MAX);
printf ("long long      | %2ld | %20lld | %20lld\n",  sizeof(long long), LLONG_MIN, LLONG_MAX);
```

char		1		-128		127
short		2		-32768		32767
int		4		-2147483648		2147483647
long		4		-2147483648		2147483647
long long		8		-9223372036854775808		9223372036854775807



Explications

La valeur numérique écrite après le % correspond au nombre de caractères qu'on réserve pour écrire cette information.

```
printf ("char          | %2ld | %20d | %20d\n",      sizeof(char),      CHAR_MIN,  CHAR_MAX);
printf ("short         | %2ld | %20d | %20d\n",      sizeof(short),     SHRT_MIN,  SHRT_MAX);
printf ("int            | %2ld | %20d | %20d\n",      sizeof(int),       INT_MIN,   INT_MAX);
printf ("long           | %2ld | %20ld | %20ld\n",      sizeof(long),      LONG_MIN,  LONG_MAX);
printf ("long long      | %2ld | %20lld | %20lld\n",    sizeof(long long), LLONG_MIN, LLONG_MAX);
```

		2		20		20
	
char		1		-128		127
short		2		-32768		32767
int		4		-2147483648		2147483647
long		4		-2147483648		2147483647
long long		8		-9223372036854775808		9223372036854775807



Explications

La lettre « **d** » signifie qu'on affiche un nombre signé
On ajoute « **l** » lorsqu'il s'agit d'un entier de type « long ».
On ajoute « **ll** » pour un entier de type « long long »

```
printf ("char          | %2ld | %20d | %20d\n",      sizeof(char),      CHAR_MIN,  CHAR_MAX);
printf ("short         | %2ld | %20d | %20d\n",      sizeof(short),     SHRT_MIN,  SHRT_MAX);
printf ("int            | %2ld | %20d | %20d\n",      sizeof(int),       INT_MIN,   INT_MAX);
printf ("long           | %2ld | %20ld | %20ld\n",      sizeof(long),      LONG_MIN,  LONG_MAX);
printf ("long long      | %2ld | %20lld | %20lld\n",    sizeof(long long), LLONG_MIN, LLONG_MAX);
```

		2		20		20
	
char		1		-128		127
short		2		-32768		32767
int		4		-2147483648		2147483647
long		4		-2147483648		2147483647
long long		8		-9223372036854775808		9223372036854775807



Exercice

On considère les macros :

- UCHAR_MAX (valeur max. d'un unsigned char),
- USHRT_MAX (valeur max. d'un unsigned short),
- UINT_MAX (valeur max. d'un unsigned int),
- ULONG_MAX (valeur max. d'un unsigned long)
- ULLONG_MAX (valeur max. d'un unsigned long long)

Modifiez le programme précédent de manière à afficher des informations concernant les 5 types utilisés pour stocker des entiers non signés.

A cet effet, on devra utiliser les formats suivants :

- `%u` à la place de `%d`
- `%lu` à la place de `%ld`
- `%llu` à la place de `%lld`

On peut trouver des informations complémentaires (notamment les formats reconnus par la fonction `printf`), en tapant « `man printf` » dans un moteur de recherche.

Cela donne accès au manuel du langage, dont on dispose aussi dans la console sous Linux (avec la même commande).

Des nouveaux types

Si on inclut le fichier `stdint.h`, on peut utiliser des nouveaux types pour manipuler les entiers. Ces types présentent l'avantage d'avoir une taille qui ne dépend pas de l'architecture sous-jacente.

Nom du type signé	Nom du type non signé	Nombre de bits
<code>int8_t</code>	<code>uint8_t</code>	8
<code>int16_t</code>	<code>uint16_t</code>	16
<code>int32_t</code>	<code>uint32_t</code>	32
<code>int64_t</code>	<code>uint64_t</code>	64

La situation vers 1961

De nombreux types d'ordinateurs existent et chacun utilise un codage des caractères qui lui est propre.

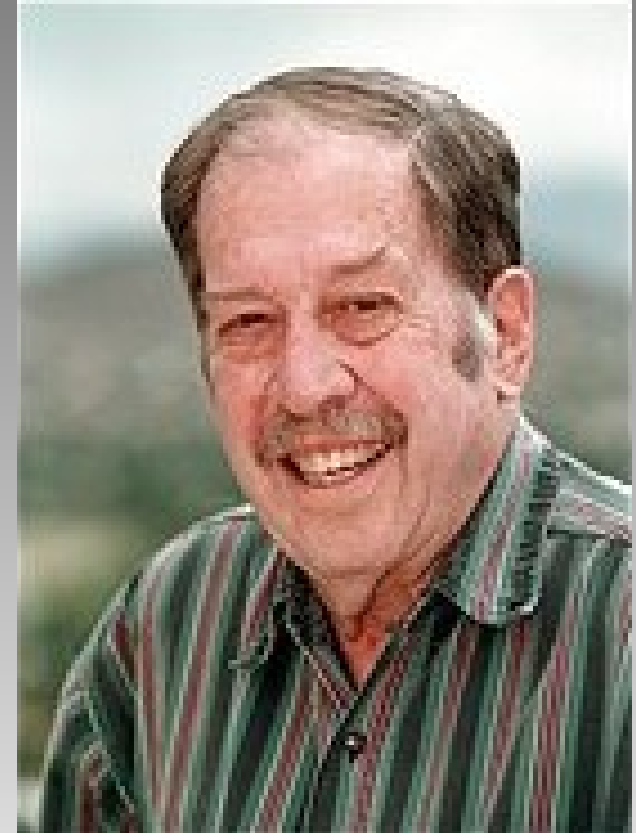
Au sein même d'une société comme IBM ou DEC différents codes existent (IBM utilise 5 codes différents, DEC utilise d'autres codes comme le RADIX-50 et le SIBIT).

Il y a plus de 60 façons de représenter les caractères dans un ordinateur ce qui rend difficile la transmission d'informations entre les ordinateurs.

La gêne engendrée par cette situation

Cette situation gêne Robert BEMER, le responsable des standards de programmation chez IBM. Il y fait allusion lors d'une interview en 1999 :

« Ils commençaient à parler de familles d'ordinateurs qui auraient besoin de communiquer. Je dis alors « Hey, vous ne pouvez même pas vous parler entre vous alors laissez le monde extérieur tranquille » »



La conception de la table ASCII 7 bits

Il s'attelle à alors à la conception d'un nouveau code baptisé **ASCII** (American Standard Code for Interchange Information) et il le propose en mai 1961 à l'ANSI (American National Standards Institute) afin de mettre fin à cette situation.

Comme le montre la plaque de sa voiture, BEMER se présente comme le « père de l'ASCII »



La normalisation de la table ASCII 7 bits

BEMER prend naturellement alors part au comité qui va établir la norme X 3.4 en 1968.

Cette norme fut ensuite proposée aux organismes internationaux et l'ASCII devient alors :

- la norme ISO 636 à l'ISO (International Standard Organisation) en 1973 ;
- la norme V.2 au CCITT (Comité Consultatif International Télégraphique et Téléphonique).

La table ASCII 7 bits

0	NUL	<null> ou ESP	32	ESP	64	@	96	`
1	SOH	<start of heading> ou ☺	33	!	65	A	97	a
2	STX	<start of text> ou ☼	34	"	66	B	98	b
3	ETX	<end of text> ou ♥	35	#	67	C	99	c
4	EOT	<end of transmission> ou ♦	36	\$	68	D	100	d
5	ENQ	<enquiry> ou ♣	37	%	69	E	101	e
6	ACK	<acknowledge> ou ♠	38	&	70	F	102	f
7	BEL	<bell>	39	'	71	G	103	g
8	BS	<backspace>	40	<	72	H	104	h
9	TAB	<horizontal tab>	41	>	73	I	105	i
10	LF	<NL line feed, new line>	42	*	74	J	106	j
11	VT	<vertical tab> ou ♂	43	+	75	K	107	k
12	FF	<NP form feed, new page> ou ♀	44	,	76	L	108	l
13	CR	<carriage return>	45	-	77	M	109	m
14	SO	<shift out> ou ♂	46	.	78	N	110	n
15	SI	<shift int> ou *	47	/	79	O	111	o
16	DLE	<data link escape> ou ►	48	0	80	P	112	p
17	DC1	<device controle 1> ou ◄	49	1	81	Q	113	q
18	DC2	<device controle 2> ou ‡	50	2	82	R	114	r
19	DC3	<device controle 3> ou !!	51	3	83	S	115	s
20	DC4	<device controle 4> ou ¶	52	4	84	T	116	t
21	NAK	<negative acknowledge> ou ⚡	53	5	85	U	117	u
22	SYN	<synchronous idle> ou =	54	6	86	V	118	v
23	ETB	<end of transmission block> ou ‡	55	7	87	W	119	w
24	CAN	<cancel> ou ↑	56	8	88	X	120	x
25	EM	<end of medium> ou ↓	57	9	89	Y	121	y
26	SUB	<substitute> ou →	58	:	90	Z	122	z
27	ESC	<escape> ou ←	59	;	91	[123	<
28	FS	<file separator> ou ⊥	60	<	92	\	124	!
29	GS	<group separator> ou ⇄	61	=	93]	125	>
30	RS	<record separator> ou ▲	62	>	94	^	126	~
31	US	<unit separator> ou ▼	63	?	95	_	127	Δ

La table ASCII n'utilise que 7 bits. Comme l'électronique n'est pas encore très fiable, le 8^{ème} bit est utilisé comme **bit de parité** pour détecter des erreurs dans les traitements des octets :

- l'émetteur met ce bit à 0 s'il y a un nombre pair de 1, et à 1 dans le cas contraire ;
- le récepteur compte le nombre de 1 afin de connaître la parité puis il lit le 8^{ème} bit : si le résultat diffère, il y a un problème (ce système peut être facilement trompé).

Exemple : **1**0010110

L'arrivée des pages de code

Plus tard, lorsque d'autres méthodes de détection et de correction des erreurs ont été mises en place, le 8^{ème} bit a été utilisé pour **coder de nouveaux symboles**.

Le DOS proposait alors de charger **des pages de codes** supplémentaires adaptés au pays (par exemple, il existait une page de code « latin-1 » qui proposait les caractères accentués pour les Français).

La page de code « latin-1 »

128	Ç	160	á	192	Ł	224	ó
129	ü	161	í	193	±	225	ø
130	é	162	ó	194	⌈	226	ô
131	â	163	ú	195	⌋	227	õ
132	ä	164	ñ	196	—	228	ö
133	à	165	Ñ	197	+	229	õ
134	å	166	•	198	±	230	μ
135	ç	167	•	199	±	231	þ
136	ê	168	ç	200	±	232	ó
137	ë	169	ø	201	⌈	233	ú
138	è	170	¬	202	⌈	234	û
139	ï	171	½	203	⌈	235	ü
140	î	172	¾	204	⌈	236	ý
141	ì	173	•	205	=	237	ÿ
142	ä	174	«	206	±	238	—
143	å	175	»	207	±	239	—
144	É	176	■	208	±	240	—
145	æ	177	■	209	±	241	±
146	Æ	178	■	210	±	242	=
147	ô	179	—	211	±	243	¾
148	ö	180	—	212	±	244	¶
149	ò	181	±	213	±	245	§
150	û	182	±	214	±	246	÷
151	ù	183	±	215	±	247	—
152	ÿ	184	ø	216	±	248	ö
153	ö	185	⌈	217	±	249	—
154	ü	186	⌈	218	±	250	—
155	ø	187	⌈	219	■	251	1
156	£	188	⌈	220	■	252	3
157	Ø	189	ç	221	■	253	2
158	×	190	¥	222	■	254	■
159	f	191	¬	223	■	255	ESP

Le problème des pages de code

La multitude des langues et des symboles a entraîné la création de nombreuses pages de code, mais ce système, pensé en premier lieu pour les américains, n'était pas adapté à un contexte d'utilisation international :

- la communication (par mail) entre personnes utilisant des alphabets différents pouvait être complexe (certains caractères étaient mal interprétés) ;
- le déploiement des logiciels au niveau international était plus coûteux.

L'Universal Character Set

En **1984**, un groupe de travail commun à l'ISO et l'IEC (International Electrotechnical Commission) a été formé en vue « d'élaborer une norme établissant un répertoire de caractères graphiques des langues écrites du monde et son codage ».

Le résultat de ce travail est la norme **ISO/IEC 10646** qui est régulièrement révisée.

Ce groupe de travail essaye donc de recenser tous les caractères, symboles, glyphes, lettres, nombres, idéogrammes, logogrammes du monde entier afin de constituer **l'Universal Character Set** (UCS).

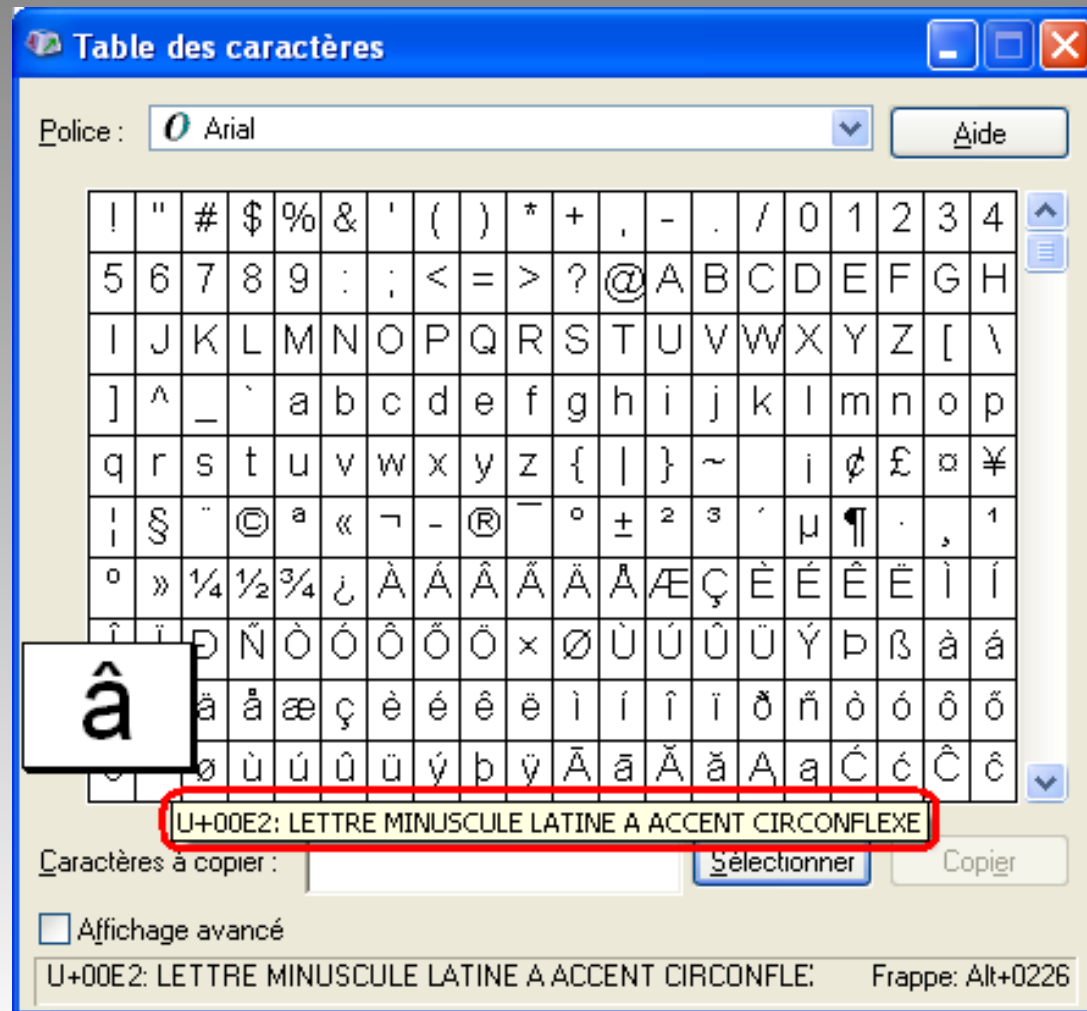
Les caractères abstraits

Le gros défi à surmonter est qu'un même symbole peut être vu de différentes manières d'une langue à l'autre.

Pour contourner ce problème, il a fallu d'utiliser des caractères abstraits (a ... z, ^, ", Щ, گ ...) qui sont considérés comme des symboles de base.

Certains symboles usuels sont donc décomposés en caractères abstraits : par exemple le « â » est vu comme un « a » et un « ^ ».

Les caractères abstraits



Le début de l'Unicode

À la même époque, Joe BECKER de Xerox à Palo Alto travaillait sur un jeu de caractères universel qu'il nommait Unicode en 1987.



La convergence Unicode – UCS

Plus tard, en 1988, plusieurs industriels, dont Xerox, se réunissent pour former le consortium Unicode.

Le consortium Unicode et le groupe de travail de l'ISO/IEC commencent à collaborer afin de faire converger leurs jeux de caractères respectifs en janvier 1992 (depuis les travaux de ces deux entités sont synchronisés).

Voir <http://www.unicode.org/fr/charts/>
<http://ascii-table.com/unicode.php>

Le plan multilingue de base

L'UCS comprend plus d'un million de caractères abstraits, mais actuellement, seuls les 65536 premiers, qui constituent le plan multilingue de base, sont utilisés. Il nécessite 16 bits, soit 2 octets, pour être codé.

Une certaine compatibilité ascendante est assurée entre ce PMB et les tables existantes : les 256 premiers caractères correspondent à la norme ISO/CEI 8859-1 qui comporte les caractères de la table ASCII 7 bits et nos caractères accentués (de la page de code « latin-1 »).

Une partie du PMB

	!	"	#	\$	%	&	'	()	*	+	,	-	.	/		σ	ϰ	ϱ	ρ	β	℞	ε	Σ	ι	τ	Τ	f	Τ	U	u		
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?		Ů	U	Y	y	Z	z	Ʒ	Ʒ	Ʒ	Ʒ	Ʒ	Ʒ	Ʒ	Ʒ	Ʒ		
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O			+	!	DŽ	Dž	dž	LJ	Lj	lj	NJ	Nj	nj	Ǻ	ǻ	Ǽ	ǽ	
P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_		Ŏ	ö	Ů	ů	Ů	ů	Ů	ů	Ů	ů	Ů	ů	Ů	ů	Ů	ů	
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o		ǻ	Æ	æ	G	g	Ğ	ğ	Ķ	ķ	Q	q	Ō	ō	Ǻ	ǻ	Ǽ	
p	q	r	s	t	u	v	w	x	y	z	{		}	~			DZ	Dz	dz	Ġ	ġ	Hu	Đ	Đ	Đ	Đ	Đ	Đ	Đ	Đ	Đ	Đ	
ı	ç	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯	°		ǻ	Ā	ā	Ē	ē	Ê	ê	Ĭ	ĭ	Ĭ	ĭ	Ō	ō	Ō	ō	Ō	ō
±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿	À		ř	Ř	ř	Ů	ů	Ů	ů	Ş	ş	Ţ	ţ	Ȣ	ȣ	Ȥ	ȥ	Ȧ	ȧ
Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ĭ	ĭ	Ĭ	ĭ	Đ		đ	đ	đ	Z	z	Á	á	Ě	ě	Ō	ō	Ō	ō	Ō	ō	Ō	ō
Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß	à		ō	Ÿ	ÿ	Ł	ł	Ł	ł	Ł	ł	Ł	ł	Ł	ł	Ł	ł	Ł	ł
á	â	ã	ä	å	æ	ç	è	é	ê	ë	ĭ	ĭ	ĭ	ĭ	ð		¿	¿	¿	¿	¿	¿	¿	¿	¿	¿	¿	¿	¿	¿	¿	¿	¿
ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ	Ā		ɑ	ɑ	ɑ	ɑ	ɑ	ɑ	ɑ	ɑ	ɑ	ɑ	ɑ	ɑ	ɑ	ɑ	ɑ	ɑ	ɑ
ā	Ă	ă	Ą	ą	Ć	ć	Ĉ	ĉ	Ĉ	ĉ	Ĉ	ĉ	Ĉ	ĉ	Đ		g	g	g	g	g	g	g	g	g	g	g	g	g	g	g	g	g
đ	Ē	ē	Ĕ	ĕ	Ė	ė	Ę	ę	Ė	ė	Ġ	ġ	Ġ	ġ	Ġ		η	η	η	η	η	η	η	η	η	η	η	η	η	η	η	η	η
ġ	Ģ	ģ	Ĥ	ĥ	Ħ	ħ	Ĭ	ĭ	Ĭ	ĭ	Ĭ	ĭ	Ĭ	ĭ	ı		Ɓ	Ɓ	Ɓ	Ɓ	Ɓ	Ɓ	Ɓ	Ɓ	Ɓ	Ɓ	Ɓ	Ɓ	Ɓ	Ɓ	Ɓ	Ɓ	Ɓ
ı	ı	ıj	Ĵ	ĵ	ķ	ķ	ķ	Ĵ	Ĵ	Ĵ	Ĵ	Ĵ	Ĵ	Ĵ	ı		z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z
Ł	ł	Ń	ń	Ņ	ņ	Ň	ň	ň	Ň	ň	Ň	ň	Ň	ň	Ň		z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z
ő	Œ	œ	Ŕ	ŕ	Ŗ	ŗ	Ŗ	ŗ	Ŗ	ŗ	Ŗ	ŗ	Ŗ	ŗ	Ŗ		z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z
š	Ţ	ţ	Ť	ť	Ŧ	ŧ	Ũ	ũ	Ū	ū	Ū	ū	Ū	ū	Ū		h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h	h
ū	Ů	ů	Ű	ű	Ų	ų	Ŷ	ŷ	Ÿ	ž	ž	ž	ž	ž	ž		ſ	<	>	^	v	^	v	^	v	^	v	^	v	^	v	^	v
Ɓ	Ƃ	ƃ	Ƅ	ƅ	Ɔ	Ƈ	ƈ	Ɖ	Ɗ	Ƌ	ƌ	ƍ	Ǝ	Ə	Ɛ		ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı	ı
Ƒ	ƒ	Ɠ	Ɣ	ƕ	Ɩ	Ɨ	Ƙ	ƙ	ƚ	ƛ	Ɯ	Ɲ	ƞ	Ɵ	Ơ		<	>	^	v	^	v	^	v	^	v	^	v	^	v	^	v	^

Le cas particulier du type char

Le type char permet de stocker un nombre entier compris entre -128 et 127, et le type unsigned char est utilisé pour manipuler des nombres entiers compris entre 0 et 255.

Ce nombre entier peut représenter :

- une « vraie » valeur numérique, utilisée dans un calcul
- un **numéro de symbole**, par rapport à la table **ASCII**

Des nouveaux types

Avec l'arrivée de l'**Unicode**, la norme C a été adaptée de manière à gérer ces caractères codés sur 16 bits, grâce à l'introduction de nouveaux types :

- `wchar_t`
- `char16_t`
- `char32_t`

La description détaillée de ces nouveaux types ne sera pas faite dans le cadre de ce cours. Vous pouvez obtenir des informations complémentaires grâce à l'URL ci-dessous :

<http://openclassrooms.com/courses/mettez-des-accent-dans-vos-programmes-avec-le-type-wchar-t>

3 types

Le langage C propose 3 types de base pour stocker des nombres réels :

- **float** ;
- **double** ;
- **long double** (C90).

La représentation interne des nombres dits « à virgules flottantes » suit la norme « IEC 60559:1989 » qui dérive elle-même d'une autre norme (IEEE 754).

Voir http://en.wikipedia.org/wiki/Single-precision_floating-point_format
http://en.wikipedia.org/wiki/Double-precision_floating-point_format
http://en.wikipedia.org/wiki/Long_double
http://en.wikipedia.org/wiki/Extended_precision

Tableau de synthèse

Le tableau ci-dessous énumère les 3 types permettant de stocker des nombres à virgules flottante (qui seront toujours signés) avec la plage de valeurs associée.

Type	Taille	Valeur minimale	Valeur maximale
float	4 o.	$2^{-126} \approx 1.18 \times 10^{-38}$	$(2-2^{-23}) \times 2^{127} \approx 3.402823 \times 10^{38}$
double	8 o.	$2^{-1022} \approx 2.23 \times 10^{-308}$	$(1 + (1-2^{-52})) \times 2^{1023} \approx 1.80 \times 10^{308}$
long double	10 o. ou 16 o.	Le type long double varie selon les implémentation mais il a, au moins, la même capacité qu'un double.	

On considère les macros définies dans `float.h` :

- `FLT_MAX` (valeur max. d'un `float`)
- `DBL_MAX` (valeur max. d'un `double`)
- `LDBL_MAX` (valeur max. d'un long `double`)
- `FLT_MIN` (valeur min. d'un `float`)
- `DBL_MIN` (valeur min. d'un `double`)
- `LDBL_MIN` (valeur min. d'un long `double`)

Modifiez le programme précédent de manière à afficher des informations concernant les 3 types utilisés pour stocker des réels (nombres à virgules flottantes).

Cette fois-ci, on peut utiliser les formats suivants :

- `%f` permet d'écrire un nombre de type `float`
- `%lf` permet d'écrire un nombre de type `double`
- `%Lf` permet d'écrire un nombre de type `long double`

On peut aussi utiliser la notation scientifique avec les formats ci-dessous :

- `%e` pour un nombre de type `float`
- `%le` pour un nombre de type `double`
- `%Le` pour un nombre de type `long double`

Voir <http://www.cplusplus.com/reference/cfloat/>

On crée une variable en langage C en respectant la syntaxe ci-dessous :

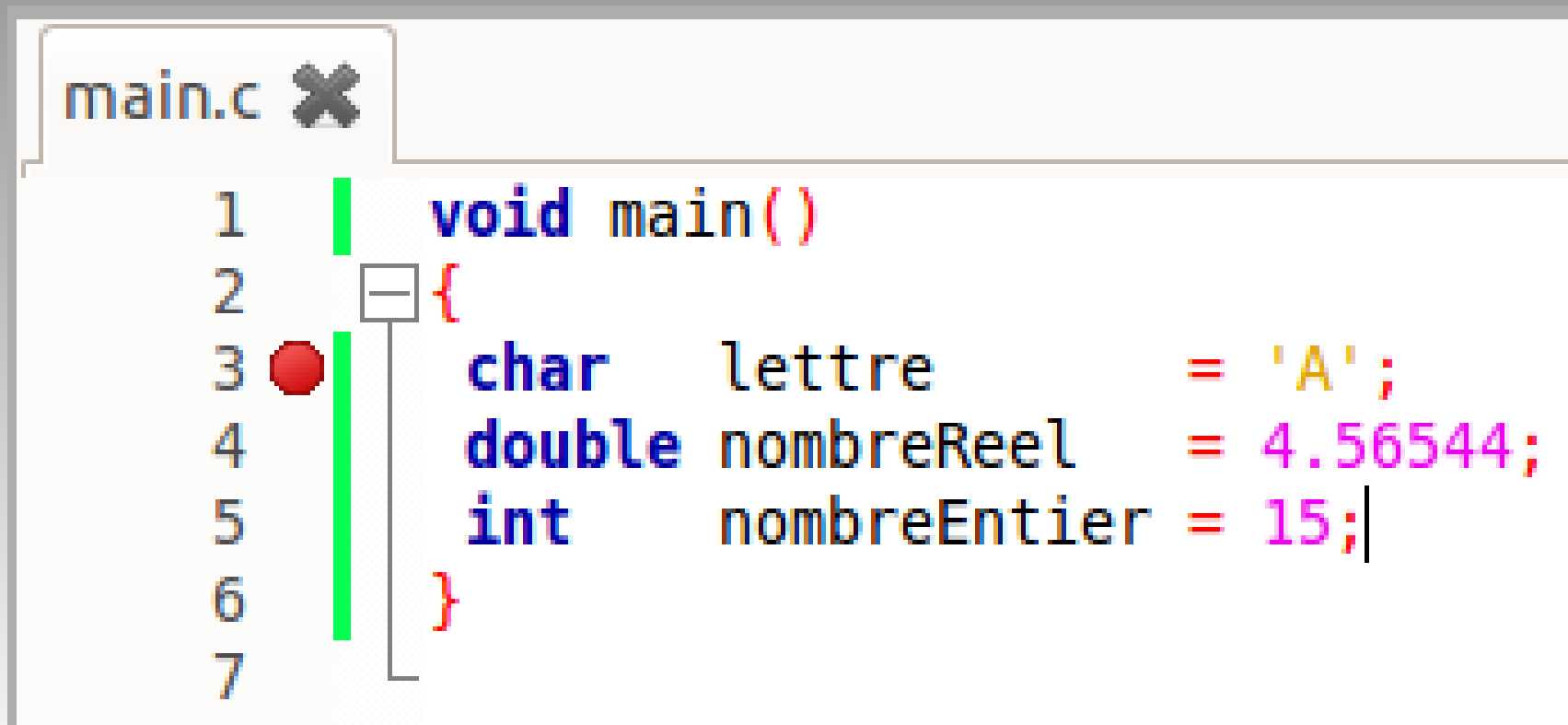
```
type nomVariable;
```

Remarques : Ne pas oublier le « ; » qui sert de délimiteur d'instructions pour le compilateur (sous peine d'obtenir une erreur de compilation)

La syntaxe du langage C est inversée par rapport à celle du langage algorithmique

Le programme de test

Voici un programme qui crée quelques variables. On place un point d'arrêt et on l'exécute en mode « débogage ».



The image shows a code editor window titled 'main.c' with a close button. The code is as follows:

```
1 void main()  
2 {  
3     char    lettre    = 'A';  
4     double  nombreReel    = 4.56544;  
5     int     nombreEntier = 15;  
6 }  
7
```

Visual details: A red circle breakpoint is set on line 3. A green vertical bar highlights the code block from line 2 to line 6. A small square icon is next to the opening brace on line 2.

Suivi des variables

On clique sur l'icône « Debugging window » puis on choisit l'item « Watches » dans le menu contextuel.

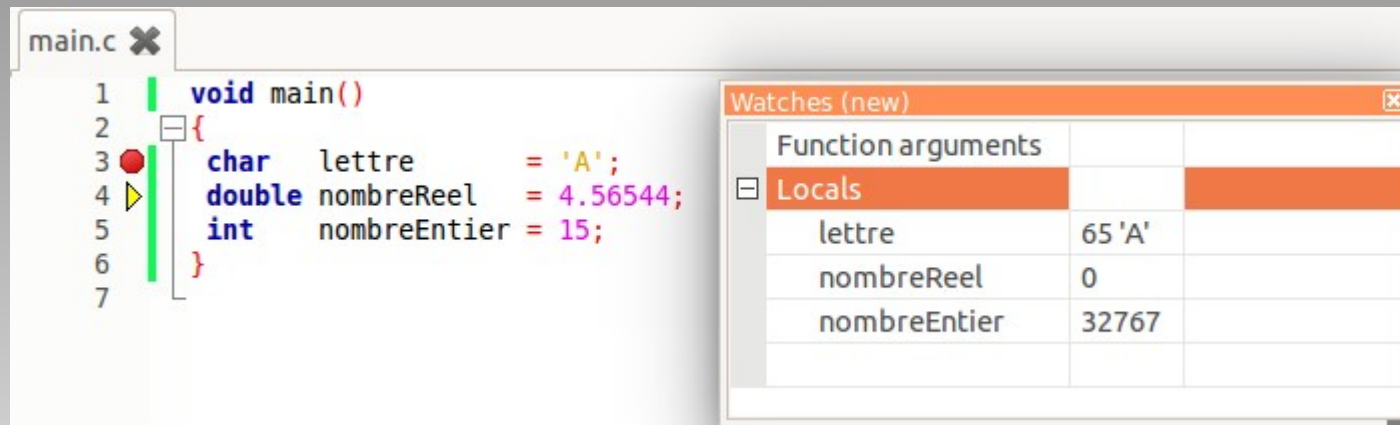


On obtient alors la petite fenêtre ci-dessous.

Watches (new)		
Function arguments		
Locals		
lettre	-1 '\377'	
nombreReel	0	
nombreEntier	32767	

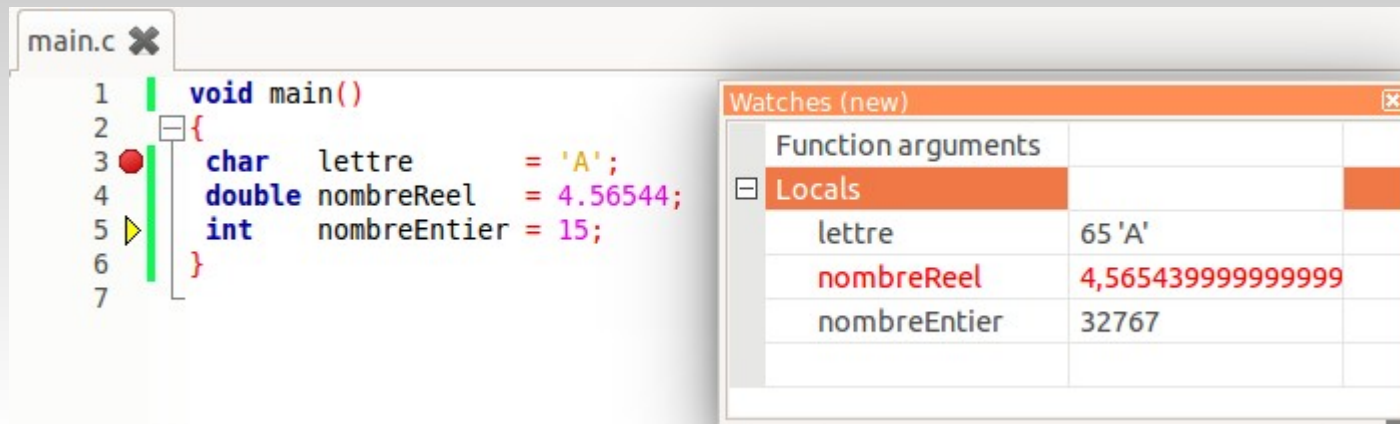
Suivi des variables

Au fur et à mesure de l'avancement dans le programme, on constate que les variables contiennent les bonnes valeurs (alors qu'au départ, elles contiennent n'importe quoi...).



```
1 void main()
2 {
3     char lettre = 'A';
4     double nombreReel = 4.56544;
5     int nombreEntier = 15;
6 }
7
```

Watches (new)		
Function arguments		
Locals		
lettre	65 'A'	
nombreReel	0	
nombreEntier	32767	



```
1 void main()
2 {
3     char lettre = 'A';
4     double nombreReel = 4.56544;
5     int nombreEntier = 15;
6 }
7
```

Watches (new)		
Function arguments		
Locals		
lettre	65 'A'	
nombreReel	4,5654399999999999	
nombreEntier	32767	

Constantes



Deux types de constantes

Comme le langage algorithmique, le langage C permet de définir des constantes (par exemple, une valeur approchée de π , de φ ...).

Il existe deux manières de définir ces constantes :

- en utilisant la directive **#define** du pré-processeur ;
- en utilisant le mot-clé **const** du langage C.

Chacune présente des avantages et des inconvénients.

Voir http://www.tutorialspoint.com/cprogramming/c_constants.htm

L'utilisation de #define

Cette première manière permet de créer des **constantes symboliques** en respectant la syntaxe ci-dessous :

```
#define NOM_CONSTANTE valeur
```

Le pré-processeur **remplace** toutes les occurrences de « NOM_CONSTANTE » par « valeur » dans le fichier de code où il est placé.

Remarques : il s'agit d'une **directive du pré-processeur** et non d'une instruction du langage C, il ne faut donc **pas mettre de « ; »**.

Voici un exemple d'utilisation d'une constante symbolique.

```
#define PI 3.14159
```

```
void main ()  
{  
    double rayon = 4.8;  
    double perimetre = 2 * PI * rayon;  
}
```

PI est remplacé en mémoire par 3.14159, par le pré-processeur avant la phase de compilation

L'utilisation de const

Cette seconde manière permet de créer des **constantes** en figeant la valeur des variables :

```
const type NOM_CONSTANTE = valeur;
```

Il s'agit, cette fois-ci d'une instruction, il est donc nécessaire de **mettre un « ; »**.

Comme la valeur de la constante ne peut être modifiée, il est nécessaire de **l'initialiser dès sa création**.

Voici un exemple d'utilisation d'une constante symbolique.

```
void main ()  
{  
    const double PI = 3.14159;  
    double rayon = 4.8;  
    double perimetre = 2 * PI * rayon;  
}
```

PI est remplacé en mémoire par
3.14159, par le compilateur
durant la phase de compilation

Comparaison des deux méthodes

	#define	const
Occupation mémoire	La constante symbolique n'occupe pas de place en mémoire (remplacement de texte)	La constante occupe une place en mémoire, équivalente à celle d'une variable
Portée	La constante est reconnue dans tout le fichier dans laquelle elle est définie	La constante est reconnue dans le bloc de code dans lequel elle est définie ou au niveau du programme global si elle est déclaré en dehors des fonctions
Accessibilité	La constante n'est pas accessible car elle n'a pas d'existence en mémoire	La constante peut être accédée par nom ou par son adresse comme une variable normale

Expressions et opérateurs



Notion identique à celle de l'algorithmique

Comme pour l'algorithmique, une expression est une combinaison d'opérateurs, unaires, binaires ou ternaires, et d'opérandes qui peuvent être :

- des valeurs écrites « en dur » dans l'expression ;
- des constantes ;
- des variables ;
- des « sous-expressions » qui seront remplacées de façon dynamique par leur valeur, lors de leur évaluation

L'évaluation des expressions complexes s'effectuent en commençant par les « sous-expressions » les plus imbriquées et en tenant compte des niveaux de priorité des opérateurs (par exemple \times est prioritaire sur $+$).

L'opérateur d'affectation

L'opérateur est noté cette fois-ci avec **UN** signe égal :

=

Il permet d'écrire une **valeur**, placé à **droite** de la flèche, dans la variable ou la constante, dont le **nom** est écrit à **gauche** de la flèche.

```
val = 10;  
b = val;  
c = b + c;  
val = val + 1;
```

La permutation d'information

Comme pour l'algorithmique, l'affectation est une opération destructrice.

Ainsi, si on souhaite permuter deux valeurs, il est nécessaire d'utiliser une troisième variable pour un stockage temporaire.

```
void main ()
{
    double a = 4.856;
    double b = 3.57;
    double temp;

    temp = a;
    a     = b;
    b     = temp;
}
```

Les opérateurs arithmétiques

On retrouve les opérateurs arithmétiques utilisés dans le langage algorithmique :

- + (addition)
- − (soustraction)
- * (multiplication)
- / (division)
- % (modulo ou reste d'une division entière)

Combinaison d'opérateurs

Le langage C propose des opérateurs combiné qui effectuent une opération arithmétique puis une affectation :

- $+=$ (addition puis affectation)
- $-=$ (soustraction puis affectation)
- $*=$ (multiplication puis affectation)
- $/=$ (division puis affectation)
- $\%=$ (modulo puis affectation)

Exemples :

- $\ll a \ += \ 2; \gg$ équivaut à $\ll a \ = \ a \ + \ 2; \gg$
- $\ll b \ \ *= \ 4; \gg$ équivaut à $\ll b \ = \ b \ * \ 4; \gg$
- $\ll c \ \ /= \ b \ + \ a; \gg$ équivaut à $\ll c \ = \ c \ / \ (b+a); \gg$

Incrémentation et décrémentation

Le langage C propose d'autres opérateurs unaire qui permettent de diminuer ou d'augmenter une variable de 1 unité :

- ++ (incrémentement)
- -- (décrémentement)

Exemples :

- « a++ ; » équivaut à « a += 1 ; » et « a = a + 1 ; »
- « b-- ; » équivaut à « b -= 1 ; » et « b = b - 1 ; »

Incrémentation et décrémentation

Le « ++ » peut être placé :

- **avant** la variable : il s'agit d'une **pré**-incrémentation
- **après** la variable : il s'agit d'une **post**-incrémentation

Ce positionnement a une influence sur l'ordre de traitement des opérateurs et éventuellement sur le résultat. Cette remarque est aussi valable pour l'opérateur « -- ».

```
int a =10;
```

```
/* On fait l'affectation puis l'incrémentation */  
int b = a++; /* b contient 10 et a contient 11 */
```

```
/* On fait l'incrémentation puis l'affectation */  
int c = ++a; /* c et a contiennent 12 */
```

Coder le petit exemple ci-dessous :

```
void main ()
{
    int a =10;

    /* On fait l'affectation puis l'incréméntation */
    int b = a++; /* b contient 10 et a contient 11 */

    /* On fait l'incréméntation puis l'affectation */
    int c = ++a; /* c et a contiennent 12 */
}
```

Suivre le fonctionnement de ce programme en mode « debugage, pas-à-pas » afin de suivre l'évolution des variables.

Les opérateurs de comparaison

Le langage C utilise aussi les opérateurs de comparaison ci-dessous :

- `<`
- `>`
- `<=` (correspond à \leq)
- `>=` (correspond à \geq)
- `==` **(ne surtout pas confondre avec l'affectation !)**
- `!=` (correspond à \neq)

Les opérateurs de comparaison

Les opérateurs de comparaison renvoient un résultat booléen (vrai si le test est vérifié et faux dans le cas contraire).

Il faut cependant remarquer que le type `_bool` n'a été introduit qu'à partir de la norme « C99 ».

Beaucoup de programmes utilisent un nombre entier pour représenter une valeur booléenne :

- **0** représente « **faux** » ;
- un nombre **différent de 0** (souvent 1) représente « **vrai** ».

Instructions d'affichage et de saisie



La bibliothèque `stdio.h`

Un programme doit communiquer avec le monde extérieur pour échanger des informations ce qui implique d'utiliser les composants de l'ordinateur (clavier, souris, écran...).

Des morceaux de code sont donc mis à disposition du programmeur pour accéder à ces composants.

Ils sont présentés sous forme de **fonctions**, groupées dans une **bibliothèque** appelée « **stdio.h** » : il est donc nécessaire d'écrire « **#include <stdio.h>** » au début du programme.

« `stdio` » signifie « standard input/output ».

La fonction `printf`

La fonction `printf` permet d'écrire des informations dans la sortie standard (autrement dit la console). Il existe d'autres instructions que nous étudierons un peu plus tard.

Pour afficher un texte simple, on l'écrit simplement entre parenthèses et entre guillemets.

Exemple :

```
printf ("Bonjour !");
```

Les « " » et le « \ »

Les « " » sont utilisés pour délimiter la chaîne de caractères.

Si on souhaite les afficher, ils doivent être précédés d'un anti-slash « \ » qui joue le rôle de **caractère d'échappement**.

Pour afficher le « \ », il est nécessaire de le doubler : « \\ ».

Exemple :

```
printf ("Il a dit \"Bonjour\" !");  
printf ("On veut écrire  \\");
```


D'autres caractères spéciaux

Le caractère d'échappement est utilisé pour d'autres caractères spéciaux (non affichables) :

- `\n` correspond au passage à la ligne ;
- `\t` correspond à la tabulation ;
- `\r` permet de revenir au début de la même ligne.

Exemple :

```
printf ("Afficher un texte \n et passer à la ligne");
```

```
Afficher un texte  
passer à la ligne
```

Voir http://www.linux-france.org/prj/embedded/sdcc/sdcc_course.formatted_io.html

Sous google, tapez « man printf » pour trouver des sites web proposant le manuel en ligne du C

Le caractère « % »

Le caractère « % » est aussi un caractère spécial qui permet d'utiliser le mécanisme de formatage.

Si on souhaite afficher un « % », il est nécessaire de le doubler : « %% ».

Exemple :

```
printf ("Il y 50%% de chances de trouver...");
```

Le mécanisme de formatage

Ce mécanisme permet, dans le cas du `printf`, d'afficher la valeur d'une information sous une « forme » symbolisée par une lettre :

- **%d, %ld, %lld** : pour un nombre entier signé ;
- **%u, %lu, %llu** : pour un nombre entier non signé ;
- **%x, %X** pour un nombre entier en hexadécimal ;
- **%f, %lf, %Lf** pour un nombre réel ;
- **%e, %le, %Le** pour un nombre réel avec la notation scientifique ;
- **%c** pour un caractère ;
- **%s** pour une chaîne de caractères (voir plus loin) ;
- **%p** pour une adresse mémoire (voir plus loin).

Le mécanisme de formatage

Ces éléments de formatage sont remplacés, à la volée, par les valeurs des variables, écrites juste après la chaîne de caractères à afficher :

Exemple :

```
int a = 10;  
double b=20;  
printf ("a vaut %d et b vaut %f\n", a, b);
```



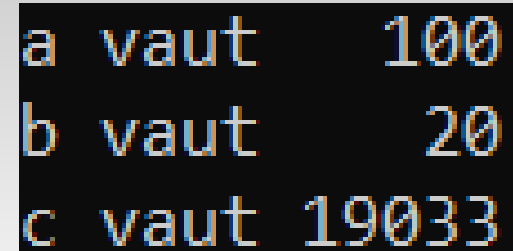
Le mécanisme de formatage

On peut ajouter un premier nombre entre le « % » et la lettre afin d'indiquer le nombre de caractères réservés pour l'affichage (ce qui permet d'aligner des nombres).

Exemple :

```
int a = 100;  
int b = 20;  
int c = 19033;
```

```
printf ("a vaut %5d \n", a);  
printf ("b vaut %5d \n", b);  
printf ("c vaut %5d \n", c);
```



```
a vaut   100  
b vaut    20  
c vaut 19033
```

Le mécanisme de formatage

Dans le cas des nombres réels, on peut aussi préciser le nombre de chiffres à écrire après la virgule en ajoutant « . » suivi de cette précision.

Exemple :

```
float a = 20.45;  
float b = 20.451;  
float c = 20.45564;
```

```
printf ("a vaut %6.3f \n", a);  
printf ("b vaut %6.3f \n", b);  
printf ("c vaut %6.3f \n", c);
```

```
a vaut 20.450  
b vaut 20.451  
c vaut 20.456
```

La fonction scanf

La fonction **scanf** permet de lire des informations depuis l'entrée standard

Cette fonction s'appuie sur le mécanisme de formatage pour « filtrer » les informations (accepter la saisie si elle respecte un certain format).

Exemple :

```
int a ;  
scanf ("%d", &a);
```

Le symbole « & »

Pour enregistrer la valeur saisie dans une variable, il est nécessaire de créer la variable puis de la spécifier dans la fonction `scanf` en la faisant précéder de « **&** ».

On expliquera la signification de ce symbole lorsqu'on abordera la notion d'adresse, de pointeur...

Exemple :

```
double var ;  
scanf ("%lf", &var);
```


Exemple

Voici l'exemple d'un programme qui calcule le périmètre d'un cercle en fonction du rayon saisi par l'utilisateur.

```
#include <stdio.h>

#define PI 3.14159

void main ()
{
    float r;

    printf ("Saisir un rayon : ");
    scanf ("%f", &r);

    printf ("Le perimètre est : %.2f\n", (2*r*PI));
}
```

Structure d'un programme simple en langage C



Des petits programmes

Dans un premier temps, nous n'allons considérer que des petits programmes qui tiennent dans **un seul fichier** de code.

Plus tard, lorsque nous souhaiterons créer des programmes plus complexes, nous éprouverons le besoin de mieux le code source en le scindant en plusieurs fichiers (on parlera alors de « modularisation »).

Une première structure simpliste

La structure générale d'un petit programme C est la suivante :

```
#include <stdio.h>    /* inclure une bibliothèque */  
  
#define PI 3.14159    /* constante symbolique  
  
void main ()          /* Point d'entrée du programme */  
{  
  
    /* Déclaration des variables  
       et des constantes locales */  
  
    /* instruction */  
  
}
```

Les commentaires

Le couple « /* » et « */ » permet de délimiter des zones de textes qui ne sont pas prises en considération par le pré-processeur et le compilateur.

Cela permet de mettre :

- des descriptions d'un morceau de code
- des « cartouches » indiquant le nom du programme, le nom du développement, le numéro de version...

Cela permet aussi d'inhiber des zones de codes pour faciliter la recherche de bugs.

```
/* **** */
/* Mon premier programme */
/* ----- */
/* Auteur   : Moi          */
/* Date      : 28/04/2015   */
/* Version   : 1.0          */
/* **** */

#include <stdio.h>

#define PI 3.14159

void main ()
{
    float r;

    printf ("Saisir un rayon : ");
    scanf ("%f", &r);

    printf ("Le périmètre est : %.2f\n", (2*r*PI));
}
```

Structures de contrôle conditionnelles



Rappel de la syntaxe en algorithmique

Dans un algorithme, cette structure conditionnelle s'écrit de la manière suivante :

```
SI condition
| ALORS instructions
| SINON instructions
FIN_SI
```

Voir http://fr.wikibooks.org/wiki/Algorithmique_imp%C3%A9rative/Condition
<http://isn.codelab.info/ressources/algorithmique/memo-pseudo-codes/>

Traduction en langage C

Dans un programme en langage C, sa traduction s'écrit de la manière suivante :

```
if (condition)
    instruction;
else instruction;
```

Écriture dans un algorithme

Remarques :

- La structure `if ... else` n'accepte qu'**une seule instruction !**
- L'indentation est très fortement conseillée voire obligatoire pour conserver une bonne lisibilité du code C.
- Le bloc « ALORS » (juste après `if`) est obligatoire alors que le bloc « SINON » (juste après `else`) est facultatif.
- La structure `if ... else`, dans son ensemble, est **considérée comme une seule instruction.**

Notion de blocs de code

Un bloc de code (un bloc d'instructions) est délimité par des accolades.

Depuis l'extérieur, ce bloc de code **est vu comme une seule instruction**.

```
{  
    instruction_1;  
    instruction_2;  
    instruction_3;  
    instruction_4;  
}
```

if ... else avec un bloc de code

Si on souhaite exécuter plusieurs instructions au sein d'une structure de contrôle `if ... else`, on utilise un bloc de code

```
if (condition)
{
    instruction_1;
    instruction_2;
}
else {
    instruction_3;
    instruction_4;
}
```

Exemple

Voici l'exemple d'un programme qui effectue la division de deux nombres fourni par l'utilisateur

```
// Programme division
```

```
void main ()  
{  
    float a = 0;  
    float b = 0;  
    float c = 0;  
  
    scanf ("%f", &a);  
    scanf ("%f", &b);
```

Exemple

```
if (b == 0)
{
    printf ("Division impossible");
}
else {
    c = a/b;
    printf ("%f", c);
}
```

Remarques :

- Bien faire attention au « == » pour la comparaison.
- Les accolades du « ALORS » ne sont pas obligatoires car il n'y a qu'une seule instruction.

Deux if ... else imbriqués

On ajoute un test imbriqué

```
if (b == 0)
{
    printf ("Division impossible");
}
else {
    if (b == 2)
    {
        printf ("La moitié de a vaut :");
        c = a/2;
        printf ("%f", c);
    }
    else {
        c = a/b;
        printf ("%f", c);
    }
}
```

Rappel de la syntaxe en algorithmique

Dans un algorithme, cette structure conditionnelle s'écrit de la manière suivante :

```
SELON nomVariable PARMI :  
    Valeur 1      : instruction 1  
    Valeur 2      : instruction 2  
    Valeur 3      : instruction 3  
    PAR_DEFAULT : instruction 4  
FIN_SELON
```


Traduction en langage C

Dans un programme en langage C, sa traduction s'écrit de la manière suivante :

```
switch (nomVariable)
{
    case Valeur_1 : instruction_1; break;
    case Valeur_2 : instruction_2; break;
    case Valeur_3 : instruction_3; break;
    default      : instruction_4;
};
```

L'instruction break

L'instruction **break** permet de « casser » une structure de contrôle afin d'en **sortir prématurément**.

Dans le cas du switch ... case, si on ne met pas l'instruction `break` et que `nomVariable` vaut « Valeur_1 », on exécute alors les instructions 1 à 4.

```
switch (nomVariable)
{
    case Valeur_1 : instruction_1;
    case Valeur_2 : instruction_2;
    case Valeur_3 : instruction_3;
    default      : instruction_4;
};
```

Exercice

Écrire un programme qui demande le nombre d'articles contenu dans le panier.

- si ce nombre est 0, on doit afficher le message « aucun article » ;
- si ce nombre est 1, on doit afficher le message « un seul article » ;
- S'il s'agit d'une autre valeur, on doit :
 - afficher le message « N articles » (N a remplacer par la valeur), si N est positif
 - afficher le message « valeur erronée », si N est négatif

Tester le programme en mode « debogage, pas-à-pas » afin de vérifier, notamment, l'effet de l'instruction break.

Structures de contrôle itératives



Rappel de la syntaxe en algorithmique

Dans la version pseudo-code de l'algorithme, la boucle POUR est écrite selon la syntaxe ci-dessous :

```
POUR num DE dep A arr PAS saut FAIRE  
|   Instruction 1  
|   Instruction 2  
FIN_POUR
```

Remarque : si le pas est omis, il est considéré comme valant 1, par défaut.

Traduction en langage C

Dans un programme en langage C, sa traduction s'écrit de la manière suivante :

```
int num;
```

```
for (num = dep ; num <= arr ; num ++)  
{  
    instruction_1;  
    instruction_2;  
}
```

Exemple

Le programme ci-dessous énumère les entiers entre 0 et 10

```
void main ()  
{  
    int i;  
  
    for (i=0; i<=10; i++)  
        printf ("%d\n", i);  
}
```

La variable `i`, joue le rôle de **variable d'itération**.

L'instruction break

On peut utiliser l'instruction `break` pour quitter une boucle `for` de façon prématurée.

```
void main ()
{
    int i;
    int j=5;

    for (i=0; i<=10; i++)
    {
        if (i>j)
        {
            printf ("Trop fatigué, j'arrête !\n");
            break;
        }
        printf ("%d\n", i);
    }
}
```


L'instruction continue

On peut utiliser l'instruction continue pour terminer une itération de façon prématurée.

```
void main ()
{
    int i;

    for (i=-5; i<=5; i++)
    {
        if (i==0)
        {
            printf ("On saute le 0 !\n");
            continue;
        }
        printf ("10/%d=%f\n", i, (10.0/i));
    }
}
```

Syntaxe en pseudo-code

Dans la version pseudo-code de l'algorithme, la boucle TANT_QUE est écrite selon la syntaxe ci-dessous :

```
TANT_QUE condition FAIRE  
|   Instruction 1  
|   Instruction 2  
FIN_TANT_QUE
```

Remarque : si la condition est toujours vraie, on obtient alors une **boucle infinie**.

Traduction en langage C

Dans un programme en langage C, sa traduction s'écrit de la manière suivante :

```
while (condition)
{
    instruction_1;
    instruction_2;
}
```

Exemple

On peut utiliser la boucle `while` dans le cadre d'une recherche par dichotomie :

```
// Recherche de  $x^2-6=0$  par dichotomie
```

```
void main ()
{
    float borneSup=10;
    float borneInf= 0;
    float x;
    float c;

    while ((borneSup - borneInf)>0.05)
    {
        x = (borneSup + borneInf) / 2;
        c = x*x-6;
        if ( c > 0 ) borneSup = x;
        if ( c < 0 ) borneInf = x;
        if ( c == 0 ) break;
    }

    printf ("x vaut %.4f\n", x);
}
```

Syntaxe en pseudo-code

Dans la version pseudo-code de l'algorithme, la boucle TANT QUE est écrite selon la syntaxe ci-dessous :

REPETER

| Instruction 1

| Instruction 2

TANT_QUE condition

Remarque : si la condition est toujours vraie, on obtient, aussi dans ce cas de figure, une **boucle infinie**.

Traduction en langage C

Dans un programme en langage C, sa traduction s'écrit de la manière suivante :

```
do
{
    instruction_1;
    instruction_2;
}
while (condition);
```

Exemple

On peut utiliser la boucle do ... while pour demander une information à un utilisateur (qui peut être borné).

```
void main ()
{
    int age;

    do
    {
        printf ("Donnez un age supérieur à 18 ans : ");
        scanf ("%d", &age);
    }
    while (age<=18);
}
```

Exercices



Écrire un programme qui permet de dessiner damier 8x8 en utilisant les lettres X et O.

```
XOXOXOXO
OXOXOXOX
XOXOXOXO
OXOXOXOX
XOXOXOXO
OXOXOXOX
XOXOXOXO
OXOXOXOX
```

Écrire un programme qui permet de dessiner un triangle dont la hauteur est déterminée par l'utilisateur (lors d'une saisie)

```
      *  
    * * *  
  * * * * *  
* * * * * * *  
* * * * * * * *  
* * * * * * * * *  
* * * * * * * * * *
```

L'aire de jeu – Version 1

Écrire un programme qui permet de dessiner une aire de jeu dont les dimensions sont fournies par l'utilisateur

```
#####  
#                                     #  
#                                     #  
#                                     #  
#                                     #  
#                                     #  
#                                     #  
#####
```

L'aire de jeu – Version 2

Modifier le programme de manière à ajouter un joueur au centre de l'aire de jeu en fonction des coordonnées xJoueur et yJoueur.

```
#####  
#                                     #  
#                                     #  
#           0                       #  
#                                     #  
#                                     #  
#                                     #  
#####
```

L'aire de jeu – Version 3

Modifier le programme qui demande à l'utilisateur de choisir une action :

- 1 permet de déplacer le joueur vers le haut ;
- 2 permet de déplacer le joueur vers la gauche ;
- 3 permet de déplacer le joueur vers la droite ;
- 4 permet de déplacer le joueur vers le bas ;
- 5 permet de sortir du programme.

Le programme doit afficher l'aire de jeu, le petit menu puis le texte d'invitation « Choix de l'action ».

Une fois, la saisie effectuée, on affiche de nouveau l'aire de jeu, le menu et le texte d'invitation, ou on quitte le programme.