

Cours du Langage C

Dr Sibiri TIEMOUNOU

Plan du cours

- 1. Chapitre 0 : Généralité sur la programmation**
- 2. Langage C - Notions de base**
- 3. Modularité en Langage C**
- 4. Structures de données**
- 5. Pointeurs**
- 6. Lecture et écriture dans des fichiers**
- 7. Création de jeux 2D (pour le projet de programmation)**

Présentation du cours

▪ Objectifs de ce cours

- ☞ Acquérir les notions de programmation de base du langage C
- ☞ Comprendre la convergence entre l'algorithmie et le langage C

▪ Compétences visées

- ☞ Etre autonome dans le développement d'une application en C
- ☞ Etre capable de lire, comprendre et de résoudre des bugs C

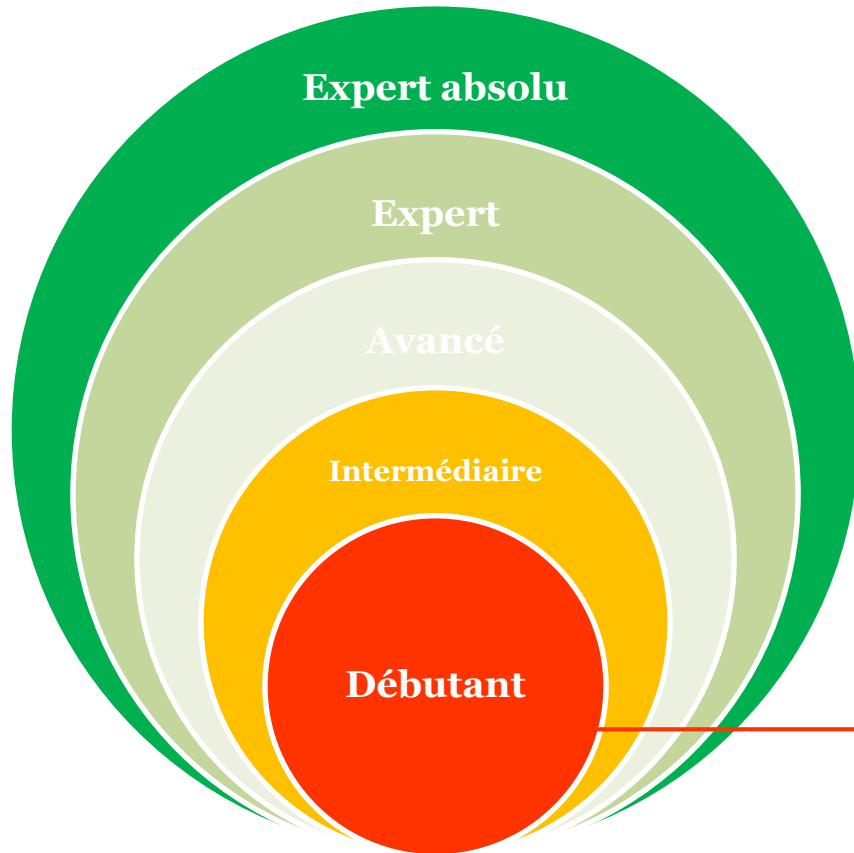
▪ Volume horaire : 50 h

- ☞ Cours magistraux : 18 h
- ☞ TD : 08 h
- ☞ TP : 21 h
- ☞ Devoir : 03 h

▪ Evaluation

- ☞ 1 devoir sur table
- ☞ 1 devoir de maison
- ☞ 1 sujet d'examen

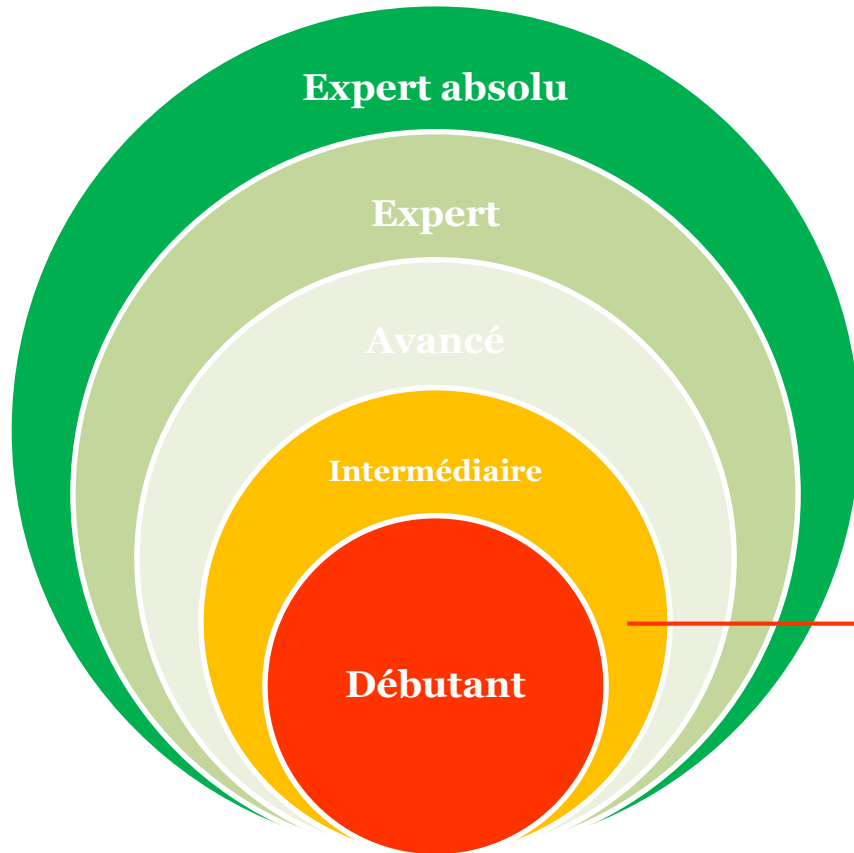
Niveau de compétences



- *Maitrise des concept de base du langage C*

- ✓ Capable d'écrire des codes élémentaires
- ✓ Capable de résoudre des erreurs élémentaire de codage

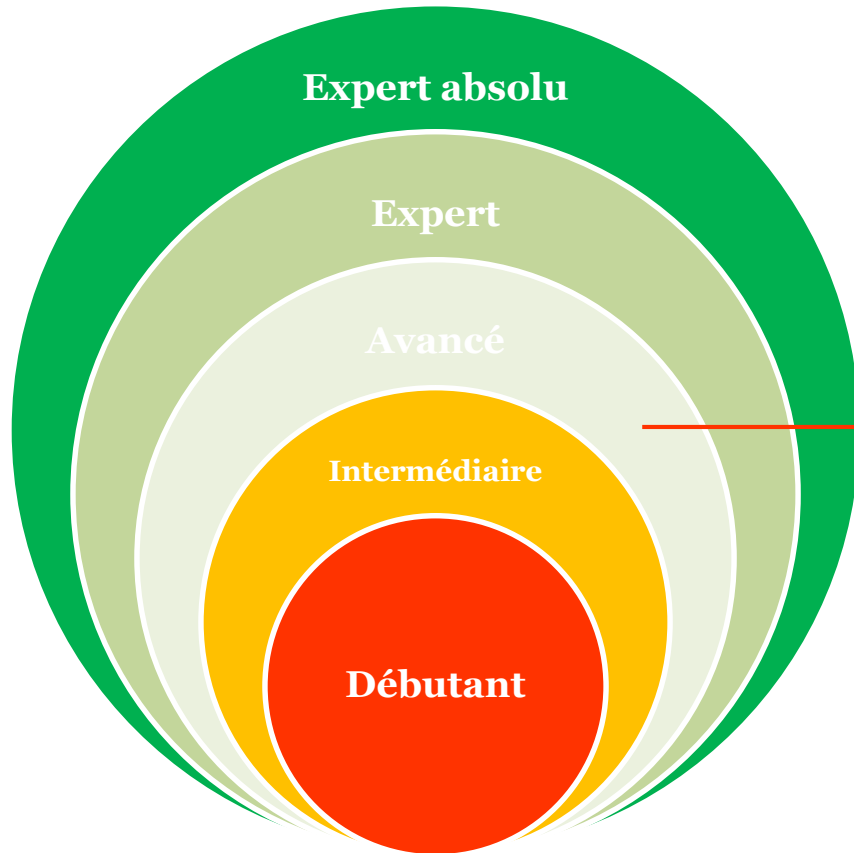
Niveau de compétences



- *Compréhension des notions avancés du C*

- ✓ Utilisation des pointeurs
- ✓ Gestion des structures de données
- ✓ Notion de base en compilation/débogage

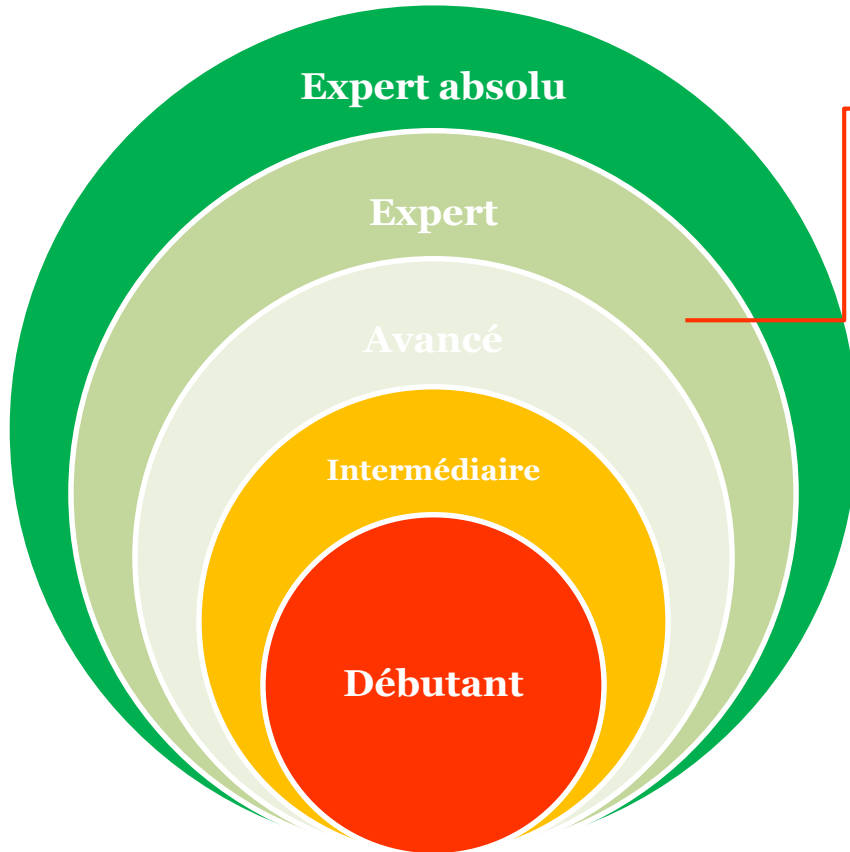
Niveau de compétences



-Aptitude à développer des applications

- ✓ Maitrise des aspects d'optimisation et de la compilation
- ✓ Forte capacité à résoudre des bugs

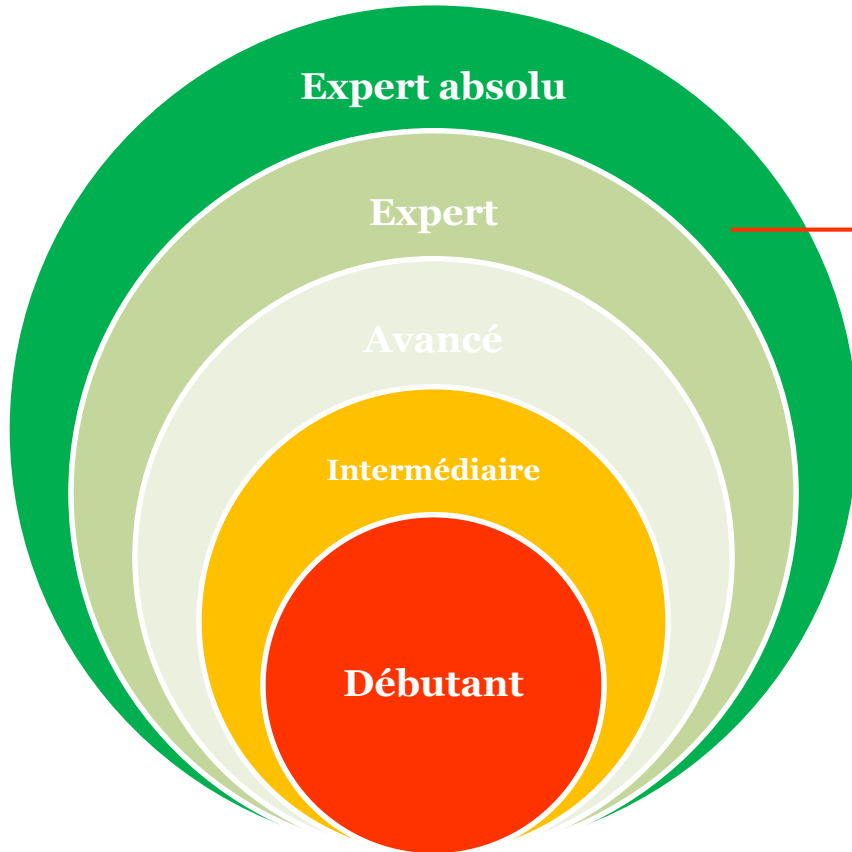
Niveau de compétences



→ **Capacité à gérer un projet de conception d'application multitâche (chef de projet)**

- ✓ Capacité à faire communiquer une application avec d'autres plateformes
- ✓ Gestion de versions d'application
- ✓ Capacité à gérer une équipe en développement informatique

Niveau de compétences



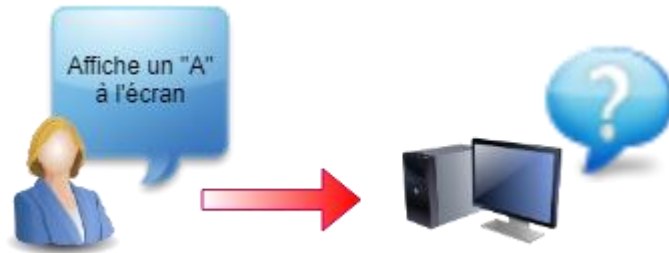
- Aptitude à proposer des améliorations du langage

o. Généralités sur la programmation

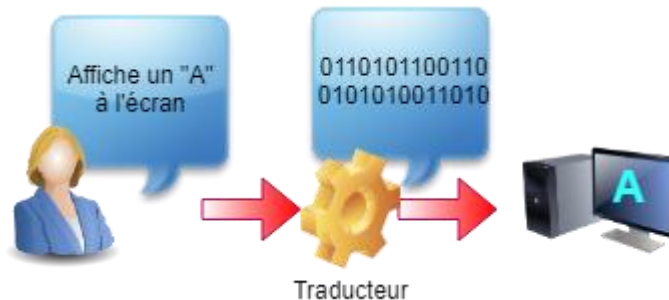
0.1 Langage de programmation

■ Qu'est-ce qu'un langage de programmation?

- ➡ Pour communiquer avec quelqu'un, vous devez parler le même langage
- ➡ Avec un ordinateur, difficile de communiquer car ne comprenant que des 0 et 1



- ➡ Pour communiquer avec un ordinateur, il faut un **interpréteur (ou logiciel)**
 - ➡ Charger de transformer un message en 0 et 1



0.1 Langage de programmation

■ Qu'est-ce qu'un langage de programmation?

☞ Un ordinateur obéit aux ordres que nous lui donnons

☞ Ces ordres s'appellent des **instructions**

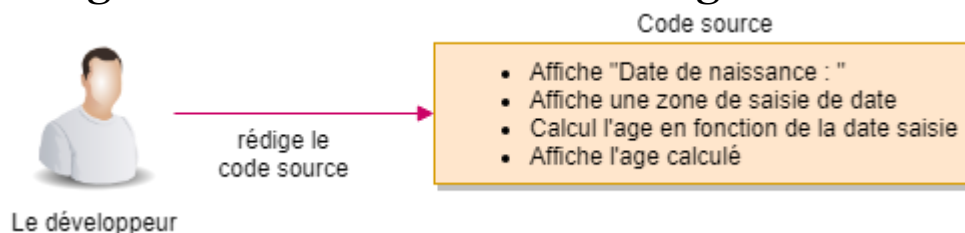
☞ Un **langage** est donc une suite d'instructions

☞ Exemple :

- Affiche « A » à l'écran
- Affiche une zone où l'utilisateur pourra saisir son nom
- Si l'utilisateur clique sur le bouton « fermer », alors le programme se ferme

☞ Un **langage de programmation** permet d'écrire toutes ces instructions (aussi appelé **code source**) dans un langage de comprise par le traducteur

☞ Charger de transformer un message en 0 et 1



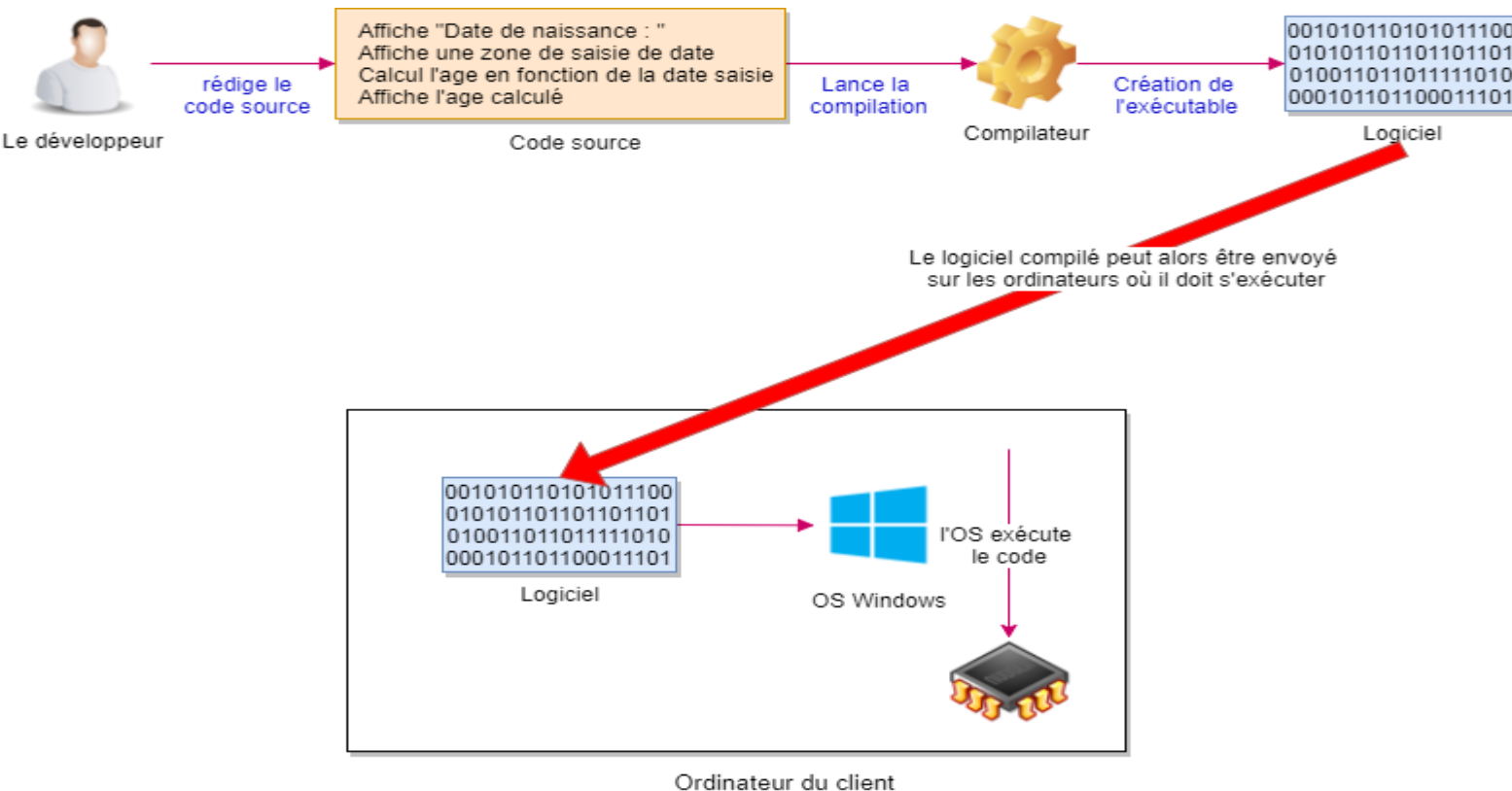
0.2 Type de langages de programmation

▪ Quels sont les types de langage ?

☞ Il existe 2 types de langages

☞ **Langages compilés**

☞ Le code écrit par le développeur est d'abord compilé (conversion du code source en langage machine) avant d'être exécuter



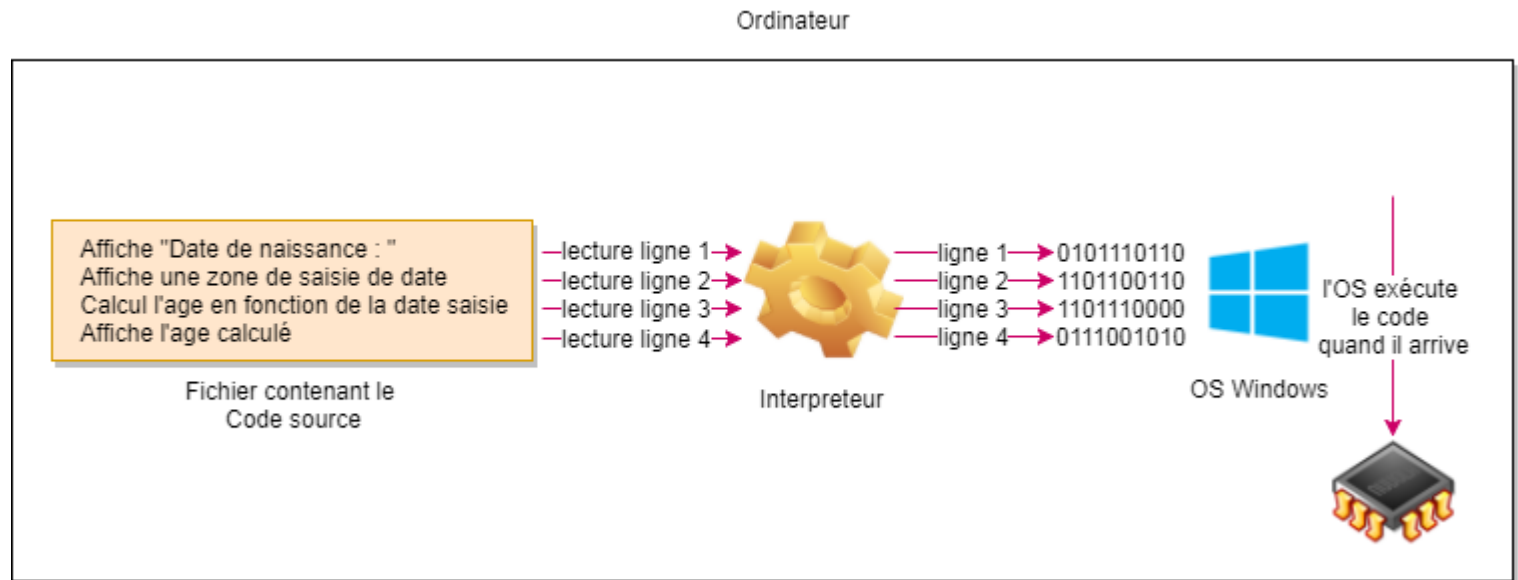
0.2 Type de langages de programmation

▪ Quels sont les types de langage ?

☞ Il existe 2 types de langages

☞ **Langages interprétés**

☞ Ces langages n'ont pas besoin de compilation. Le code source est lu et traduit pour être exécuté



0.2 Type de langages de programmation

▪ Quels sont les niveaux de langages de programmation?

☞ Il existe deux types de niveaux

☞ **Langages bas niveaux**

☞ Il s'agit des langages de programmation très proche du langage machine

☞ Des langages permettant d'accéder et de *manipuler directement les registres et les instructions machines des ordinateurs*

☞ *Avantages*

☺ Il sont donc très rapide en termes d'exécution

☺ Très utilisés dans les domaines suivants:

☺ Informatique embarquée

☺ Création de systèmes d'exploitation

☺ Développement de jeux vidéos

0.2 Type de langages de programmation

▪ Quels sont les niveaux de langages de programmation?

☞ Il existe deux types de niveaux

☞ **Langages bas niveaux**

☞ Il s'agit des langages de programmation très proche du langage machine

☞ Des langages permettant d'accéder et de *manipuler directement les registres et les instructions machines des ordinateurs*

☞ **Inconvénients**

☹ Langage très complexe à appréhender

☹ Moindre erreur pouvant endommager le système d'exploitation de l'ordinateur

0.2 Type de langages de programmation

▪ Quels sont les niveaux de langages de programmation?

☞ Il existe deux types de niveaux

☞ **Langages bas niveaux**

☞ Il s'agit des langages de programmation très proche du langage machine

☞ *Exemple : **Langage Assembleur** : code permettant d'afficher « Bonjour » à l'écran*

```
section .data                                ; Variables initialisées
    Buffer:      db 'Bonjour', 10           ; En ascii, 10 = '\n'. La virgule sert à concaténer
; Les chaines
    BufferSize:  equ $-Buffer               ; Taille de La chaine

section .text                                ; Le code source est écrit dans cette section
    global _start                           ; Définition de l'entrée du programme

_start:                                     ; Entrée du programme

    mov eax, 4                               ; Appel de sys_write
    mov ebx, 1                               ; Sortie standard STDOUT
    mov ecx, Buffer                           ; Chaine à afficher
    mov edx, BufferSize                       ; Taille de La chaine
    int 80h                                  ; Interruption du kernel

    mov eax, 1                               ; Appel de sys_exit
    mov ebx, 0                               ; Code de retour
    int 80h                                  ; Interruption du kernel
```


0.2 Type de langages de programmation

- **Quels sont les niveaux de langages de programmation?**

- ☞ Il existe deux types de niveaux

- ☞ **Langages haut niveaux**

- ☞ Langages de programmation très proche des langages naturels humains (généralement en Anglais)

- ☞ *Avantages*

- ☺ Langage indépendant de la machine (ou du système d'exploitation)
 - ☺ Pas besoin de comprendre le fonctionnement des registres ou processeur d'un ordinateur

0.2 Type de langages de programmation

- **Quels sont les niveaux de langages de programmation?**

- ☞ Il existe deux types de niveaux

- ☞ **Langages haut niveaux**

- ☞ Langages de programmation très proche des langages naturels humains (généralement en Anglais)

- ☞ *Inconvénients*

- ☹ Peu performants dans les situations où les ressources matérielles sont limitées

0.2 Type de langages de programmation

▪ Quels sont les niveaux de langages de programmation?

☞ Il existe deux types de niveaux

☞ **Langages haut niveaux**

☞ Langages de programmation très proche des langages naturels humains (généralement en Anglais)

☞ Facile à utiliser

☞ *Exemple* : langage C : code permettant d'afficher « bonjour » à l'écran

```
#include <iostream.h>

int main()
{
    std::cout << `` Bonjour `` << std::endl
    return 0;
}
```

0.2 Type de langages de programmation

- Liste de quelques langages couramment utilisés

Langages	Type	Niveau
<i>Python</i>	Interprété	Haut
<i>C++</i>	Compilé	Haut
<i>C</i>	Compilé	Bas/haut
<i>Java</i>	Compilé	Haut
<i>HTML/CSS</i>	Interprété	
<i>PHP</i>	Interprété	Haut
<i>Javascript</i>	Interprété	Haut
<i>SQL</i>	Interprété	Haut

0.3 Algorithmie et programmation

- **Quel lien entre Algorithmie et langage de programmation ?**
 - ☞ L'algorithmie sert à résoudre les problèmes de monde des vivants grâce à la logique et surtout de mettre en place un système d'automatisation
 - ☞ L'algorithme est universel pour quasiment tous les langages de programmation
 - ☞ Un algorithme peut être codé en C, C++, qu'en Python et d'autres langage
 - ☞ Le langage de programmation sert entre autre à traduire l'algorithme dans le strict respect dudit langage
 - ☞ ***La connaissance de l'algorithmie est primordial et vital pour tout programmeur***

0.4 A propos de la programmation

▪ Programmer, c'est dur ?



Il faut que je sois
un ***super-méga***
matheux
Ouais, ouais,

☞ Réponses:

➤ **Non, non et non**

- un super-niveau en maths n'est pas nécessaire
- La connaissance en maths dépend du type de logiciels (cryptage, jeux 3D, ...)
- Tout ce dont vous aurez besoin en Maths
 - Savoir effectuer les calculs de base (addition, multiplication, ...)
 - **Notion de logique (Algorithmie)**

0.4 A propos de la programmation

▪ Programmer, c'est dur ?

☞ Les qualités suivantes sont nécessaires pour être un bon programmeur

Patience

- Le programme ne marche jamais du premier cours
- Il faut savoir persévérer

Sens de la logique

- Pas besoin d'être fort en Maths mais besoin de réfléchir
- Trouver l'algorithmie répondant à un problème donné

Sérénité

- Non, on ne tape pas son ordinateur avec un marteau
- Ce n'est pas ça qui résoudra le programme
- Un peu de sérénité

1. Notion de base en langage C

1.1 Présentation du langage C

▪ Historique

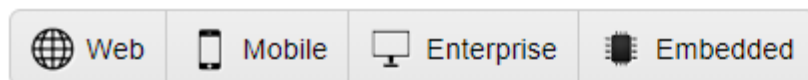
Date	Description
1978	☞ Création du langage C dans les laboratoires Bell par les chercheurs B. Kernighan et D. Ritchie
1980	☞ Invention du langage C++ par Bjarne Stroustrup (informaticien danois) <ul style="list-style-type: none">➤ Extension du langage C par la programmation Objet
1983	☞ Création du langage Objective - C par Brad Crox <ul style="list-style-type: none">➤ Extension du langage C, pour la programmation objet➤ Très utilisé par Apple pour la conception des interfaces graphiques de MacOS
1990	☞ Normalisation du Langage C par la norme ANSI/ISO <ul style="list-style-type: none">➤ Normalisé par la norme ANSI/ISO (C99 ou Ansi C)
1998	☞ Normalisation du Langage C++ par l'ANSI/ISO























1.1 Présentation du langage C

■ Pourquoi choisir le langage C ?

☞ Sa popularité

- Un des langages de programmation les plus utilisés
- Il possède une communauté très importante et de nombreux tutoriels et documentations
- Il existe beaucoup de programmes et de bibliothèques développés en/et pour le C



Language Rank	Types	Spectrum Ranking	Spectrum Ranking
1. Python	  	100.0	100.0
2. C++	  	98.4	99.7
3. C	  	98.2	99.4
4. Java	  	97.5	97.3
5. C#	  	89.8	88.7
6. PHP		85.4	88.7
7. R		83.3	86.0
8. JavaScript	 	82.8	81.9
9. Go	 	76.7	76.8
10. Assembly		74.5	76.0

1.1 Présentation du langage C

■ Pourquoi choisir le langage C ?

👉 Sa rapidité

- Connu pour être le langage très rapide
 - Langage de choix pour tout programme où la vitesse d'exécution est cruciale
- Langage à cheval entre les langages haut niveau et bas niveau

👉 Sa légèreté

- Très utile pour les programmes embarqués où la mémoire disponible est faible

👉 Sa portabilité

- Un programme développé en C marche théoriquement sur n'importe quelle plate-forme
- Le C a été conçu pour la programmation système (pilotes, systèmes d'exploitation, matériel embarqué, etc)

1.1 Présentation du langage C

▪ Quels sont les outils nécessaires au programmeur ?

☞ 3 éléments minimums sont nécessaires :

☞ **Un éditeur de texte :**

- Utile pour écrire le code source du programme
- Exemple : bloc-notes, notepad++, ...

☞ **Un compilateur:**

- Transforme (ou compile) le code source en langage binaire

☞ **Un débogueur:**

- Très utile pour traquer les erreurs dans votre programme

☞ Il existe des environnements intégrant les trois éléments à la fois. On les appelle des **IDE (Environnement de Développement Intégré)**

1.1 Présentation du langage C

▪ Les environnements de développement intégré (EDI ou IDE)

☞ Visual C++

- IDE propriétaire de Microsoft
- Fonctionne uniquement sous Windows
- Version payante
- Il existe des versions gratuites : Visual C++ express (très complet)



☞ Eclipse CDT (C++ Development Tools)

- ☞ IDE gratuit et complet offrant des fonctionnalités facilitant la programmation
- ☞ Multiplateforme (Windows, Linux et Mac)



1.1 Présentation du langage C

▪ Les environnements de développement intégré (EDI ou IDE)

👉 Dev – C++

- IDE gratuit



👉 Code::Blocks

- IDE gratuit et disponible pour la plupart des systèmes d'exploitation
- Très simple d'utilisation et possède une complétion très riche



👉 Nous allons utiliser plutôt **Code::Blocks**

1.1 Présentation du langage C

- **Les compilateurs du C++**

- ☞ **Visual C++ build (Microsoft)**

- ☞ **Borland C++ builder**

- ☞ Gratuit

- ☞ Utilisable sous Windows et linux

- ☞ **GCC (GNU Compilation Collection)**

- Excellent compilateur, gratuit et le plus utilisé

- Multiplateforme

- Compilateur multi-langage (C, C++, Java, ...)

- ☞ Nous allons utiliser plutôt le **GCC**

- ☞ **Code::blocks** intègre tous les compilateurs

1.2. IDE Code::Blocks

- Le téléchargement de cet IDE se fait sur le site officiel de **code::Blocks**

« <http://www.codeblocks.org/downloads/binaries> »

☞ Les versions de l'IDE sont proposées en fonction du système d'exploitation (Windows, Linux, MacOS)

☞ **Exemple** : version disponible pour Windows



Windows XP / Vista / 7 / 8.x / 10:

File	Date	Download from
codeblocks-17.12-setup.exe	30 Dec 2017	Sourceforge.net
codeblocks-17.12-setup-nonadmin.exe	30 Dec 2017	Sourceforge.net
codeblocks-17.12-nosetup.zip	30 Dec 2017	Sourceforge.net
codeblocks-17.12mingw-setup.exe	30 Dec 2017	Sourceforge.net
codeblocks-17.12mingw-nosetup.zip	30 Dec 2017	Sourceforge.net
codeblocks-17.12mingw_fortran-setup.exe	30 Dec 2017	Sourceforge.net

1.2. IDE Code::Blocks

- Le téléchargement de cet IDE se fait sur le site officiel de **code::Blocks**

« <http://www.codeblocks.org/downloads/binaries> »

☞ Pour télécharger la version appropriée pour son ordinateur, il faut:

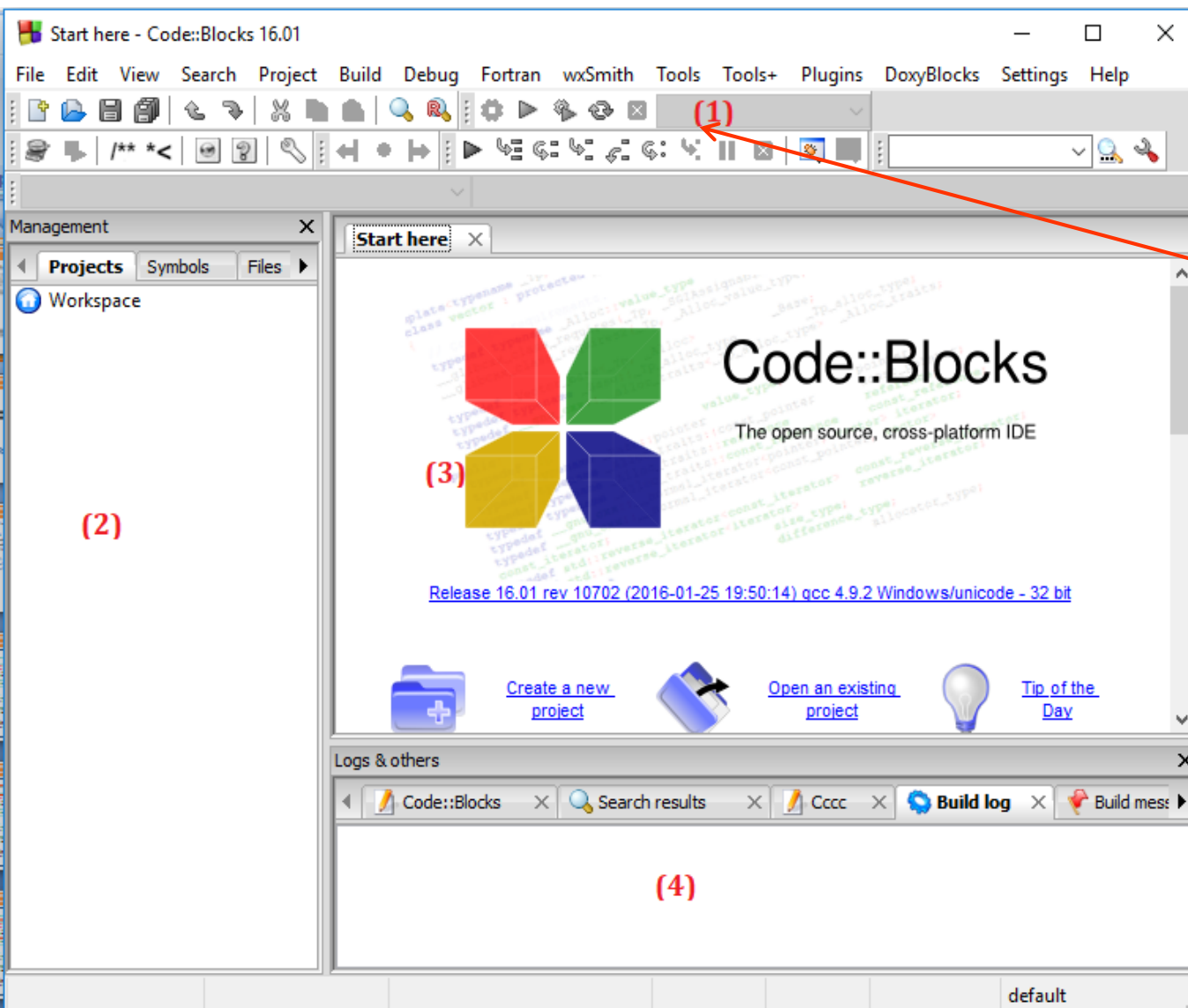
➤ Connaitre le *système d'exploitation* (Windows, Linux, MacOS)

➤ Ses *caractéristiques* (32 ou 64 bits)

☞ Exemple : Pour un pc Windows 10 à 64bits, il faut télécharger la version

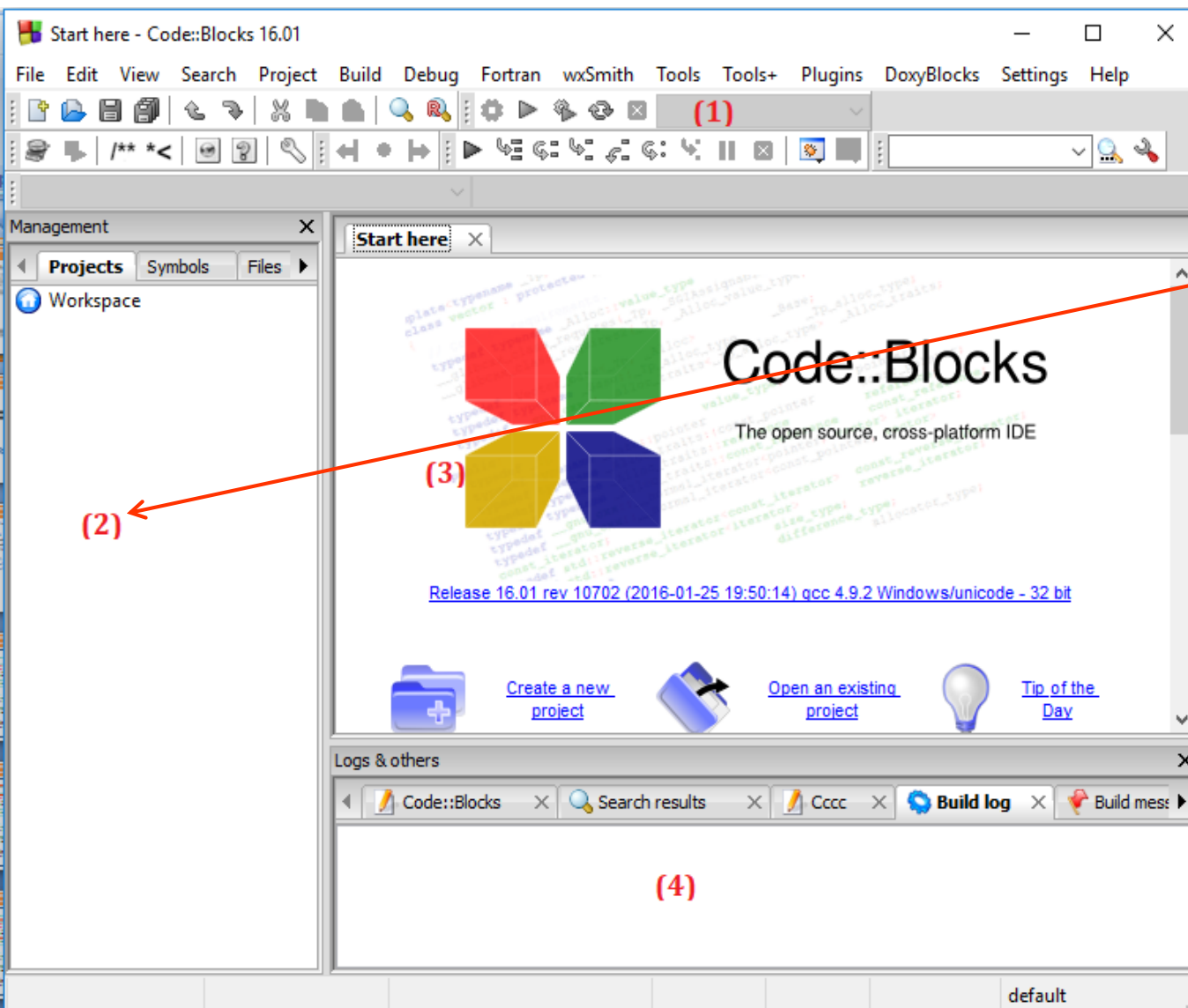
« *codeblocks-17.12mingw-setup.exe* »

1.2. IDE Code::Blocks - Présentation



- **barre d'outils** : elle comprend de nombreux boutons
- mais seuls quelques-uns nous seront régulièrement utiles.

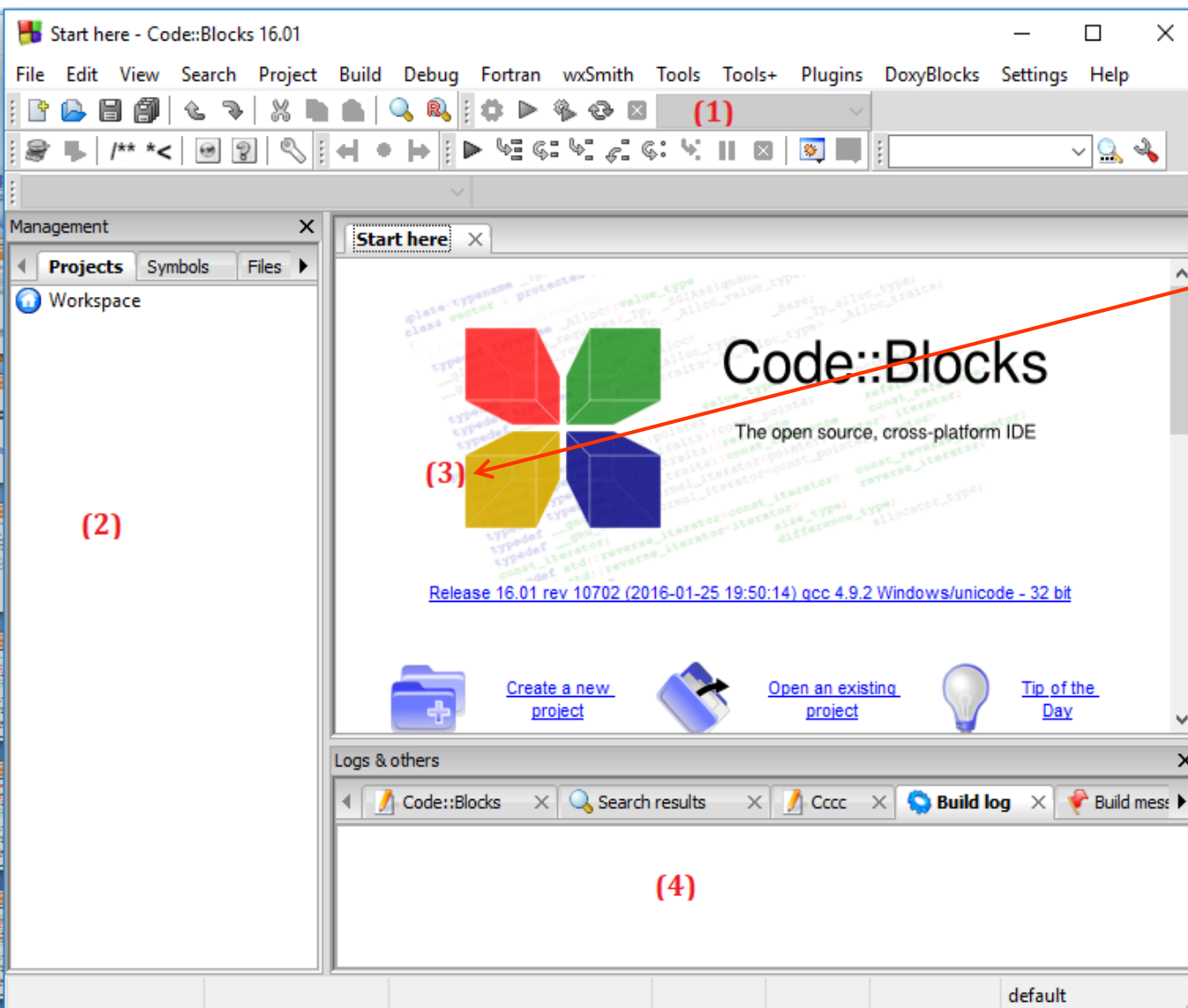
1.2. IDE Code::Blocks - Présentation



- **Liste des fichiers sources du projet**

- dans cette zone apparaitront tous les fichiers sources de votre programme

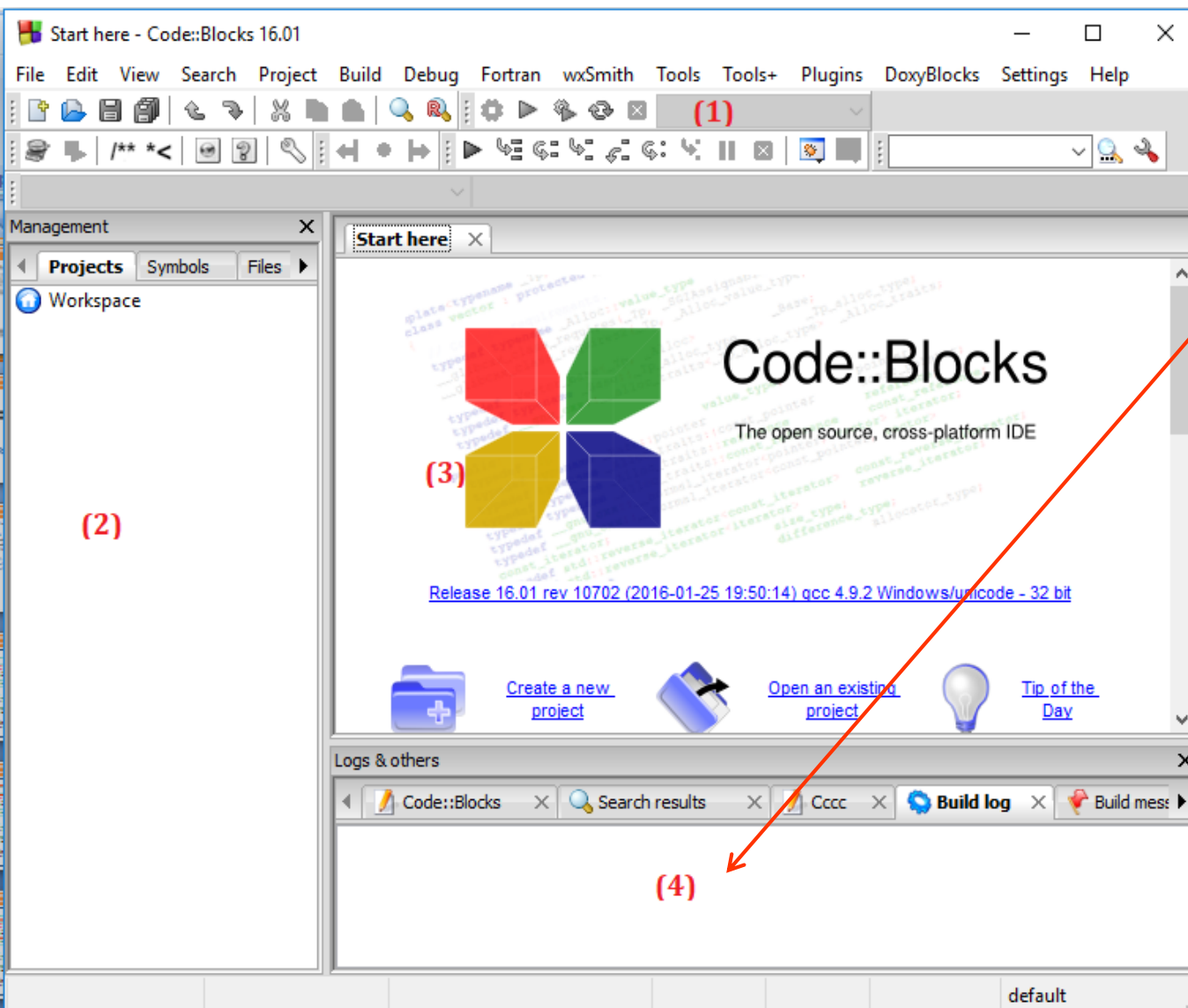
1.2. IDE Code::Blocks - Présentation



▪ **Zone principale**

▪ C'est à cet endroit que le code en langage C sera écrit

1.2. IDE Code::Blocks - Présentation



■ Zone de notification

c'est ici que vous verrez les *erreurs de compilation* s'afficher si votre code comporte des erreurs.

Cela arrive régulièrement!

1.2. IDE Code::Blocks - Présentation

■ Barre d'outils

☞ 4 icônes particuliers seront très utiles 

☞ **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, 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.

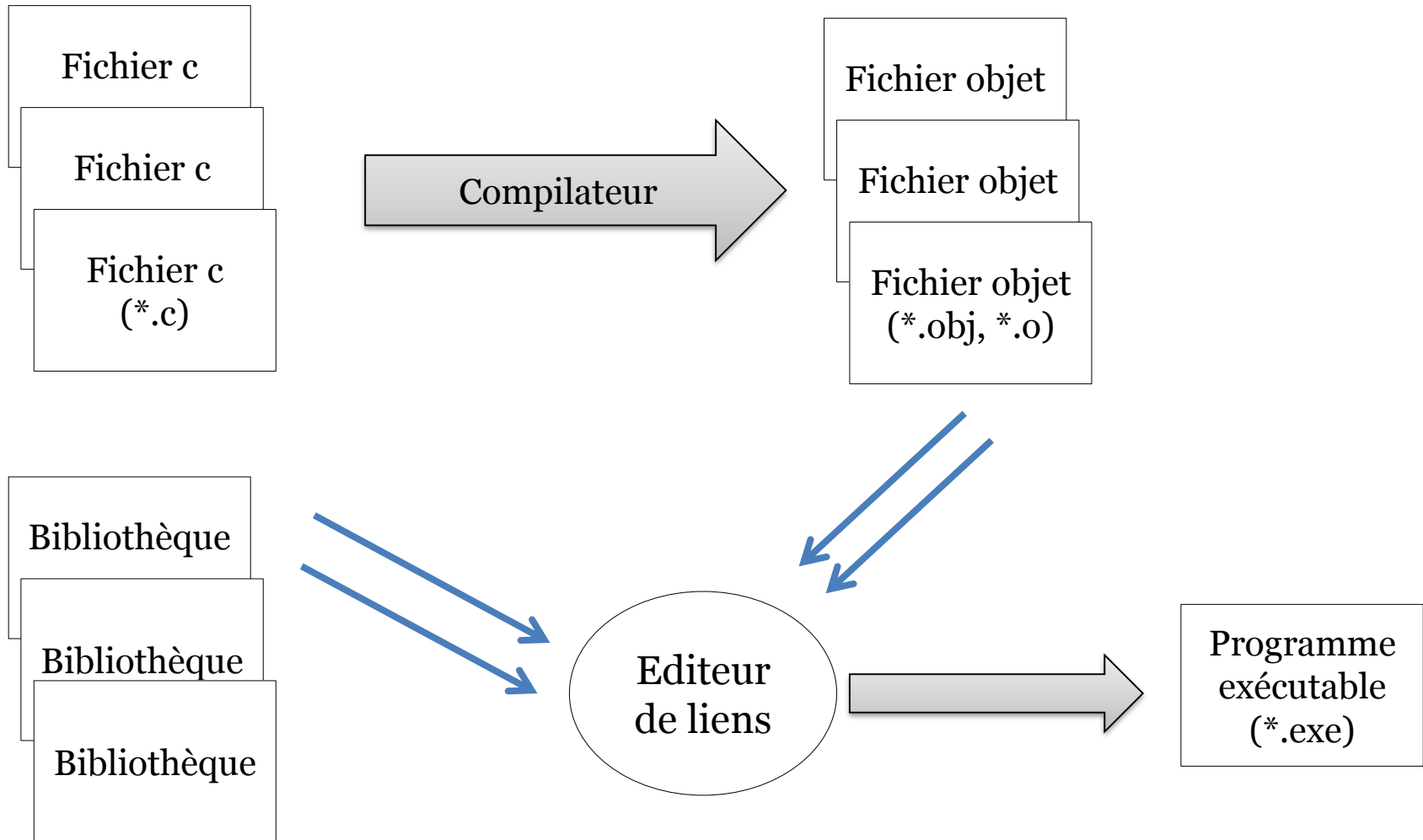
☞ **compiler & exécuter** : combinaison de la compilation et de l'exécution

- En cas d'erreurs pendant la compilation, le programme ne sera pas exécuté.
- A la place, vous aurez droit à une belle liste d'erreurs à corriger 😊

☞ **Tout reconstruire**: compilation prenant en compte les modifications récentes effectuées dans le code

1.2. IDE Code::Blocks - Présentation

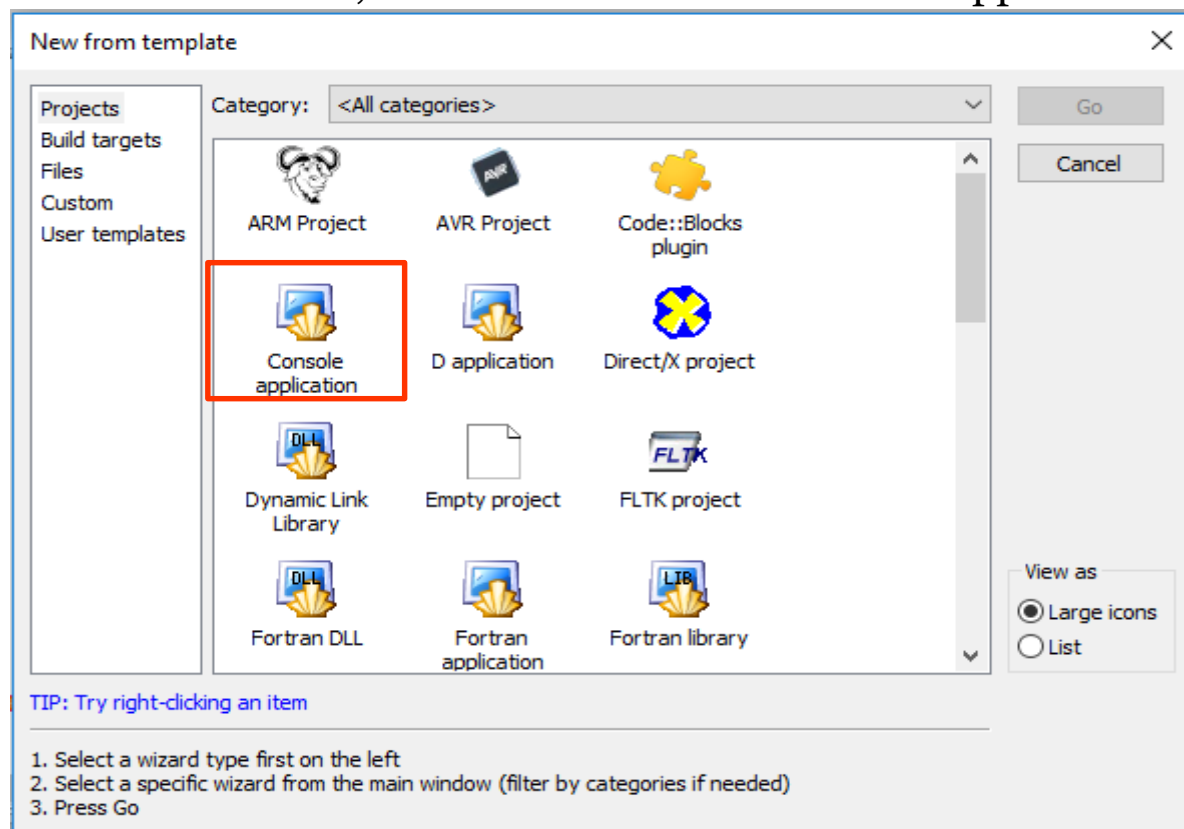
▪ A propos de la compilation



1.2. IDE Code::Blocks - Présentation

■ Création d'un projet

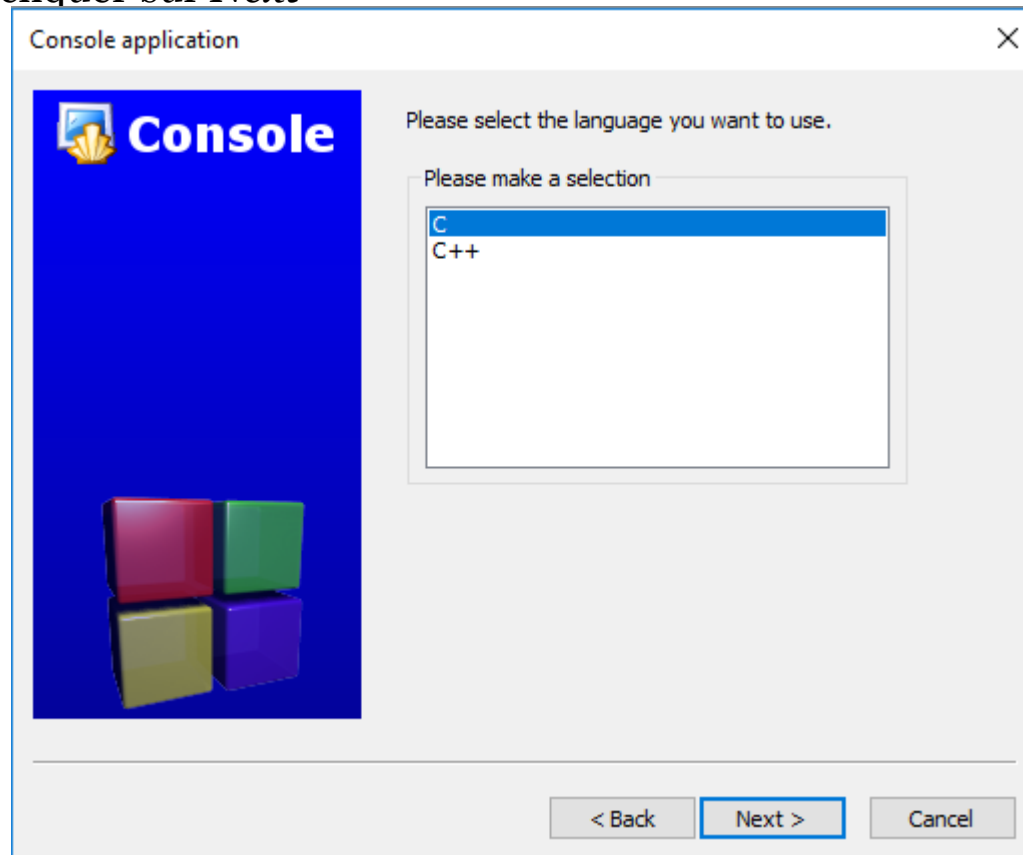
- ☞ Pour créer un nouveau projet en langage C : aller dans le menu **File / New/Project**
- ☞ Dans la fenêtre qui s'ouvrira, on sélectionnera le type d'application et on cliquera sur « go »
 - Dans notre cas, on s'intéressera d'abord à une application de type **console**



1.2. IDE Code::Blocks - Présentation

■ Création d'un projet

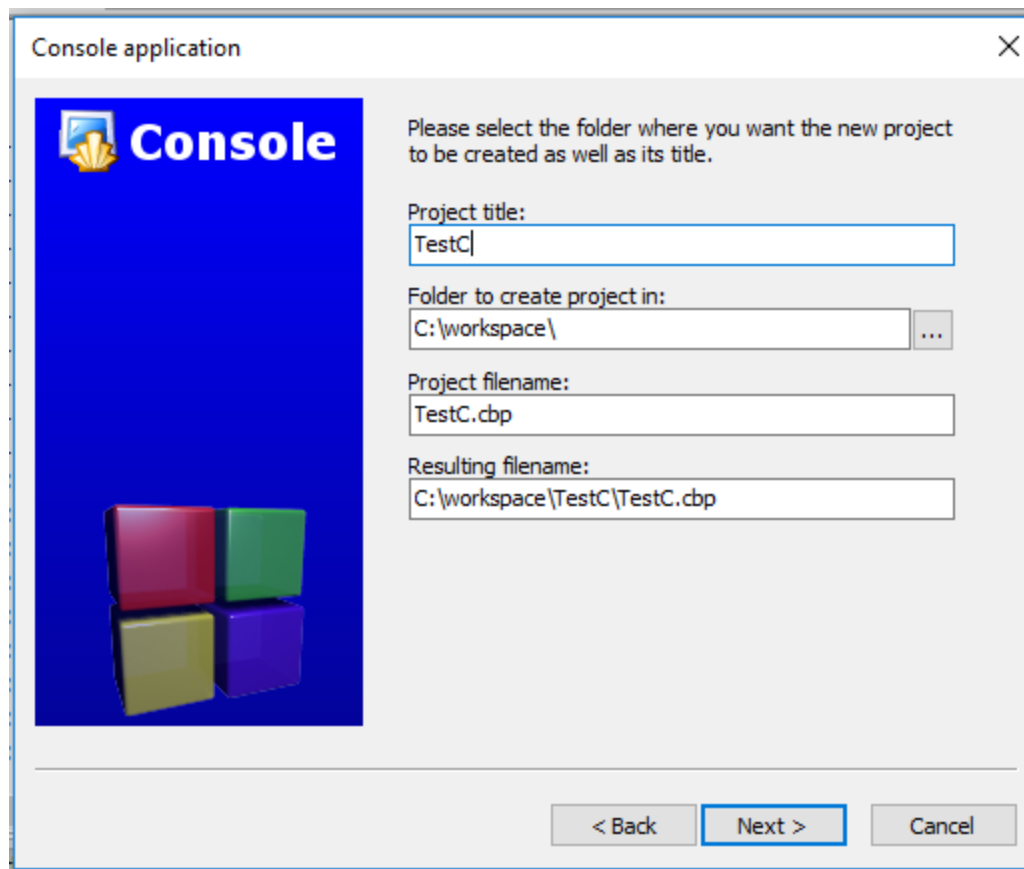
- **Cliquez** sur Go pour créer le projet. Un assistant s'ouvre. Faites *Next*, cette première page ne servant à rien. On vous demande ensuite si vous allez faire du C ou du C++ : choisir « C » et cliquer sur *Next*



1.2. IDE Code::Blocks - Présentation

■ Création d'un projet

- Préciser le nom du projet et le répertoire dans lequel sera stocké les fichiers sources du projet.

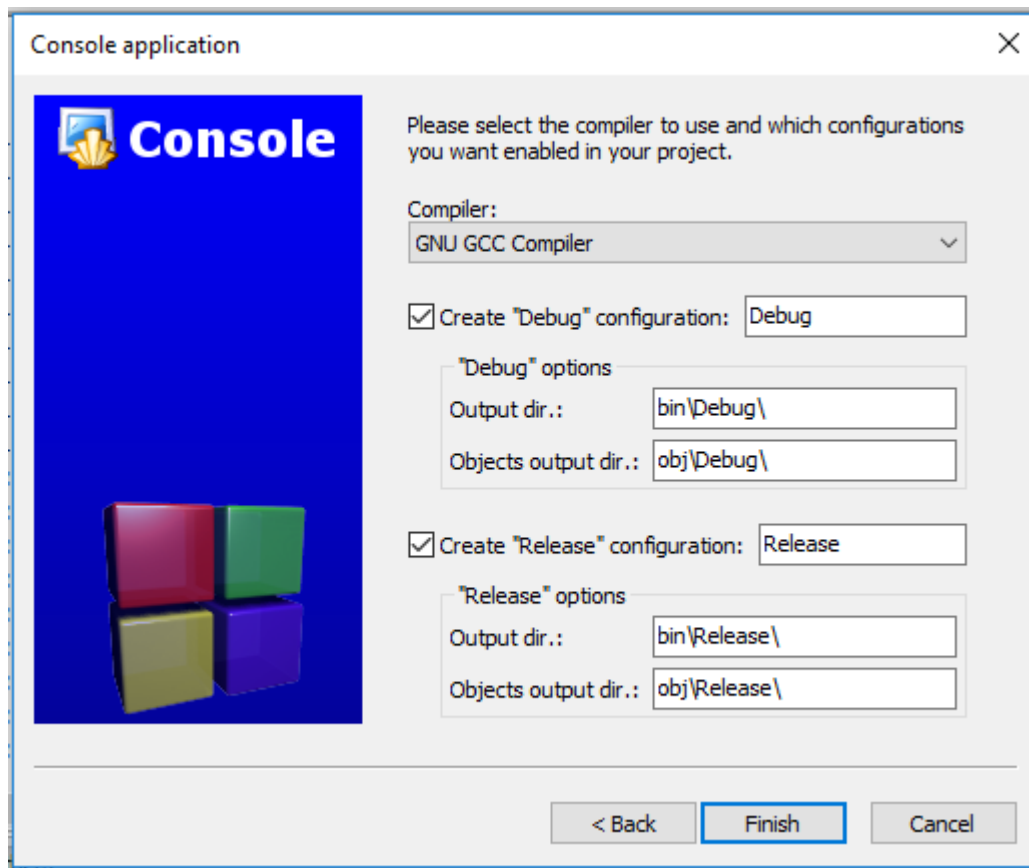


- Cliquer le bouton « *Next* »

1.2. IDE Code::Blocks - Présentation

■ Création d'un projet

- choisir la façon dont le programme devra être compilé.
- Laisser les options par défaut (on y reviendra en TP)



- Cliquer le bouton « *Finish* »

1.2. IDE Code::Blocks - Présentation

■ Type d'application

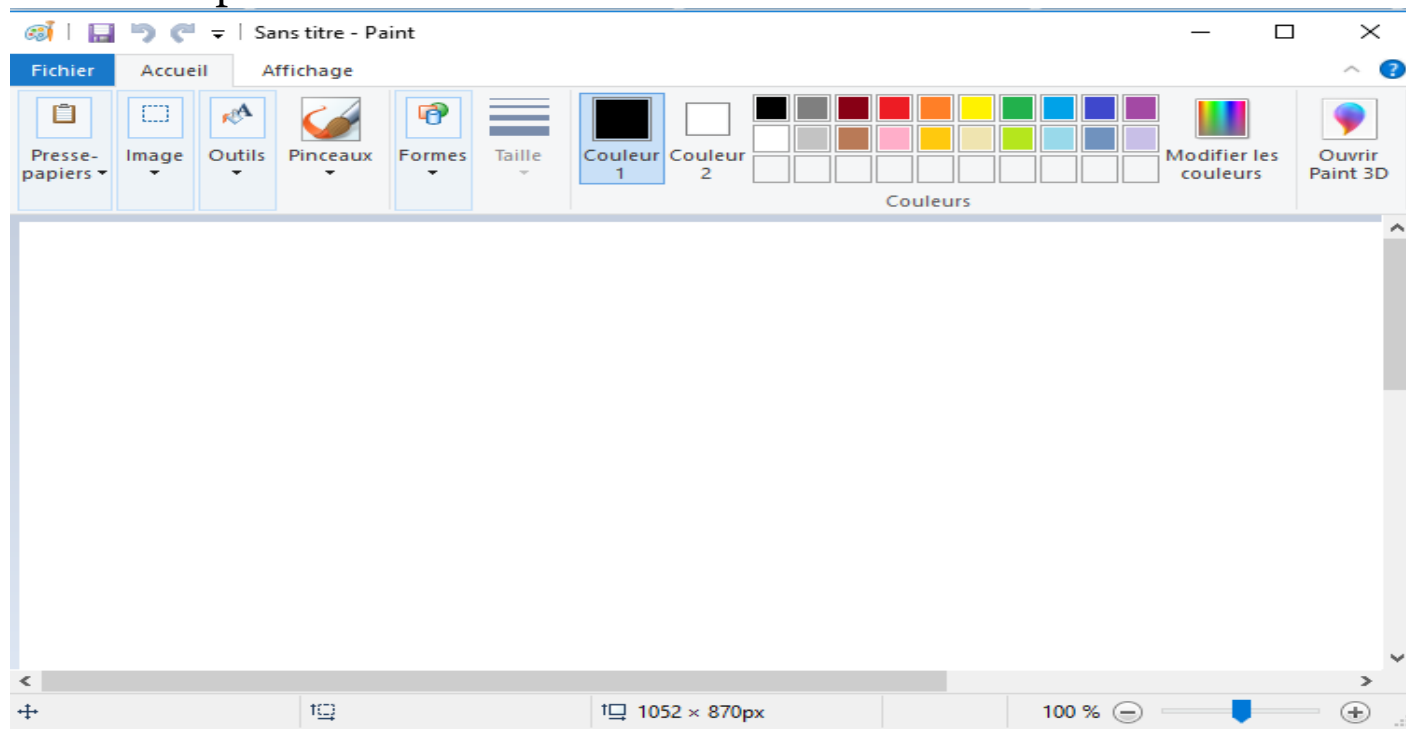
☞ Il existe 3 types d'applications :

➤ *Applications avec interface graphique*

➤ Il s'agit de programme dont la compilation aboutit à une interface graphique

➤ Exemple : Paint (de Windows), MS Word, PowerPoint, ...

➤ La plus utilisée



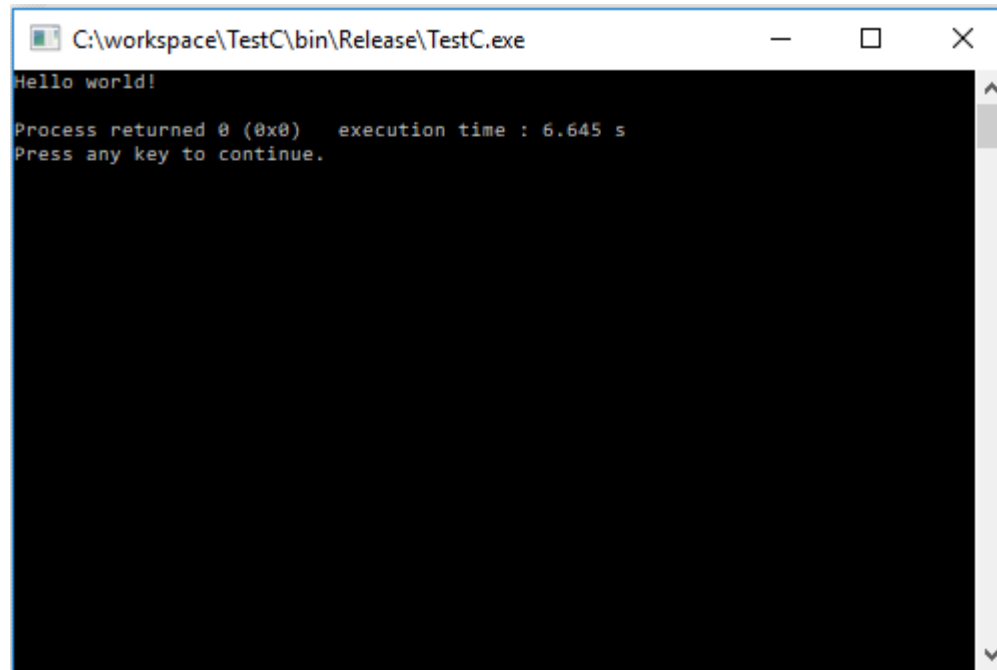
1.2. IDE Code::Blocks - Présentation

■ Type d'application

☞ Il existe 3 types d'applications :

➤ *Applications en console*

- Il s'agit de programme dont la compilation aboutit à une fenêtre similaire au MS DOS de Windows (fenêtre noir)



```
C:\workspace\TestC\bin\Release\TestC.exe
Hello world!
Process returned 0 (0x0)   execution time : 6.645 s
Press any key to continue.
```

1.2. IDE Code::Blocks - Présentation

■ Type d'application

☞ Il existe 3 types d'applications :

➤ *Applications de type Librairie*

➤ Il s'agit de programme générant une librairie

➤ dans ce cas, ce n'est pas une application visuelle mais utilisable dans un autre programme pour exécuter une tâche particulière

➤ Cette librairie peut être de type « dynamique » (extension .dll) ou statique (.lib)

1.3. Exemple de code

- Le code ci-dessous correspond au code généré lors de la création du projet
 - Code minimaliste (avec ajout de commentaires)

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

/* ce code permet d'afficher le
message « Hello world » à l'écran */
int main()
{
    //fonction d'affichage du message
    printf("Hello world!\n");

    return 0; // valeur de retour de
la fonction main
}
```

1.3. Exemple de code

- Le code ci-dessous correspond au code généré lors de la création du projet
 - Code minimaliste

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

/* ce code permet d'afficher le
message « Hello world » à l'écran */
int main()
{
    //fonction d'affichage du message
    printf("Hello world!\n");

    return 0; // valeur de retour de
la fonction main
}
```

- **Directives du préprocesseur**

- Identifiable par le dièse « # »
- « **include** » : inclure en français
- Ces lignes demandent d'inclure des fichiers au projet
- « **stdio.h** » et « **stdlib.h** » sont des fichiers déjà disponible dans le standard C
- ces fichiers contiennent du code tout prêt permettant d'afficher du texte à l'écran.

1.3. Exemple de code

- Le code ci-dessous correspond au code généré lors de la création du projet
 - Code minimaliste

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

/* ce code permet d'afficher le
message « Hello world » à l'écran */
int main()
{
    //fonction d'affichage du message
    printf("Hello world!\n");

    return 0; // valeur de retour de
la fonction main
}
```

- **Commentaires multi lignes**

`/* commentaire`

`...
*/`

- **Commentaires sur une ligne**

`// commentaire`

1.3. Exemple de code

- Le code ci-dessous correspond au code généré lors de la création du projet
 - Code minimaliste

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

/* ce code permet d'afficher le
message « Hello world » à l'écran */
int main()
{
    //fonction d'affichage du message
    printf("Hello world!\n");

    return 0; // valeur de retour de
la fonction main
}
```

▪ Instruction

- On donne des ordres à machine
- Chaque instruction se termine par un point virgule « ; »

1.3. Exemple de code

- Le code ci-dessous correspond au code généré lors de la création du projet
 - Code minimaliste

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

/* ce code permet d'afficher le
message « Hello world » à l'écran */
int main()
{
    //fonction d'affichage du message
    printf("Hello world!\n");

    return 0; // valeur de retour de
la fonction main
}
```

▪ Utilisation de fonctions

- « **main** » : fonction principale du code
 - Sans cette fonction, impossible d'exécuter le programme
- Appel de la fonction « printf », existant dans le fichier « stdio.h »

1.3. Exemple de code

- Le code ci-dessous correspond au code généré lors de la création du projet
 - Code minimaliste

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

/* ce code permet d'afficher le
message « Hello world » à l'écran */
int main()
{
    //fonction d'affichage du message
    printf("Hello world!\n");

    return 0; // valeur de retour de
la fonction main
}
```

▪ valeur de retour de fonctions

- Indique que la fonction « main » doit retourner la valeur 0 à la fin de l'exécution du code

1.3. Exemple de code

- Le code ci-dessous correspond au code généré lors de la création du projet
 - Code minimaliste

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

/* ce code permet d'afficher le
message « Hello world » à l'écran */
int main()
{
    //fonction d'affichage du message
    printf("Hello world!\n");

    return 0; // valeur de retour de
la fonction main
}
```

▪ valeur de retour de fonctions

- Indique que la fonction « main » doit retourner la valeur 0 à la fin de l'exécution du code

1.3. Exemple de code

- **Exercice** : dans le votre code, ajouter les commentaires et afficher l'information suivante dans la console :
« *Hello, Je veux être un élite en informatique.
Cependant, pour y arriver, je dois bien suivre les cours et assouvir ma curiosité.
Je dois travailler sans relâche (en dormant bien).
Car c'est en forgeant que l'on devient forgeron!
Je suis prêt et toi? »*

1.4. Les variables

■ Généralités

- ☞ **Les variables** servent à stocker des informations (des nombres, des chaînes de caractères, ...) dans la mémoire
- ☞ La notion de variables est intrinsèquement liée à la mémoire de l'ordinateur

☞ **Questions :**

- ☞ Comment fonctionne la mémoire d'un ordinateur
- ☞ Combien un ordinateur possède t-il de mémoires ?

1.4. Les variables

▪ Type de mémoire d'un ordinateur

☞ Il en existe 4 types :

☞ *les registres* : une mémoire ultra-rapide située directement dans le processeur

☞ *La mémoire cache* : elle fait le lien entre les registres et la mémoire vive

☞ *La mémoire vive* : c'est la mémoire avec laquelle nous allons travailler le plus souvent

☞ *Le disque dur* : que vous connaissez sûrement, c'est là qu'on enregistre les fichiers

1.4. Les variables

▪ Type de mémoire d'un ordinateur

- ☞ En programmation (haut niveau), nous allons beaucoup travailler avec :
 - ☞ La *mémoire vive* pour l'exécution de nos programme
 - ☞ Le *disque dur* pour le stockage des données
- Quant aux *registre et mémoire cache*, ils sont plutôt utilisés par les langages bas niveau comme *l'assembleur*
 - Ce sont des mémoires temporaires
 - Elles se vident lorsqu'on éteint l'ordinateur

1.4. Les variables

■ Mémoire vive d'un ordinateur

- La mémoire vive d'un ordinateur se trouve dans la partie encadrée en rouge sur la figure (contenu de l'unité centrale d'un ordinateur)



- La mémoire vive est aussi appelé « *RAM* »

1.4. Les variables

■ Mémoire vive d'un ordinateur

➤ Schéma

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

- La mémoire vive contient 2 éléments (en colonne)
- **les adresses** : une adresse est un nombre qui permet à l'ordinateur de se repérer dans la mémoire vive
 - Commence à l'adresse 0 (au tout début de la mémoire)
 - Plus vous avez de mémoire vive, plus il y a d'adresses,
- **Les valeurs** : à chaque adresse, on peut stocker une valeur (un nombre) : votre ordinateur stocke dans la mémoire vive des nombres pour pouvoir s'en souvenir par la suite.

■ **On ne peut stocker qu'un nombre par adresse !**

1.4. Les variables

■ Mémoire vive d'un ordinateur

➤ *Principe de fonctionnement de l'ordinateur*

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

- 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 35).
- Plus tard, lorsqu'il veut savoir à nouveau quel est ce nombre, il va chercher à la « case » mémoire n° 35 ce qu'il y a, et il trouve la valeur... 5 !

1.4. Les variables

▪ Déclaration d'une variable

☞ 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

☞ En C, chaque variable doit porter un nom

☞ Exemple : une variable qui retient la somme des entiers naturels peut être appelé : « *somme* »

1.4. Les variables

▪ Déclaration d'une variable

☞ **Remarque :**

☞ *Le langage C est sensible à la casse (majuscule/minuscule)*

➤ *Exemple : somme #Somme*

☞ *Les espaces sont interdits dans les noms de variables*

☞ A la place, on peut utiliser le caractère underscore « _ »

➤ *Exemple : variable indiquant le nombre de vie d'un joueur :
nombre_de_vie ou nombreDeVie*

☞ *Vous n'avez pas le droit d'utiliser des accents (é, à,ê, etc.).*

☞ Exemple : on ne peut pas écrire « *précaution* » mais « *precaution* »

1.4. Les variables

▪ Déclaration d'une variable

☞ **Remarque :**

☞ *Le langage C est sensible à la casse (majuscule/minuscule)*

➤ *Exemple : somme #Somme*

☞ *Les espaces sont interdits dans les noms de variables*

☞ A la place, on peut utiliser le caractère underscore « _ »

➤ *Exemple : variable indiquant le nombre de vie d'un joueur :
nombre_de_vie ou nombreDeVie*

☞ *Vous n'avez pas le droit d'utiliser des accents (é, à,ê, etc.).*

☞ Exemple : on ne peut pas écrire « *précaution* » mais « *precaution* »

1.4. Les variables

▪ Les types de variables

- ☞ Pour déclarer une variable, l'ordinateur a besoin que l'on spécifie son type
- ☞ En langage C, il existe trois grands types de variables prédéfinie
 - *Les entiers*
 - *Les réels*
 - *Les caractères*

1.4. Les variables

▪ Les entiers

- ☞ Type **int, short, long**
 - Entiers signés, taille dépendante de la machine (32 bits / 64 bits)
- ☞ Pour des entiers non signés, ajouter le mot clé « **unsigned** »
 - **unsigned int, unsigned short, unsigned long**
- ☞ Constantes : -1, 1...

▪ Les réels

- ☞ Type **float** : réel simple précision (4 octets)
- ☞ Type **double** : réel double précision (8 octets)
- ☞ Type **long double** : précision maximale, celle du FPU (10 octets)
- ☞ Constantes : 1.0, -1.0...

1.4. Les variables

▪ Les caractères

- ➡ Type **char, unsigned char** (1 octet)
- ➡ Aussi considéré comme un entier signé / non signé
- ➡ Constantes : 'a', 'b', '#' ou 1, 2, -1, -2...

1.4. Les variables

■ Déclaration d'une variable

☞ *Syntaxe*

```
Type identifiant;
```

➤ **Attention** : variable non initialisée

☞ Syntaxe avec initialisation à la déclaration

```
Type identifiant = valeur;
```

■ **Attention**

☞ Le langage C n'initialise pas les variables avec des valeurs par défaut

☞ Une variable non initialisée possède une valeur indéterminée

➤ Source de nombreux « bugs »

☞ **Règle : toujours initialiser les variables**

1.4. Les variables

■ Déclaration d'une variable

☞ *Exemple*

```
int min; // Attention : non initialisée
```

```
// Variable de type réel
```

```
double pi = 3.1415926535;
```

```
double pi(3.1415926535) ;
```

```
// Variable de type caractère
```

```
char c = 'a' ;
```

```
char c('a') ;
```

```
// Déclaration de plusieurs variables de même type
```

```
int min=0, max=100 ;
```

```
int min(0), max(100) ;
```

1.4. Les variables

▪ Affichage du contenu d'une variable

- ➡ Nous avons vu que la fonction « printf » permet d'afficher du texte à l'écran
- ➡ Pour afficher le contenu d'une variable, on utilise une commande spécifique :
- ➡ Exemple : code permettant d'afficher le nombre de vie d'un joueur :

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

- ➡ Ce « symbole spécial » est 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 entier « int »
- ➡ Il existe plusieurs autres possibilités, mais pour des raisons de simplicité on va se contenter de retenir celles-ci

1.4. Les variables

- **Affichage du contenu d'une variable**

Format	Type attendu
« %d »	<i>int</i>
« %ld »	<i>long</i>
« %f »	<i>Float, double</i>

1.4. Les variables

▪ Affichage du contenu d'une variable

☞ Exemple :

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    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;
}
```

1.4. Les variables

▪ Affichage du contenu d'une variable

- ☞ Il est tout à fait possible d'afficher les valeurs de plusieurs variables dans un seul ***printf***

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int nombreDeVies = 5, niveau = 1;
    printf("Vous avez %d vies et vous etes au niveau n° %d\n", nombreDeVies,
    niveau);
}
```


1.4. Les variables

▪ Lecture d'information sur le clavier

- ☞ Il est tout à fait possible qu'un utilisateur saisisse une information dans la console
- ☞ Une fois l'information saisie, on va le récupérer et le stocker dans une variable
- ☞ On utilisera la fonction « **scanf** »

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

☞ *Exemple*

```
#include <stdio.h>  
#include <stdlib.h>  
int main()  
{  
    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);  
}
```

1.5. Les Opérations de base

- Dans le langage C, il est tout à fait possible d'effectuer les opérations élémentaires

☞ **Affectation: =**

- On affecte une valeur à une variable
- *Exemple :*

```
int x = 5;
```

☞ **Addition: +**

- On effectue une addition entre 2 nombres
- Le résultat doit être sauvegarder une variable

```
int resultat = 0;  
resultat = 5 + 3;  
printf("5 + 3 = %d", resultat);
```

☞ Il en est de même pour la **soustraction (-)**

1.5. Les Opérations de base

- Dans le langage C, il est tout à fait possible d'effectuer les opérations élémentaires

☞ **Multiplication: ***

➤ Exemple :

```
double x = 5.0;  
int y = 40;  
double produit = x*y;  
  
printf("%f * %d = %f", x, y, produit);
```

☞ **Division: **

➤ **Attention** : une division entre 2 entiers est une division entière

```
double division= 3/2;  
printf("3 / 2 = %d", division);
```

➤ Pour une division avec décimal, il faut qu'au moins un des nombres soit un réel

```
double division= 3.0/2;  
printf("3.0 / 2 = %d", division);
```

1.5. Les Opérations de base

- Dans le langage C, il est tout à fait possible d'effectuer les opérations élémentaires

- ☞ **Modulo: %**

- Il s'agit du reste de la division entière
 - Exemple :

```
double x = 5.0;  
int y = 40;  
int reste = y % x;  
  
printf("%d modulo %f = %d", y, x, reste);
```

- **Exercice** : Ecrire un programme demandant à l'utilisateur de saisir 2 nombres et d'effectuer la somme, le produit et le modulo de ces nombres. Enfin il affichera les résultats dans la console

1.5. Les Opérations de base

■ Incrémentation : ++

☞ Elle consiste à augmenter de 1, la valeur d'une variable

☞ *Exemple*

```
int somme = 2;  
printf("Avant : somme = %d", somme);  
  
//incrément de la valeur de somme  
somme++;  
printf("Après: somme = %d", somme);
```

☞ **a++ ⇔ a = a + 1**

1.5. Les Opérations de base

▪ Décrémentation : --

☞ Elle consiste à réduire de 1, la valeur d'une variable

☞ *Exemple*

```
int somme = 2;  
printf("Avant : somme = %d", somme);  
  
//incrémentation de la valeur de somme  
somme--;  
printf("Après: somme = %d", somme);
```

☞ **a-- ⇔ a = a - 1**

1.5. Les Opérations de base

■ Affectation avec opération

- ☞ Addition: **+=**, ex : $a += b \Leftrightarrow a = a + b$
- ☞ Soustraction: **-=**, ex : $a -= b \Leftrightarrow a = a - b$
- ☞ Multiplication: ***=**, ex : $a *= b \Leftrightarrow a = a * b$
- ☞ Division: **/=**, ex : $a /= b \Leftrightarrow a = a / b$
- ☞ Modulo: **%=**, ex : $a %= b \Leftrightarrow a = a \% b$

■ Opérateurs de comparaison

- ☞ **<** (inférieur) , **>** (supérieur) , **<=** (inférieur ou égal) , **>=** (supérieur ou égal) , **==** (égal) , **!=** (différent de)

■ Opérateurs logiques

- ☞ Négation: **!**
- ☞ ET logique: **&&**
- ☞ OU logique: **||**

Par exemple : $((x < 12) \&\& ((y > 0) || !(z > 4)))$

1.5. Les Opérations de base

■ Bibliothèque mathématique

- ☞ En C, il existe des bibliothèques « standard » contenant d'innombrables fonctions prêt à l'emploi
- ☞ **Intérêt** : Ces fonctions vous évitent en quelque sorte d'avoir à réinventer la roue à chaque nouveau programme.
- ☞ Nous avez déjà utilisé les fonctions « *printf* » et « *scanf* » de la bibliothèque « *stdio.h* »
- ☞ Il existe une bibliothèque contenant les fonctions mathématiques : « *math.h* »
- ☞ Pour l'utiliser, il suffit de l'inclure dans le code

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


1.5. Les Opérations de base

▪ Bibliothèque mathématique

☞ Quelques fonctions très utiles :

Fonction	Description
<i>fabs</i> (x)	Elle retourne la valeur absolue d'un nombre, c'est-à-dire $ x $
<i>ceil</i> (x)	Elle renvoie la valeur arrondi d'un nombre
<i>pow</i> (x, power)	Calcul la puissance d'un nombre
<i>sqrt</i> (x)	Calcul la racine carré d'un nombre
<i>Sin, cos, tan</i>	Calcul trigonométrique
<i>exp</i> (x)	Calcul l'exponentielle d'un nombre
<i>log</i> (x)	Calcul du logarithme népérien d'un nombre (que l'on note aussi « ln »).
<i>log10</i> (x)	calcule du logarithme base 10 d'un nombre.

1.6 Les conditions

- Les *conditions* permettent d'écrire dans le programme des règles comme

« *Si ceci arrive, alors fais cela* »

- *Exemple :*

On demande à l'utilisateur de saisir le nombre d'enfants. On demande de coder les règles suivantes:

- Si le nombre d'enfants est égal à 0 alors, on affiche à l'écran « *Eh bien alors, vous n'avez pas d'enfant ?* »
- Sinon si le nombre d'enfant est égal à 1, alors on affiche « *Alors, c'est pour quand le deuxième?* »
- Sinon si le nombre d'enfant est égal à 2, alors on affiche « *quels beaux enfants, vous avez là!* »
- Sinon, on affiche « *Bon, il faut arrêter de faire des gosses maintenant* »

1.6 Les conditions

▪ Syntaxe simple de la condition

*Si la variable vaut telle valeur
Alors fais ceci*

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

☞ Exemple

```
printf (" saisir le nombre de vos enfants!");  
int nbEnfants = 0;  
scanf("%d ", &nbEnfants);  
if (nbEnfants == 0 )  
{  
    printf (" Eh bien alors, vous n'avez pas d'enfant ?");  
}
```

1.6 Les conditions

- **Syntaxe de la condition avec « **sinon** »**

***Si** la variable vaut telle valeur **Alors** fais ceci
Sinon fais cela*

```
if(/*votre condition*/)
{
    // Instructions à exécuter si la condition vraie
}
else // sinon
{
    // Instructions si
    // condition fausse
};
```

1.6 Les conditions

- **Syntaxe simple de la condition avec « **sinon** »**

***Si** la variable vaut telle valeur **Alors** fais ceci
Sinon fais cela*

☞ Exemple

```
printf (" saisir le nombre de vos enfants!");  
int nbEnfants = 0;  
scanf("%d ", &nbEnfants);  
if (nbEnfants == 0 )  
{  
    printf (" Eh bien alors, vous n'avez pas d'enfant ?");  
}  
else  
{  
    printf (" Bon, il faut arreter de faire des gosses maintenant!");  
}
```

1.6 Les conditions

- **Syntaxe simple de la condition avec « **sinon si** »**

***Si** la variable vaut telle valeur **Alors** fais ceci
Sinon si la variable vaut ça **alors** fais ça,
Sinon fais cela*

```
if(/*votre condition*)  
{  
    // Instructions à exécuter si la condition vraie  
}  
else if(/*votre nouvelle condition*/) // sinon si  
{  
    // Instructions si la nouvelle condition fausse  
}  
else // sinon  
{  
    // Instructions si aucune condition n'est vérifiée  
};
```

1.6 Les conditions

■ Exemple

```
#include <iostream>
using namespace std;
int main()
{
    printf ( " saisir le nombre de vos enfants!");
    int nbEnfants = 0;
    scanf("%d ", &nbEnfants);
    if (nbEnfants == 0)
    {
        printf( "Eh bien alors, vous n'avez pas d'enfant ?" );
    }
    else if( nbEnfants == 1)
    {
        printf ( "Alors, c'est pour quand le deuxieme ?" );
    }
    else if(nbEnfants == 2)
    {
        printf("Quels beaux enfants vous avez la !" );
    }
    else
    {
        printf( "Bon, il faut arreter de faire des gosses maintenant !" );
    }

    printf ( "Fin du programme" );
    return 0;
}
```

1.6 Les conditions

- **Tableau de symbole à connaître par cœur**

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

- **Symbole de combinaison des conditions**

Symbole	Signification
&&	et
	ou
!	non

1.6 Les conditions

▪ Exemple

```
#include <iostream>
int main()
{
    printf ( " saisir le montant de votre argent de poche!");
    int argentDePoche= 0;
    scanf("%d ", & argentDePoche);

    printf ( " saisir votre age!");
    int age= 0;
    scanf("%d ", &age);

    if (age > 30 || argentDePoche > 100000)
    {
        printf( "Bienvenue chez PicsouBanque " );
    }
    else
    {
        printf("Hors de ma vue, miserable !");
    }
    return 0;
}
```

1.6 Les conditions

▪ Remarques

- ☞ Il est possible d'avoir des conditions « *if* », « *else* », « *else if* » qui s'imbriquent
- ☞ Pour tester une condition d'égalité, il faut obligatoirement utiliser la double égalité « **==** » et non simple égalité « **=** » (une affectation)
- ☞ Exercice :

1.6 Les conditions

▪ Exercice :

On souhaite écrire un programme permettant à l'utilisateur de saisir son année de naissance et de déterminer si l'année saisie est bissextile ou non. Il affiche à l'écran le résultat.

☞ La règle est la suivante:

- Si une année n'est pas multiple de 4, on s'arrête là, elle n'est pas bissextile.
- Si elle est multiple de 4, on regarde si elle est multiple de 100.
 - Si c'est le cas, on regarde si elle est multiple de 400.
 - Si c'est le cas, l'année est bissextile.
 - Sinon, elle n'est pas bissextile.
 - Sinon, elle est bissextile.

1.6 Les conditions

▪ La condition « **switch** »

- ☞ La condition « *if... else* » que l'on vient de voir est le type de condition le plus souvent utilisé
 - Il permet de gérer quasiment tous les cas de conditions
 - Cependant, il peut être très répétitif

☞ Exemple

```
#include <iostream>
using namespace std;
int main()
{
    printf (" saisir votre age!");
    int age= 0;
    scanf("%d ", &age);

    if (age > 30 || argentDePoche > 100000)
    {
        printf( "Bienvenue chez PicsouBanque " );
    }
    else
    {
        printf("Hors de ma vue, miserable !");
    }
    return 0;
}
```

1.6 Les conditions

■ La condition « **switch** »

☞ Exemple

```
#include <iostream>
using namespace std;
int main()
{
    printf (" saisir votre age!");
    int age= 0;
    scanf("%d ", &age);
    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");
    }
    return 0;
}
```

1.6 Les conditions

▪ La condition « **switch** »

☞ La condition « **switch** » permet d'écrire les conditions de façon plus limpide.

☞ *Attention* : Il ne s'applique que dans les cas où la variable à tester est un entier ou un caractère

☞ *Syntaxe*

```
switch(expression)
{
    case valeur1 :
        // instructions exécutées
        // si
        // expression == valeur1
        break ;
    case valeur2 :
        // instructions
        break ;
    default :
        // instructions
}
```

▪ L'expression doit être de type

- ☞ Caractère
- ☞ Entier
- ☞ Enuméré (mot clé « enum »)

▪ Le mot clé **break**

- ☞ Provoque la sortie du switch
- ☞ Si absent, l'exécution continue séquentiellement

▪ Le bloc default

- ☞ Optionnel
- ☞ Exécuté si aucun des cas n'est valide

2.3 La condition « switch »

▪ La condition « **switch** »

☞ Exemple

```
#include <iostream>
using namespace std;
int main()
{
    printf (" saisir le nombre de vos enfants!");
    int nbEnfants = 0;
    scanf("%d ", &nbEnfants);
    switch (nbEnfants)
    {
        case 0:
            cout << "Eh bien alors, vous n'avez pas d'enfant ?" << endl;
            break;
        case 1:
            cout << "Alors, c'est pour quand le deuxieme ?" << endl;
            break;
        case 2:
            cout << "Quels beaux enfants vous avez la !" << endl;
            break;
        default:
            cout << "Bon, il faut arreter de faire des gosses maintenant !" << endl;
            break;
    }
    return 0;
}
```

2.3 La condition « switch »

▪ La condition « **switch** »

☞ Exercice :

On souhaiterait gérer le menu d'une console de jeux vidéo. 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 :

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

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

1.7 Les boucles

▪ Définition

☞ C'est une technique permettant de répéter les mêmes instructions plusieurs fois



☞ Il existe trois type de boucles

- *while* : (*tant que ... faire*)
- *do ... while* : (*faire ... tant que*)
- *for* : *pour ...*

1.7 Les boucles

▪ Boucle « **while** »

```
while(condition)
{
    // Instructions à répéter
}
```

- Ordre des traitements
 - ☞ Test de la condition
 - Sortie si condition fausse
 - ☞ Exécution des instruction
- Instruction pas forcément exécutées

☞ Exemple 1 :

```
int nombreEntre = 0;
printf("Tapez un nombre! ");
scanf("%d", &nombreEntre);
while (nombreEntre != 47)
{
    printf("Tapez le nombre 47 ! ");
    scanf("%d", &nombreEntre);
}
```

1.7 Les boucles

▪ Boucle « **while** »

```
while(condition)
{
    // Instructions à répéter
}
```

- Ordre des traitements
 - ☞ Test de la condition
 - Sortie si condition fausse
 - ☞ Exécution des instruction
- Instruction pas forcément exécutées

☞ *Exemple 2 : Utilisation de l'incrémentation*

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

☞ **Remarque** : Attention aux boucles infinies. Cela arrive lorsque la condition est toujours vérifiée

1.7 Les boucles

▪ Boucle «do ... while »

```
do
{
    // Instructions
}
while(condition) ;
```

- Ordre des traitements
 - ☞ Exécution des instruction
 - ☞ Test de la condition
 - Sortie si condition fausse
- Instructions exécutées au moins une fois
- **Attention** : mettre un point virgule à la fin du while

☞ Exemple

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

1.7 Les boucles

▪ Boucle «**for**»

```
for(initialisation ; condition ; progression)
{
    // Instructions
}
```



```
Initialisation
while(condition)
{
    // Instructions
    progression
}
```

▪ Partie initialisation

- ☞ Peut contenir une déclaration de variable
- ☞ Cette variable est **locale** à la boucle

1.8 Les fonctions

- Jusque là, nous avons utiliser un certain nombre de fonctions déjà disponible dans le standard C.

☞ *main()*

☞ *Printf(...)*

☞ *Scanf(...)*

☞ *Pow(x,a)*

☞ ...

- **Question ?**

☞ *A quoi servent réellement les fonctions ?*

☞ *Comment fonctionnent-elles?*

☞ *Pouvons-nous créer nos propres fonctions ?*

1.8 Les fonctions

▪ A quoi servent les fonctions ?

☞ **Exemple** : On souhaiterait calculer la moyenne de notes des étudiants en 1^{ère} année d'informatique. Ils ont en tout 10 matières. Chaque étudiant est identifié par son numéro de carte d'étudiant. Ecrire un programme permettant d'entrer successivement les 10 notes et de calculer sa moyenne des étudiants suivants.

➤ *Etudiant 1 : 52348*

➤ *Etudiant 2 : 52344*

➤ *Etudiant 3 : 52286*

➤ ...

1.8 Les fonctions

■ A quoi servent les fonctions ?

☞ **Exemple :**

```
#include <stdio.h>

int main()
{
    double moyenne = 0.0;
    double somme = 0.0;
    printf("-----Calcul de la moyenne de l'étudiant n° 52348 -----\\n");
    for(int i = 0; i < 9; i++)
    {
        printf("Entrer la note n°%d : \\n", i);
        double note = 0;
        scanf("%lf", &note);
        somme += note;
    }
    moyenne = somme/10;
    printf("la moyenne est %f ", moyenne);

    printf("-----Calcul de la moyenne de l'étudiant n° 52286 -----\\n");
    for(int i = 0; i < 9; i++)
    {
        printf("Entrer la note n°%d : \\n", i);
        double note = 0;
        scanf("%lf", &note);
        somme += note;
    }
    moyenne = somme/10;
    printf("la moyenne est %f ", moyenne);
    return 0;
}
```


1.8 Les fonctions

▪ A quoi servent les fonctions ?

☞ On constate une répétition de code pour le calcul de la moyenne de chaque étudiant

☞ Imaginons qu'on veuille calculer la moyenne des 100 étudiants !!!

☞ *Utilité des fonction*

☞ En utilisant les fonctions, on pourrait écrire le code de calcul de la moyenne une seule fois.

☞ Les *fonctions* permettent de mieux structurer les programmes en bout de code et utilisable autant de fois que l'on souhaite

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

1.8 Les fonctions

■ Comment fonctionne 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*
- ☞ *Schéma d'une fonction*

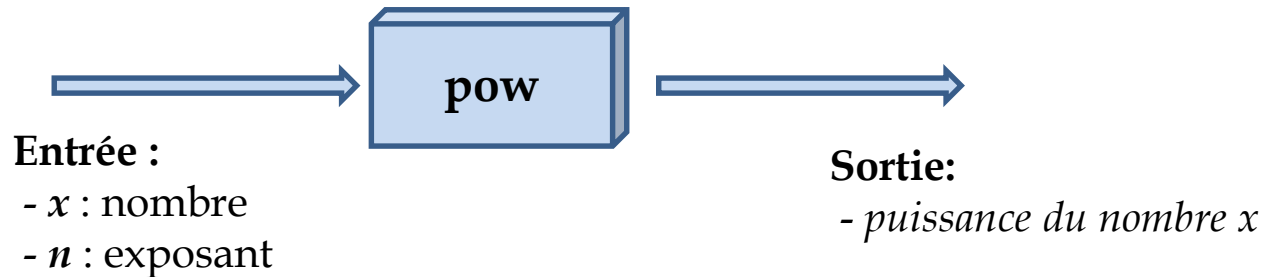


- **Entrée** : on fait « rentrer » des informations dans la fonction (en lui donnant des informations avec lesquelles travailler)
- **Les calculs** : grâce aux informations qu'elle a reçues en entrée, la fonction travaille
- **La sortie** : une fois qu'elle a fini ses calculs, la fonction renvoie un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

1.8 Les fonctions

▪ Comment fonctionne une fonction?

☞ *Exemple* : nous avons vu que la fonction « `pow(x,n)` » permettait de calculer la puissance d'un nombre x



1.8 Les fonctions

▪ Comment fonctionne une fonction?

☞ *Syntaxe d'une fonction*

```
Type nomFonction( Type idParam1, Type idParam2, ..., Type idParamN ) ;
```

→ Liste des paramètres

→ Nom de la fonction

→ Type de retour

- int, char, float, double, ...

1.8 Les fonctions

▪ Comment fonctionne une fonction?

☞ *Syntaxe d'une fonction*

- En fonction du type de retour, il existe 2 types 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 »). On les appelle des *procédures*

☞ *Exemple :*

- ☞ La fonction « *pow* » est déclarée comme suit

☞ *double pow(double x, double n)*

- ☞ La fonction « *scanf* » est déclarée comme suit

☞ *void scanf(char * a, void *n)*

1.8 Les fonctions

▪ Comment fonctionne une fonction?

☞ *Syntaxe d'une fonction*

➤ **Remarque :**

- 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
- *Idéalement, le nom de la fonction doit correspondre à la tâche qu'elle effectue*
- Une fonction peut être définie sans paramètres

1.8 Les fonctions

- **Comment créer nos propres fonctions?**

- ☞ *Oui, en utilisant la syntaxe de définition de fonction*

- *Cependant, il ya une certaine règle de conduite à tenir*

- ☞ *Exemple : on souhaite écrire une fonction prenant en paramètre un entier et renvoyant le triple du paramètre en entrée*

```
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
}
```

1.8 Les fonctions

■ Comment créer nos propres fonctions?

```
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
}
```

☞ La fonction est de type : *int*

- Elle doit retourner une valeur de type « *int* »
- Elle prend en paramètre une variable de type « *int* »

☞ « *return resultat* » : 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

1.8 Les fonctions

▪ Comment créer nos propres fonctions?

☞ *Remarque :*

- Il est tout à fait possible de créer des fonctions acceptant plusieurs paramètres
- *Exemple :* soit la fonction « addition » additionnant deux entiers a et b

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

- Il suffit de séparer les différents paramètres par une virgule comme vous le voyez.

1.8 Les fonctions

▪ Comment créer nos propres fonctions?

☞ *Remarque :*

- Il est également possible d'avoir des fonctions sans paramètres
- *Exemple :* soit la fonction « Bonjour » affichant *bonjour* à l'écran

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

- La fonction ne prend aucun paramètre.
- Elle est de type *void* ➔ pas besoin de return.

1.8 Les fonctions

▪ Comment créer nos propres fonctions?

☞ *Appeler une fonction:*

- Elle se fait de la même façon qu'on utilise les fonctions « *pow* », « *printf* », « *scanf* », ...
- Règle générale
 - On écrit généralement la fonction AVANT la fonction *main*
 - Si elle est écrite avant le main, cela ne fonctionnera pas

1.8 Les fonctions

■ Comment créer nos propres fonctions?

☞ *Appeler une fonction:*

➤ Exemple : Implémentons la fonction « triple ».

```
#include <stdio.h>
#include <stdlib.h>
int triple(int nombre)
{
    return 3 * nombre;
}
int main()
{
    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;
}
```

1.8 Les fonctions

■ Comment créer nos propres fonctions?

☞ *Appeler une fonction:*

➤ Il est tout à fait possible d'appeler la fonction « *triple* » dans le « *printf* »

```
#include <stdio.h>
#include <stdlib.h>
int triple(int nombre)
{
    return 3 * nombre;
}
int main()
{
    int nombreEntre = 0, nombreTriple = 0;
    printf("Entrez un nombre... ");
    scanf("%d", &nombreEntre);
    printf("Le triple de ce nombre est %d\n", triple(nombreEntre));
    return 0;
}
```

1.8 Les fonctions

■ Comment créer nos propres fonctions?

☞ *Fonctionnement:*

- 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.

1.8 Les fonctions

▪ La fonction « main »

☞ Fonction spéciale permettant d'exécuter un programme chargé en mémoire

☞ **Attention:** la fonction « main » est un nom réservé car elle permet au compilateur de savoir le début du programme. On n'a pas le droit de nommer une fonction quelconque « main » dans un programme

☞ La fonction « main » est appelée par le système d'exploitation, elle ne peut pas être appelée par le programme.

☞ *Forme globale*

```
int main(int argc, char *argv[])  
{  
    // instructions  
    return 0;  
}
```

1.8 Les fonctions

- **Exercice 1 :** *Ecrire la fonction « conversion » permettant de convertir les euros en franc cfa. (1 euro = 655,957 francs). Elle prend en entrée un nombre réel et renvoie un réel. Dans la fonction « main », convertissez 10, 50, et 150 euros en CFA*
- **Exercice 2 :** *Soit le code suivant ci-dessous: que fait la fonction « punition »? Quel est le résultat obtenu dans le main*

```
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;
}
```


1.8 Les fonctions

- **Exercice 3 :** *Ecrire la fonction « aireRectangle » prenant en paramètre la longueur et la largeur du rectangle et renvoie la surface. Dans la fonction « main » calculer la surface des rectangles suivantes:*
 - ☞ 1. $L = 10$ et $l = 5$
 - ☞ 2. $L = 56$ et $l = 10$
- **Exercice 4 :** *Reprendre la même fonction mais en affichant la surface du rectangle à l'intérieur de la fonction*

1.8 Les fonctions

■ Exercice 5 : *Que fait le code suivant :*

```
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 sures a la sauce de myrtilles enrobees de chocolat\n");
        printf("3 : Escalope de kangourou saignante et sa gelee aux fraises poivrees\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 etes un sacre aventurier"); break }
    return 0;
}
```

2. Modularité en langage C

2.1 Généralités

- Jusqu'ici, nous avons travaillé dans un seul fichier « *main.c* »
 - Cela était acceptable car nos programmes étaient petits
- Que se passera t-il si l'on disposait d'un centaine voir des milliers de fonctions?
 - En les mettant toutes dans un même fichier celui-là va finir par devenir très long !
- **Solution :**
 - ☞ Plutôt que de placer tout le code de notre programme dans un seul fichier (*main.c*), nous le « séparons » en plusieurs petits fichiers
 - ☞ C'est qu'on appelle la *programmation modulaire (ou modularité)*

2.2 Prototypes

- Jusqu'ici, les fonctions sont mises avant la fonction main.
 - ☞ l'ordre a une réelle importance ici :
 - En mettant la fonction avant le main dans votre code source, votre ordinateur l'aura lue et la connaîtra
 - Lorsque vous ferez un appel à la fonction dans le main, l'ordinateur connaîtra la fonction et saura où aller la chercher.
 - En revanche, si vous mettez votre fonction après le main, ça ne marchera pas car l'ordinateur ne connaîtra pas encore la fonction. Essayez, vous verrez !
 - ☞ Grâce à la modularité, vous pourrez écrire vos fonctions sans vous soucier de son emplacement.

2.2 Prototypes

▪ Définition

- ☞ Le *prototype d'une fonction* est le fait de déclarer ladite fonction sans le coder
- ☞ *Exemple* : soit la fonction « aireRectangle » permettant de calculer l'aire d'un rectangle

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

- Le prototype de cette fonction s'écrit :

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

Attention : il ne faut pas oublier le point virgule « ; »

2.2 Prototypes

■ *Principe*

☞ *On écrit le prototype de la fonction juste avant la fonction « main ». Ensuite on peut implémenter ladite fonction avant ou après la fonction principale*

☞ *Exemple :*

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

// La ligne suivante est le prototype de la fonction aireRectangle
double aireRectangle(double largeur, double hauteur);
int main(int argc, char *argv[])
{
    printf("Rectangle de largeur 5 et hauteur 10. Aire = %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;
}

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

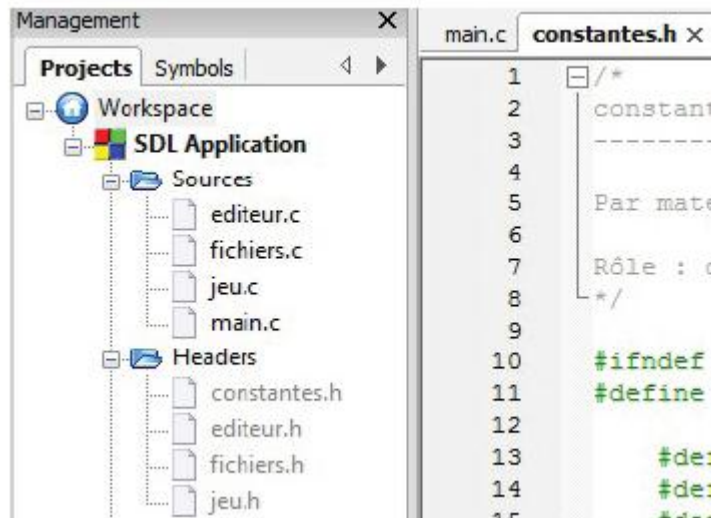
2.2 Prototypes

▪ **Principe**

- ☞ Un **prototype** est en quelque sorte une indication pour l'ordinateur.
 - Cela lui indique à la fonction « main » qu'il existe une fonction appelée « *aireRectangle* » qui prend tels paramètres en entrée et renvoie une sortie du type que vous indiquez. nombreuses fonctions : mieux vaut prendre dès maintenant la bonne habitude d'écrire le prototype de chacune d'elles.
- ☞ Désormais, toutes nos fonctions devront au préalable avoir leur prototypes.
- ☞ **Remarques :**
 - ☞ La fonction « main » est la seule fonction qui n'a pas de prototype
 - ☞ L'ordinateur la connaît

2.3 Plusieurs fichiers par projet

- Dans la pratique, pour réaliser des applications, on est amené à écrire nos programmes dans plusieurs fichiers.
 - ☞ Cela permet d'assurer la lisibilité du code
- Nous verrons comment gérer nos codes dans plusieurs fichiers
- *Exemple d'un projet*



2.3 Plusieurs fichiers par projet

- **Fichiers « .h » et « .c »**

- ☞ Il existe deux types de fichiers

- ☞ Les **fichiers « headers »** (ou *entête*)

- Identifiable par l'extension « .h »

- Ces fichiers contiennent les *prototypes des fonctions*

- ☞ Les **fichiers source** (ou *code source*)

- ☞ Ces fichiers contiennent les fonctions elles-mêmes (i.e. l'implémentation de ces fonctions)

2.3 Plusieurs fichiers par projet

- **Fichiers « .h » et « .c »**

☞ *Remarques :*

- En général, on met donc rarement les prototypes dans les fichiers « .c » (sauf si le programme est tout petit).
- Pour chaque fichier .c, il y a son équivalent .h qui contient les prototypes des fonctions.

- **Question :** *Mais comment faire pour que l'ordinateur sache que les prototypes sont dans un autre fichier que le « .c » ?*

- **Solution :** Il faut inclure le « .h » dans le code source grâce aux directives du préprocesseur

2.3 Plusieurs fichiers par projet

▪ Atelier

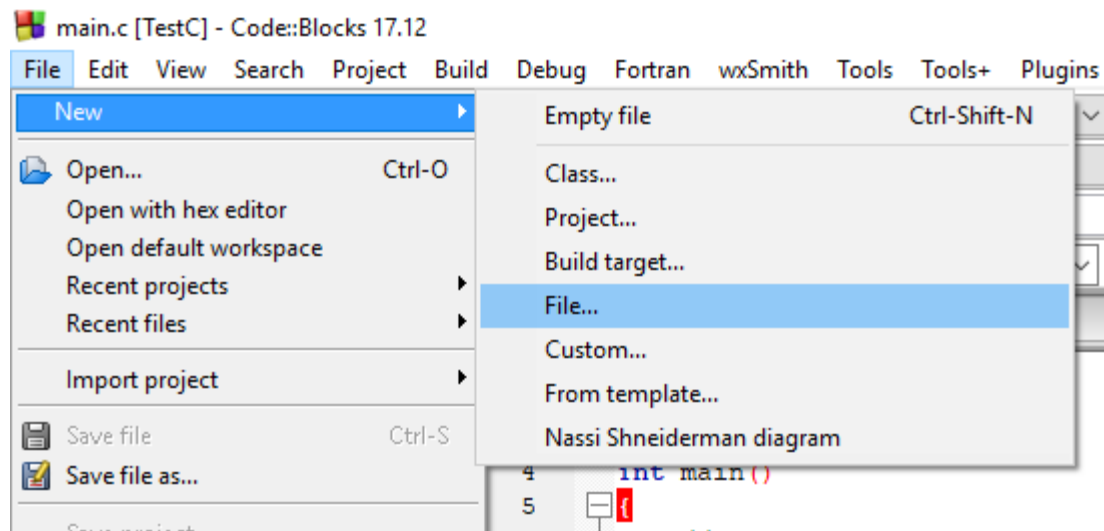
Nous souhaitons écrire un programme permettant de réaliser des calculs mathématiques. Pour cela, nous allons regrouper nos fonctions en deux modules:

- *1^{er} module: « common » contenant les fonctions de calcul de bases*
- *2^{ème} module : « mathematique » permettant de réaliser des calculs avancés*
- *Nous allons créer un projet nommé « Atelier1 » à sauvegarder dans le repertoire « workspace »*
- *Pour créer les entêtes (headers) et code source de deux modules, il faudra suivre les étapes suivantes sous Code::Blocks :*

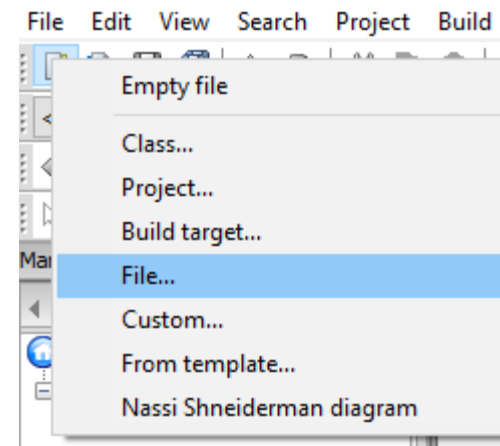
2.3 Plusieurs fichiers par projet

■ Atelier

☞ 1^{ère} étape



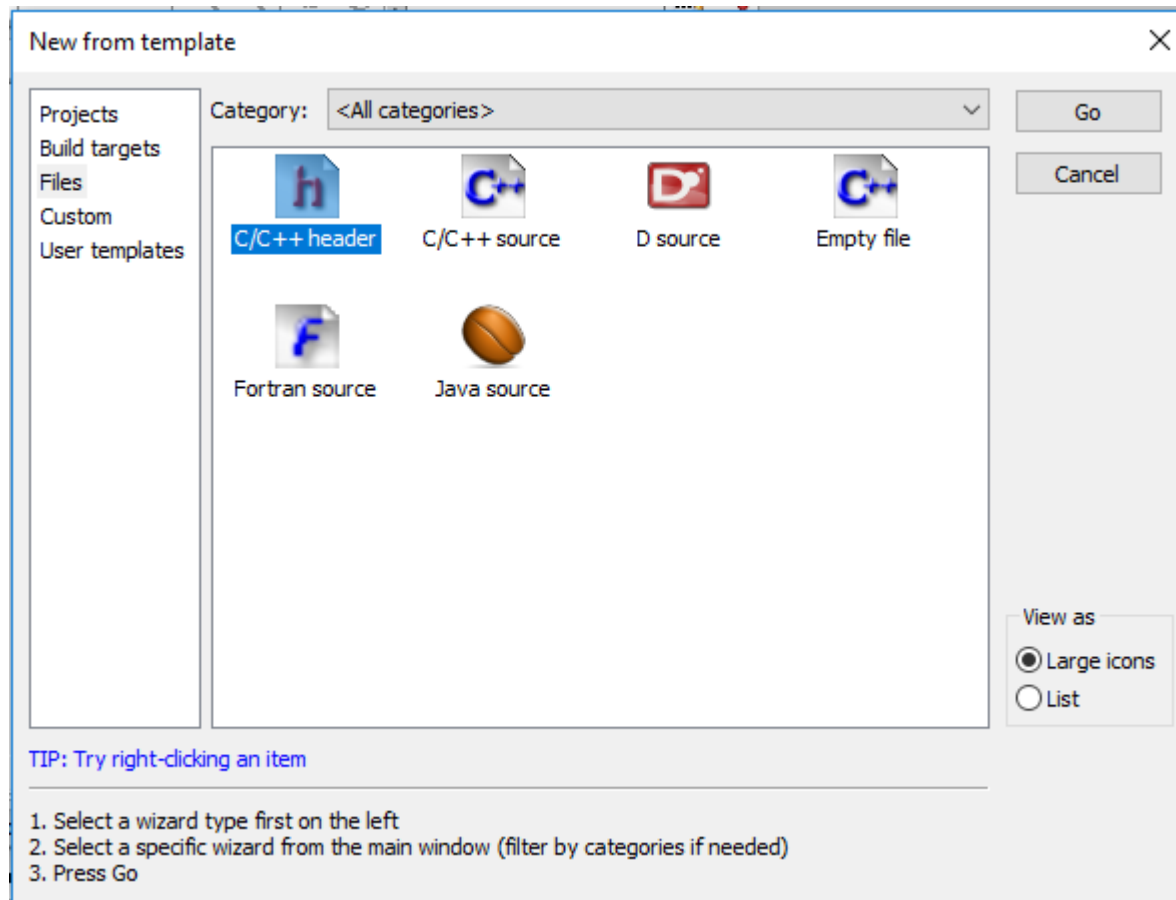
➤ Ou cliquant sur l'icône  dans le toolbar



2.3 Plusieurs fichiers par projet

■ Atelier

☞ 2^{ème} étape : création de l'entête



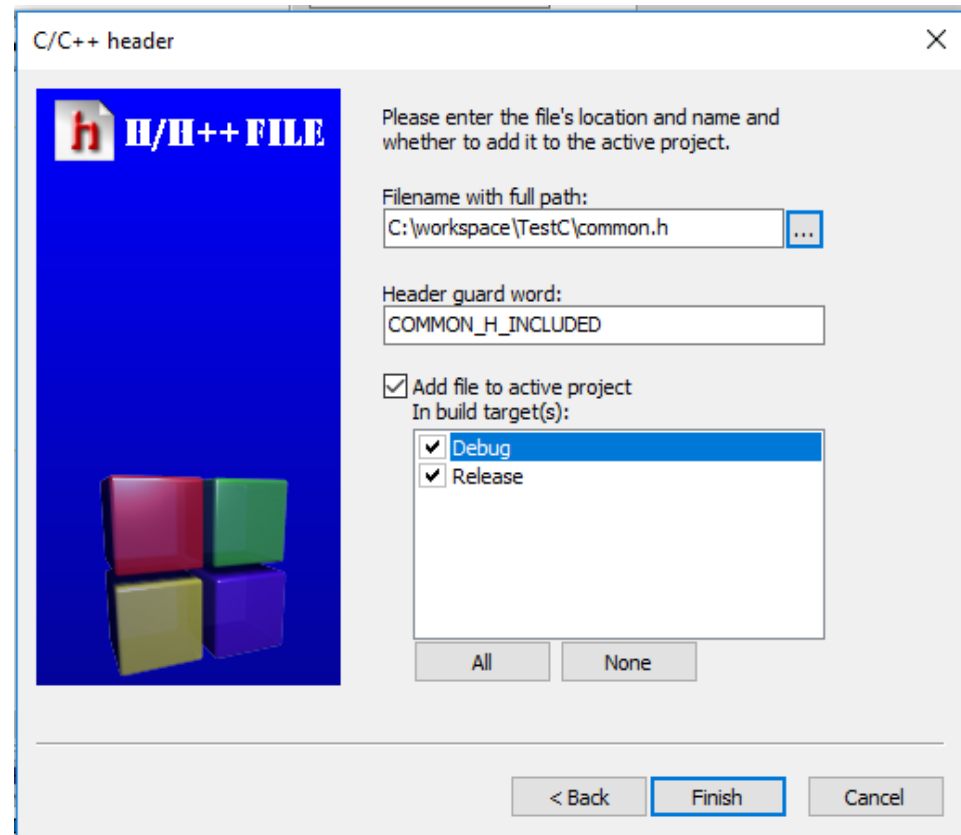
➤ Une fois la sélection effectuée, cliquez sur le bouton « Go »

2.3 Plusieurs fichiers par projet

▪ Atelier

☞ 3^{ème} étape : continuer jusqu'à atteindre la page ci-dessous.

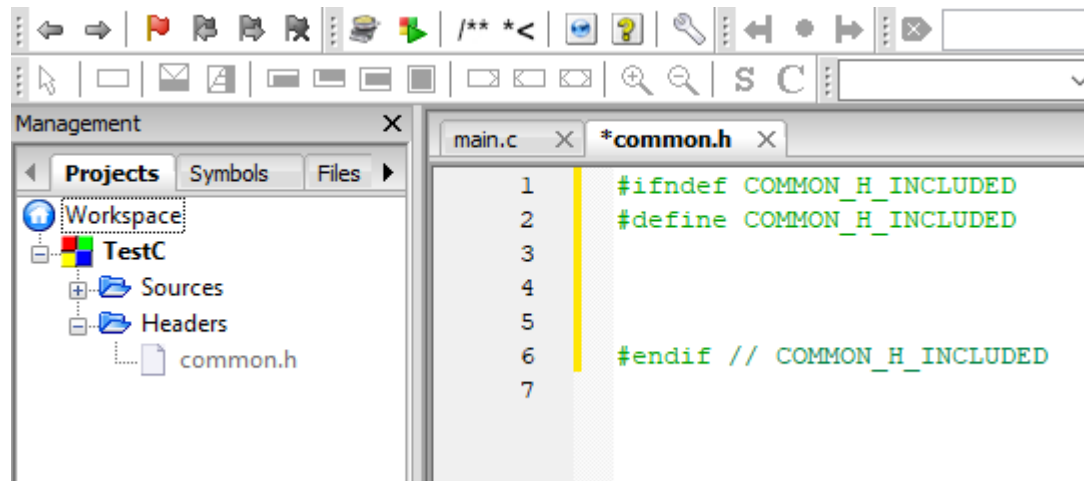
- Saisir le nom du fichier en cliquant sur le bouton « ... »
- Cocher les options
- Cliquer sur « finish »



2.3 Plusieurs fichiers par projet

■ Atelier

☞ 4^{ème} étape : On aboutit alors au résultat suivant



➤ L'entête créée est donc prêt pour accueillir les prototypes.

☞ Suivre les mêmes étapes pour les fichiers suivants :

- « **common.c** » (mais en choisissant « C/C++ sources » de l'étape 2)
- « *mathematique.h* » et « *mathematique.c* »

2.3 Plusieurs fichiers par projet

▪ Atelier

☞ Dans le fichier « *common.h* » écrire les prototypes de fonctions suivantes:

- « *addition* » prenant en paramètres deux entiers et renvoyant la somme des deux
- « *multiplication* » prenant en paramètres deux nombres réels et renvoyant leur produit
- « *puissance* » prenant en paramètres un nombre réel et un exposant réel et renvoie le résultat
- « *estMultiple* » prenant en paramètre deux entiers a et b. Il renvoie 1 si a est un multiple de b. Sinon il renvoie 0
- « *racineCarre* » prenant en paramètre un nombre réel et retourne sa racine carrée. Attention au nombre négatif
- « *afficher* » prenant en paramètre 3 entiers et affiche le minimum parmi les trois

2.3 Plusieurs fichiers par projet

▪ Atelier

☞ Dans le fichier « *common.c* » implémenter ces fonctions:

- Pour ce faire, il faudra que l'ordinateur sache où se trouve les prototypes
 - Autrement dit l'entête « *common.h* »
 - Il suffit d'inclure le fichier « *common.h* » à l'aide des directives du préprocesseur

➤ **#include "common.h"**

☞ *Remarque* : On adoptera la nomenclature suivante

- ☞ *les chevrons < > pour inclure les librairies de la bibliothèque standard*
- ☞ *les guillemets " " pour inclure les fichiers que nous avons nous-même créé*

2.3 Plusieurs fichiers par projet

▪ Atelier

☞ Dans le fichier « *mathematique.h* » écrire les prototypes de fonctions suivantes:

- « *distance* » prenant en paramètres quatre nombre réels x_A , y_A (coordonnées du point A) et x_B , y_B (coordonnées du point B) puis calcule la distance entre les 2 points.
 - NB: On utilisera les fonctions définies dans « *common.h* ». Il faudra donc l'inclure dans le fichier « *mathematique.h* ».
- « *produitScalaire* » prenant les mêmes paramètres que la fonction précédente et renvoie le produit scalaire

2.3 Plusieurs fichiers par projet

▪ Atelier

- ☞ Dans le fichier « *mathematique.c* » implémenter ces fonctions:
- ☞ Dans la fonction principale, inclure les entêtes « *common.h* » et « *mathematique.h* ». Proposer à l'utilisateur les options suivantes:
 - ☞ 1. Calcul de distance
 - ☞ 2. Calcul de produit scalaire
 - ☞ 3. Déterminer si un nombre est multiple 8.
- ☞ En fonction des choix, demander à l'utilisateur d'entrer les paramètres adéquates et appeler les fonctions correspondantes

2.4 Portée des variables

- En déclarant une variable à l'intérieur d'une fonction, celle-ci est supprimée de la mémoire à la fin de la fonction
- Exemple : reprenons la fonction « triple »

```
int triple(int nombre)
{
    int resultat = 0;
    resultat = 3 * nombre; // On multiplie le nombre fourni par 3
    return resultat; // La fonction est terminée, la variable resultat est supprimée
                        de la mémoire
}
```

- Une variable déclarée dans une fonction n'existe donc que pendant que la fonction est exécutée.
- **Qu'est-ce que ça veut dire, concrètement ?**
 - On ne peut pas y accéder depuis une autre fonction

2.4 Portée des variables

- Exemple

```
int triple(int nombre);

int main(int argc, char *argv[])
{
    printf("Le triple de 15 est %d\n", triple(15));
    printf("Le triple de 15 est %d", resultat); // Erreur
    return 0;
}

int triple(int nombre)
{
    int resultat = 0;
    resultat = 3 * nombre;
    return resultat;
}
```

- Dans le *main*, on essaie d'accéder à la variable « *resultat* ». Or, comme cette variable « *resultat* » a été créée dans la fonction « *triple* », elle n'est pas accessible dans la fonction *main* !

2.4 Portée des variables

▪ A RETENIR PAR CŒUR

- une variable déclarée dans une fonction n'est accessible qu'à l'intérieur de cette fonction. On dit que c'est une **variable locale**
- Il est tout à fait possible de déclarer une variable accessible à tous. On parle alors de **variable globale**
- Exemple

```
#include <stdio.h>
#include <stdlib.h>
int resultat = 0; // Déclaration de variable globale
void triple(int nombre); // Prototype de fonction
int main(int argc, char *argv[])
{
    triple(15); // On appelle la fonction triple, qui modifie la
               // variable globale resultat
    printf("Le triple de 15 est %d\n", resultat); // On a accès à
    resultat
    return 0;
}
void triple(int nombre)
{
    resultat = 3 * nombre;
}
```

- La variable « resultat » est accessible à tous

3. Structure de données

3.1 Généralités

- Jusqu'ici, nous avons travaillé les variables
 - **Rappel** : une variable ne permet de stocker qu'une seule et unique valeur
- Que se passe t-il si on devait stocker plusieurs valeurs du même type?
- Dans ce chapitre, nous verrons particulièrement deux types de variables permettant de stocker plusieurs valeurs :
 - ☞ *Les tableaux*
 - ☞ *Les structures*

3.2 Les tableaux

▪ Définition

- ☞ **Les tableaux** sont une suite de variables de même type, situées dans un espace contigu en mémoire
- ☞ « Grosse variable » pouvant contenir plusieurs nombre du même type (**long, int, char, double...**)
- Un tableau a une dimension précise. Il peut occuper 2, 3, 10, 150, 2 500 cases, c'est vous qui décidez !!!
- **NB:** si un tableau est de type « int », alors toutes ses cases sont également de type « int »
 - ☞ *On ne peut pas faire de tableau contenant à la fois des « int » et des « double » par exemple*

3.2 Les tableaux

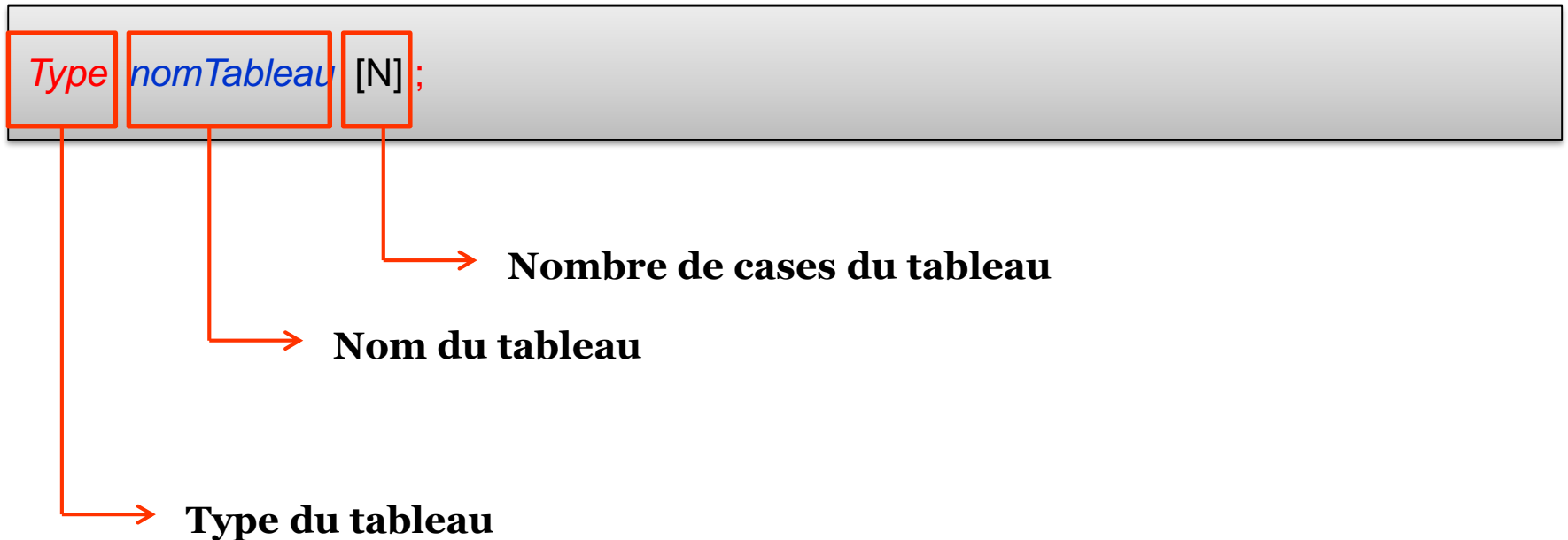
- **Représentation d'un tableau**

Adresse	Valeur
1600	10
1601	23
1602	505
1603	8

- Le tableau a 4 cases en mémoire qui commence à l'adresse 1600.

3.2 Les tableaux

- Comment définir un tableau ?



👉 **Attention :** La taille (ou nombre de cases) est forcément fournie à la déclaration

👉 *Exemple*

```
int tab[1000] ; // Déclaration d'un tableau d'entiers de 1000 éléments
```

3.2 Les tableaux

▪ Comment définir un tableau ?

☞ On peut également déclarer un tableau initialisé

```
Type nomTableau[] = {valeur1, valeur2,...,valeurN} ;
```

☞ La taille est déduite du nombre de valeurs fournies

☞ *Exemple*

```
// Déclaration d'un tableau de 4 éléments initialisés  
int tab[] = {10, 11, -1, 5} ; // tableau variable
```

3.2 Les tableaux

▪ Comment accéder à une case du tableau ?

☞ Syntaxe

```
nomTableau[ numeroDeLaCase ]
```

➤ On accède à une case grâce à son indice « *numeroDeLaCase* »

☞ Un tableau commence toujours à l'indice 0.

➤ Un tableau de taille N aura les indices compris entre 0 et N-1

☞ *Exemple*

```
int tab[5] ; // Déclaration d'un tableau d'entiers de 5 éléments
// ces indices vont de 0 à 5
int valeur = tab[2]; // « valeur » contiendra la valeur contenue dans
l'indice 2 du tableau (3ème case)
```

3.2 Les tableaux

▪ Comment accéder à une case du tableau ?

☞ Syntaxe

```
nomTableau[ numeroDeLaCase ]
```

➤ On accède à une case grâce à son indice « *numeroDeLaCase* »

☞ Un tableau commence toujours à l'indice 0.

➤ Un tableau de taille N aura les indices compris entre 0 et N-1

☞ *Exemple*

```
int tab[5] ; // Déclaration d'un tableau d'entiers de 5 éléments
// ces indices vont de 0 à 5
int valeur = tab[2]; // « valeur » contiendra la valeur contenue dans
l'indice 2 du tableau (3ème case)
tab[3] = 10; // La case à l'indice 3 du tableau (4ème case) est initialisé
avec la valeur 10
```

3.2 Les tableaux

▪ Comment parcourir un tableau ?

☞ On peut se servir d'une boucle (idéalement la boucle « *for* »)

☞ *En utilisant les indices du tableau*

☞ *Exemple*

```
int main()
{
    int tableau[4], i = 0;
    tableau[0] = 10;
    tableau[1] = 23;
    tableau[2] = 505;
    tableau[3] = 8;
    for (i = 0 ; i < 4 ; i++)
    {
        printf("%d\n", tableau[i]);
    }
    return 0;
}
```

▪ **Attention : on peut pas accéder à la case `tableau[4]` car les indices vont de 0 à 3**

3.2 Les tableaux

▪ Exercice 1

- ☞ Ecrire un tableau de taille 500 et contenant des valeurs réelles. Initialiser le tableau avec la valeur 2.

- ☞ Ecrire un code demandant à l'utilisateur de saisir un nombre entier N et ensuite de saisir successivement des valeurs. Puis il remplit un tableau avec les valeurs saisies par l'utilisateur.
 - Afficher le contenu du tableau

3.2 Les tableaux

▪ Comment utiliser un tableau comme paramètre d'une fonction?

☞ Il faut obligatoirement préciser le tableau et sa taille

➤ Pour utiliser un tableau comme paramètre : *type* ***nomFonction*** []

☞ Exemple

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

3.2 Les tableaux

■ Exercices

Toutes ces fonctions devront être implémentées dans « calcul.h » et « calcul.c »

- ☞ Créez une fonction « *sommeTableau* » prenant en paramètre un tableau *t* et sa taille *n* puis calcule et renvoie la somme des valeurs contenues dans le tableau
- ☞ Créez une fonction « *moyenneTableau* » prenant en paramètre un tableau et renvoie la moyenne des éléments du tableau
- ☞ Créez une fonction « *copierTableau* » qui prend en paramètre deux tableaux. Elle copie le contenu du premier tableau dans le second tableau.
- ☞ Créez une fonction « *maximumTableau* » prenant en paramètre un tableau « *tab* », sa taille « *n* » et un entier « *valeurMax* ». Elle aura pour rôle de remettre à 0 toutes les cases du tableau ayant une valeur supérieure à « *valeurMax* ».

3.3 Les chaînes de caractères

▪ Définition

- ☞ Une « *chaîne de caractères* », c'est un nom pour désigner du texte
 - C'est du texte que l'on peut retenir sous forme de variable en mémoire. On pourrait ainsi stocker le nom de l'utilisateur
- ☞ Nous avons vu que le type « *char* » permettait de créer un caractère
- ☞ Une chaîne de caractères est en réalité un tableau de caractère

3.3 Les chaines de caractères

■ Création et initialisation d'une chaine de caractères

☞ *Exemple* : on veut initialiser le tableau « chaine » avec le texte « Salut » :

☞ Méthode 1 : utilisation des indices du tableau

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char chaine[6]; // Tableau de 6 char pour stocker S-a-l-u-t +
                    // le \0
    // Initialisation de la chaîne (on écrit les caractères un à un
    // en mémoire)
    chaine[0] = 'S';
    chaine[1] = 'a';
    chaine[2] = 'l';
    chaine[3] = 'u';
    chaine[4] = 't';
    chaine[5] = '\0'; //caractère de fin de chaine
    // Affichage de la chaîne grâce au %s du printf
    printf("%s", chaine);
    return 0;
}
```

3.3 Les chaines de caractères

■ Création et initialisation d'une chaine de caractères

☞ *Exemple* : on veut initialiser le tableau « chaine » avec le texte « Salut » :

☞ Méthode 2 : Initialisation directe

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    char chaine[6]; // Tableau de 6 char pour stocker S-a-l-u-t +
    chaine = "Salut";
    // Affichage de la chaîne grâce au %s du printf
    printf("%s", chaine);
    return 0;
}
```

3.3 Les chaines de caractères

▪ Parcours d'une chaine de caractères

☞ Pour parcourir une chaine de caractères on utilise une boucle (comme un tableau)

☞ Exemple

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    char chaine[6]; // Tableau de 6 char pour stocker S-a-l-u-t +
                    // le \0
    // Initialisation de la chaîne (on écrit les caractères un à un
    // en mémoire)
    chaine[0] = 'S';
    chaine[1] = 'a';
    chaine[2] = 'l';
    chaine[3] = 'u';
    chaine[4] = 't';
    chaine[5] = '\0'; //caractère de fin de chaine
    for(int i = 0; i < 6; i++)
    {
        printf("%c\n", chaine[i]);
    }
    return 0;
}
```

3.4 Les structures

■ Définition

- ☞ **Une structure** est un assemblage de variables qui peuvent avoir différents types
- ☞ Contrairement aux tableaux qui vous obligent à utiliser le même type dans tout le tableau, vous pouvez créer une structure comportant des variables de types *long*, *char*, *int* et *double* à la fois.
- ☞ Les structures sont généralement définies dans les fichiers « .h » au même titre que les prototypes

■ Syntaxe

```
struct nom_structure  
{  
    type1 champ1 ;  
    type2 champ2 ;  
    ...  
};
```

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

3.4 Les structures

▪ Exemple

- ☞ Déclaration d'une structure pour gérer les informations d'un étudiant

```
struct Etudiant
{
    char nom[20];
    char prenom[20];
    int genre;
    int numeroEtudiant;
    double listeNotes[10];
    int scolarite;
}
```

- ☞ NB : Une structure peut contenir des tableaux et mêmes d'autres structures

3.4 Les structures

■ Utilisation d'une structure

- ☞ **Exercice 1** : Créer un nouveau projet console nommé « Structure ». Créer deux fichiers « calcul.h » et « calcul.c ». Dans ces fichiers, déclarer une structure dénommée « Coordonnees » définissant les coordonnées d'un point. Un point est caractérisé par 2 éléments:
 - ☞ x : nombre réel
 - ☞ y : nombre réel
- ☞ Dans la fonction principale, on souhaiterait définir un point A avec ses coordonnées et les modifier. Pour cela :
 - Déclarer le point A comme suit:
 - **struct** coordonnees A;
 - *Demander à l'utilisateur de préciser les valeurs de x, y et z et modifier les coordonnées du point A*
 - *Pour accéder à un champ de A (exemple. y), il suffit de faire :*
 - **Variable.nomDuChamp**
 - Pour modifier le champ y de A, on fait : **A.y = y**
 - Déclarer un autre point B de coordonnées (5, 10)
 - Pour déclarer : **struct nom_structure variable = {valeur1, valeur2, ...};**

3.4 Les structures

■ Utilisation d'une structure

☞ *Remarque :*

- A chaque fois qu'on déclare un nouveau point, on est toujours obligé de le précéder par le mot clé « **struct** ».
- Que se passera t-il s'il l'on déclare plusieurs nouveau point?
- Ce sera redondant d'appeler à chaque fois le mot clé « **struct** »
- Il existe une méthode permettant de ne plus utiliser le mot clé « **struct** ».
- On utilisera le mot clé « **typedef** ». Dans le fichier « calcul.h », juste avant la déclaration de la structure, nous allons ajouter une ligne commençant par « **typedef** »

```
typedef struct Coordonnees Coordonnees;  
struct Coordonnees  
{  
    double x;  
    double y;  
    double z;  
};
```

3.4 Les structures

■ Utilisation d'une structure

☞ **Remarque :**

➤ Cette ligne doit être découpée en trois morceaux

➤ **typedef** : indique que nous allons créer un alias de structure ;

➤ **struct Coordonnees** : c'est le nom de la structure dont vous allez créer un alias (c'est-à-dire un « équivalent ») ;

➤ **Coordonnees** : c'est le nom de l'équivalent.

☞ En clair, cette ligne dit « Écrire le mot **Coordonnees** est désormais équivalent à écrire **struct Coordonnees** ».

☞ Désormais plus besoin de mettre le mot « struct » à chaque définition de variable de type Coordonnees.

☞ On peut donc retourner dans notre main et écrire tout simplement :

☞ **Coordonnees A;**

3.4 Les structures

■ Utilisation d'une structure

☞ **Exercice 2** : Déclarer les structures suivantes dans le fichiers « calcul.h » :

- « **Rectangle** » ayant les champs
 - *Longueur* : nombre réel
 - *Largeur* : nombre réel
- « **Cercle** »
 - « *rayon* » : nombre réel
 - « *centre* » : un entier
- « **Triangle** » :
 - « *base* » : nombre réel
 - « *hauteur* » : nombre réel
- Dans la fonction principale :
 - Initialiser un rectangle appelé « monRect » avec les valeurs (14, 15). Afficher les information du rectangle
 - Déclarer un cercle appelé « monCercle » et demander à l'utilisateur de saisir le rayon et la distance. Afficher ses informations
 - Idem pour le triangle

3.4 Les structures

■ Tableau de structure

- ☞ **Exercice 3** : On désire maintenant créer une liste de N points. Cela revient à créer un tableau de type « Coordonnees »:

```
Coordonnees tableau[N];
```

- ☞ *Tout se passe exactement de la même façon comme si l'on avait un tableau d'entier*
- ☞ *« tableau[o] » renvoie les coordonnées d'un point se trouvant à l'indice o du tableau*
- Demander à l'utilisateur de saisir la valeur de N. Créer un tableau de coordonnées. Faire une boucle et demander à l'utilisateur de saisir les coordonnées (x, y, z) et modifier la case du tableau
- Pour modifier les coordonnées la 1^{ère} case du tableau :
- `tableau[o].x = x;`
 - `tableau[o].y = y;`
- Afficher les informations sur chaque coordonnées du tableau

3.4 Les structures

■ Tableau de structure

- ☞ **Exercice 4** : On considère dans un repère cartésien, quatre point A, B, C et D ayant les coordonnées suivantes:
- A (1, 2) , B(3, 2), C(1, -2) et D(3, -2)
 - On souhaiterait calculer l'aire des figures suivantes :
 - le rectangle ABCD
 - Le cercle de centre A et de rayon AB (distance AB)
 - Le triangle ABC
- ☞ 1. Dans les fichiers « calcul.h » et « calcul.c », écrire les fonctions suivantes:
- « **distance** » prenant en paramètres quatre nombres réels x_M , y_M , x_N et y_N renvoie la distance entre les points M et N
 - « **AireRectangle** » prenant en paramètres 4 points M, N, O, P et renvoie la surface du rectangle MNOP
 - « **AireTriangle** » prenant en paramètres 3 points M, N et O et renvoie la surface de la figure MNO
 - « **AireCercle** » prenant en paramètres deux points M et N et renvoie la surface du cercle
 - « **InitRectangle** » prenant en paramètres un rectangle et deux nombre réels « longueur » et « largeur » et initialise le rectangle

3.4 Les structures

■ Tableau de structure

- ☞ **Exercice 4** : On considère dans un repère cartésien, quatre point A, B, C et D ayant les coordonnées suivantes:
 - A (1, 2) , B(3, 2), C(1, -2) et D(3, -2)
 - On souhaiterait calculer l'aire des figures suivantes :
 - le rectangle ABCD
 - Le cercle de centre A et de rayon AB (distance AB)
 - Le triangle ABC

- ☞ Dans la fonction principale, créer et initialiser les points A, B, C et D avec leurs coordonnées respectives
 - Calculer et afficher l'aire des figures suivantes :
 - le rectangle ABCD
 - Le cercle de centre A et de rayon AB (distance AB)
 - Le triangle ABC
 - Créer le rectangle « Rect1 » et l'initialiser avec les valeurs {10, 20}. Afficher ses valeurs. Modifier les valeurs de ces champs « longueur » = distance AB et « largeur » =distance « AC ». Afficher les informations du rectangle « Rect1 ». Que constatez-vous?

4. Les pointeurs

3.4 Les structures

■ Tableau de structure

- ☞ **Exercice 3** : On désire maintenant créer une liste de N points. Cela revient à créer un tableau de type « Coordonnees »:

```
N = 5;  
Coordonnees tableau[N];
```

- ☞ *Tout se passe exactement de la même façon comme si l'on avait un tableau d'entier*
- ☞ *« tableau[o] » renvoie les coordonnées d'un point se trouvant à l'indice o du tableau*
- Faire une boucle et demander à l'utilisateur de saisir les coordonnées (x, y, z) et modifier la case du tableau
 - Pour modifier les coordonnées la 1^{ère} case du tableau :
 - `tableau[o].x = x;`
 - `tableau[o].y = y;`
 - Afficher les informations sur chaque coordonnées du tableau

5. Lecture et écriture dans des fichiers

6. Création de jeux 2D

MERCI