Mise à niveau en langage C
Cours/TP 1

ETAPES PERMETTANT L'EDITION, LA MISE AU POINT, L'EXECUTION D'UN PROGRAMME

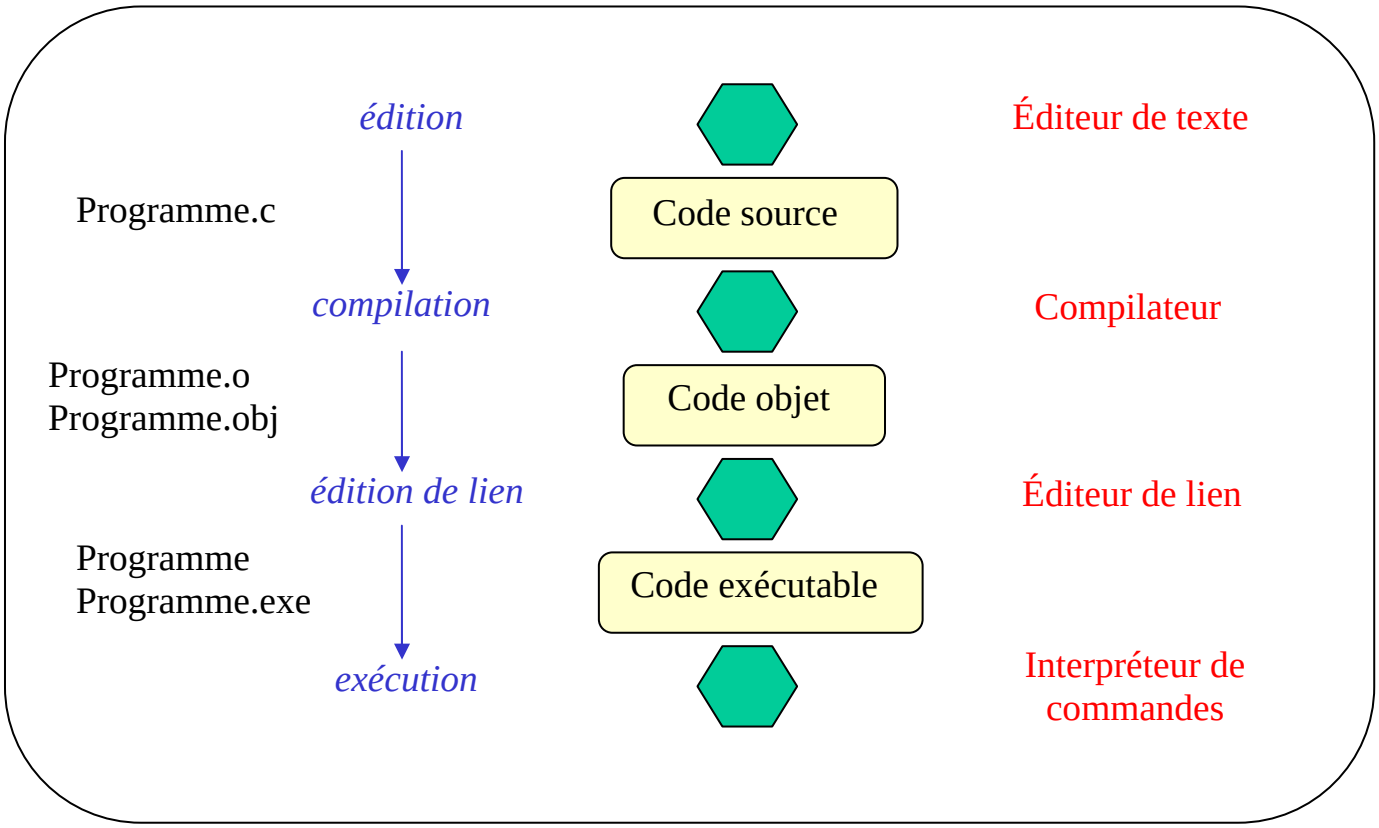
*1- **Edition du programme source***, à l'aide d'un éditeur (traitement de textes). Le nom du fichier contient l'extension .C, exemple: EXI_1.C (menu « edit »).

*2- **Compilation du programme source,*** c'est à dire création des codes machine destinés au microprocesseur utilisé. Le compilateur indique les erreurs de syntaxe mais ignore les fonctions-bibliothèque appelées par le programme.
Le compilateur génère un fichier binaire, non listable, appelé fichier objet: EXI_1.OBJ (commande « compile »).

*3- **Editions de liens***: Le code machine des fonctions-bibliothèque est chargé, création d'un fichier binaire, non listable, appelé fichier executable: EXI_1.EXE (commande « build all »).

*4- **Exécution du programme***.

Les compilateurs permettent en général de construire des programmes composés de plusieurs fichiers sources, d'ajouter à un programme des unités déjà compilées ...



Exercice 1 :
Editer, compiler et exécuter le programme suivant :

```
#include <stdio.h>    /* bibliothèque d'entrées-sorties standard */
#include <conio.h>

void main()
{
 puts("BONJOUR");     /* utilisation d'une fonction-bibliothèque */
 puts("Pour continuer frapper une touche...");
 getch();             /* Attente d'une saisie clavier */
}
```

LES DIFFERENTS TYPES DE VARIABLES

1- Les entiers

Le langage C distingue plusieurs types d'entiers :

TYPE	DESCRIPTION	TAILLE MEMOIRE
int	entier standard signé	4 octets: $-2^{31} \leq n \leq 2^{31}-1$
unsigned int	entier positif	4 octets: $0 \leq n \leq 2^{32}$
short	entier court signé	2 octets: $-2^{15} \leq n \leq 2^{15}-1$
unsigned short	entier court non signé	2 octets: $0 \leq n \leq 2^{16}$
char	caractère signé	1 octet: $-2^{7} \leq n \leq 2^{7}-1$
unsigned char	caractère non signé	1 octet: $0 \leq n \leq 2^{8}$

ⓘ Remarque : En langage C, le type **char** est un cas particulier du type entier :
un caractère est un entier de 8 bits

Quelques constantes caractères :

CARACTERE		VALEUR (code ASCII)	NOM ASCII
'\n'	interligne	0x0a	LF
'\t'	tabulation horizontale	0x09	HT
'\v'	tabulation verticale	0x0b	VT
'\r'	retour charriot	0x0d	CR
'\f'	saut de page	0x0c	FF
'\\'	backslash	0x5c	\
'\"	cote	0x2c	'
'\"'	guillemets	0x22	"

2- Les réels

Un réel est composé - d'un signe - d'une mantisse - d'un exposant
Un nombre de bits est réservé en mémoire pour chaque élément.

Le langage C distingue 2 types de réels :

TYPE	DESCRIPTION	TAILLE MEMOIRE
float	réel standard	4 octets
double	réel double précision	8 octets

SORTIES DE NOMBRES OU DE TEXTE A L'ECRAN
LA FONCTION PRINTF

Ce n'est pas une instruction du langage C, mais une fonction de la bibliothèque stdio.h.

Exemple : affichage d'un texte

```
printf("BONJOUR");    /* pas de retour à la ligne du curseur après */
                              /* l'affichage */
printf("BONJOUR\n"); /* affichage du texte, puis retour à la ligne */
                              /* du curseur */
```

Exercice 2 :

Tester le programme suivant et conclure.

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

void main()
{
 printf("BONJOUR ");
 printf("IL FAIT BEAU\n"); /* équivalent à puts("BONJOUR";  */
 printf("BONNES VACANCES");
 puts("Pour continuer frapper une touche...");
 getch();                  /* Attente d'une saisie clavier */
}
```

La fonction printf exige l'utilisation de formats de sortie, avec la structure suivante :
printf("%format",nom_de_variable);

Exercice 3 :
Tester le programme suivant et conclure.
Dans un deuxième temps, modifier ce programme pour améliorer l'interface utilisateur.

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

void main()
{
 char c;

 c =66;              /* c est le caractere alphanumerique A */
 printf("%d\n",c);   /* affichage du code ASCII en decimal */
                     /* et retour … à la ligne */
 printf("%o\n",c);   /* affichage du code ASCII en base huit
                     /* et retour … à la ligne */
 printf("%x\n",c);   /* affichage du code ASCII en hexadecimal */
                     /* et retour … à la ligne */
 printf("%c\n",c);   /* affichage du caractère */
                     /* et retour à la ligne */
 puts("Pour continuer frapper une touche...");
 getch();            /* Attente d'une saisie clavier */
}
```

Exercice 4 :
Affichage multiple de structure :
printf("format1 format2 … formatn",variable1,variable2,...,variablen);

Tester le programme suivant et conclure :

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

void main()
{
 char c;
 c ='A';        /* c est le caractere alphanumerique A */
 printf("decimal = %d  ASCII = %c\n",c,c);
 puts("Pour continuer frapper une touche...");
 getch();       /* Attente d'une saisie clavier */
}
```

Formats de sortie pour les entiers:

%d affichage en décimal (entiers de type int),
%x affichage en hexadécimal (entiers de type int),
%u affichage en décimal (entiers de type unsigned int),

D'autres formats existent.

Exercice 5 :
a et b sont des entiers, a = -21430, b = 4782, calculer et afficher a+b, a-b, a*b, a/b, a%b en format décimal, et en soignant l'interface homme/machine.
(a/b donne le quotient de la division, a%b donne le reste de la division)

Exercice 6 :
Que va-t-il se produire, à l'affichage, lors de l'exécution du programme suivant ?

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

void main()
{
 char a = 0x80;
 unsigned char b = 0x80;
 clrscr();
 printf("a en decimal vaut: %d\n",a);
 printf("b en decimal vaut: %d\n",b);
 puts("Pour continuer frapper une touche...");
 getch();       /* Attente d'une saisie clavier */
}
```

Format de sortie pour les réels: %f

LES DECLARATIONS DE CONSTANTES

Le langage C autorise 2 méthodes pour définir des constantes.

1ère méthode **: déclaration** d'une variable, dont la valeur sera constante pour tout le programme :

Exemple :
```
void main()
{
 const float PI = 3.14159;
 float perimetre,rayon = 8.7;

 perimetre =  2*rayon*PI;
....
}
```

Dans ce cas, le compilateur réserve de la place en mémoire (ici 4 octets), pour la variable pi, mais dont on ne peut changer la valeur.

2ème méthode **: définition d'un symbole** à l'aide de la directive de compilation **#define.**

Exemple:
```
#define PI = 3.14159;
void main()
{
 float perimetre,rayon = 8.7;

 perimetre =  2*rayon*PI;
....
}
```

Le compilateur ne réserve pas de place en mémoire. Les constantes déclarées par #define s'écrivent traditionnellement en majuscules, mais ce n'est pas une obligation.

LA FONCTION GETCH

La fonction getch, appartenant à la bibliothèque conio.h permet la saisie clavier d'un caractère alphanumérique, **sans écho écran**. La saisie s'arrête dès que le caractère a été frappé.

La fonction getch n'est pas définie dans la norme ANSI mais elle peut exister dans la bibliothèque d'autres compilateurs.

On peut utiliser getch de deux façons :

- sans retour de variable au programme :

Exemple :
```
printf("POUR CONTINUER FRAPPER UNE TOUCHE ");
getch();
```

- avec retour de variable au programme :

Exemple :
```
char alpha;
printf("ENTRER UN CARACTERE (ATTENTION PAS DE RETURN)");
alpha = getch();
printf("\nVOICI CE CARACTERE: %c",alpha);
```

Les parenthèses vides de getch() signifient qu'aucun paramètre n'est passé à cette fonction par le programme.

LA FONCTION SCANF

La fonction scanf, appartenant à la bibliothèque stdio.h, permet la saisie clavier de n'importe quel type de variable.
Les variables à saisir sont formatées, le nom de la variable est précédé du symbole & désignant l'adresse de la variable (On reverra ce symbole dans le chapitre sur les pointeurs).
La saisie s'arrête avec "RETURN" (c'est à dire LF), les éléments saisis s'affichent à l'écran (**saisie avec écho écran**). Tous les éléments saisis après un **caractère d'espacement** (espace, tabulation) sont ignorés.

Exemples :
```
char alpha;
int i;
float r;
scanf("%c",&alpha);  /* saisie d'un caractère */
scanf("%d",&i);      /* saisie d'un entier en décimal */
scanf("%x",&i);      /* saisie d'un entier en hexadécimal */
scanf("%f",&r);      /* saisie d'un réel */
```

ⓘ Remarque : si l'utilisateur ne respecte pas le format indiqué dans scanf, la saisie est ignorée. Aucune erreur n'est générée.

Exemple :
```
char alpha;
scanf("%d",&alpha);
```

Si l'utilisateur saisie 97 tout va bien, alpha devient le caractère dont le code ASCII vaut 97.
Si l'utilisateur saisie la lettre a, sa saisie est ignorée.

QUELQUES OPERATEURS

Il existe plusieurs opérateurs en Langage C ; à ceux présentés ci-dessous, il faudrait ajouter les opérateurs bits-à-bits (que nous ne verrons pas en cours) ainsi que les opérateurs sur tableaux, pointeurs et structures (que nous verrons plus loin).

1- opérateurs arithmétiques

+	addition
-	soustraction (ou changement de signe)
*	multiplication
/	division (entière si les deux opérandes sont entiers, flottante sinon)
%	modulo (reste de la division entière)
++	incrémentation
	++i; (pré-incrémentation) : incrémentation (augmentation de 1) puis utilisation de i
	i++; (post-incrémentation) : utilisation de i puis incrémentation
--	décrémentation

2- opérateurs logiques

Il n'a pas de type booléen en C. Toute expression sera interprétée comme vraie si sa valeur est différente de 0 (c'est-à-dire 1, 36, -1963, …). Toute expression sera interprétée comme fausse si sa valeur est 0.

2.1- comparateurs

>	strictement supérieur
<	strictement inférieur
>=	supérieur ou égal
<=	inférieur ou égal
==	égalité
!=	différence

2.2- logiques

!	NON logique
&&	ET logique
||	OU logique

2.3- expression conditionnelle

condition ? expression1 : expression2 signifie expression1 si condition est vraie et expression2 si condition est fausse

par exemple : le minimum, min, de deux expressions a et b peut s'écrire : min = a<b ? a : b ;

3- opérateur d'affectation

= peut s'utiliser sous deux formes :

variable = expression1 opérateur expression2

variable opérateur = expression (° variable =variable opérateur expression)

(exemple : x+=5 ° x=x+5)

(ce qui le plus proche de la logique humaine, on ajoute 5 à x …)

LES CONVERSIONS DE TYPES

Le langage C permet d'effectuer des opérations de conversion de type : On utilise pour cela l'opérateur de "cast" ().

Exemple:

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

void main()
{
 int i=0x1234,j;
 char d,e;
 float r=89.67,s;
 j = (int)r;
 s = (float)i;
 d = (char)i;
 e = (char)r;
 printf("Conversion float -> int: %5.2f -> %d\n",r,j);
 printf("Conversion int -> float: %d -> %5.2f\n",i,s);
 printf("Conversion int -> char: %x -> %x\n",i,d);
 printf("Conversion float -> char: %5.2f -> %d\n",r,e);
 printf("Pour sortir frapper une touche "); getch();
}
```

Commenter les résultats obtenus.

LES STRUCTURES DE CONTROLE

1- instruction, bloc

Une **instruction** en C se termine toujours par un point-virgule (;). Les expressions (constantes, variables) peuvent être combinées ensemble à l'aide d'opérateur, pour former des expressions plus complexes ; elles peuvent contenir des appels à des fonctions, et aussi apparaître comme paramètres dans des appels de fonctions.

Différentes instructions peuvent être rassemblées en **bloc** à l'aide d'accolades { et }

Remarque : lorsqu'un bloc ne contient qu'une et une seule instruction alors les accolades ne sont plus obligatoires.

2- sélection

2.1- if (test)

```
if ( condition )
        {
        instruction1;
```

```
        instruction2;
        }
else
        {
        instruction3;
        instruction4;
        }
```

Si la condition est vraie (c'est-à-dire sa valeur différente de 0) alors les *instruction1* et *instruction2* sont réalisées, si la condition est fausse alors les *instruction3* et *instruction4* sont réalisées.

La négation de la condition else est totalement facultative ; si elle existe, la clause else se rapporte au dernier if ouvert :

```
if ( condition1 )
        if ( condition2 )
                instruction1;
        else
                instruction2;
```

```
if ( condition1 )
        {
        if ( condition2 )
                instruction1;
        else
                instruction2;
        }
```

si après une instruction else , un autre if est ouvert, on peut écrire else if

```
if ( condition1 )
        instruction1;
else
        {
        if ( condition2 )
                instruction2;
        else
                instruction3;
        }
```

```
if ( condition1 )
        instruction1;
else if ( condition2 )
        instruction2;
else
        instruction3;
```

2.2- switch (test à choix multiple)

```
switch ( expression )
{
  case  constante1 :
                instruction1.1;
                instruction1.2;
  case  constante2 :
                instruction2;
  case  constante3 :
  case  constante4 :
                instruction4.1;
                instruction4.2;
  default :
        instructionD;
}
```

L’expression est évaluée puis comparée aux différentes valeurs de la liste qui sont des constantes ; si la valeur de l’expression est trouvée dans la liste ou si l’option default est présente, l’exécution commence à partir de la liste d’instructions et continue ensuite jusqu’à la fin du bloc switch ; l’option default est facultative.

3- itérations

3.1- while (boucle tant que)

```
while ( condition )
        {
        instruction1;
        instruction2;
        }
```

les deux instructions s’exécuteront tant que la condition sera vraie

3.2- do ... while (boucle jusqu’à ce que)

```
do
        {
        instruction1;
        instruction2;
        } while ( condition );
```

les deux instructions s’exécuteront une fois puis autant de fois que la condition sera vraie

3.3- for (boucle avec compteur)

```
for ( initialisation ; condition ; in(dé)crémentation )
        {
        instruction1;
        instruction2;
        }
```

l’initialisation est exécutée avant d’entrer dans la boucle, la condition est évaluée à chaque tour de boucle, l’in(dé)crémentation est effectuée après les instructions et avant de remonter à la condition

exemples :

```
int  i, factoriel=1, n=10;
for ( i=1; i<=n ; i++ )
{
  factoriel*=i ;
}

int  i, factoriel=1, n=10;
for ( i=n; i>0 ; i-- )
{
```

```
factoriel*=i ;

}
```

4- ruptures avec break (échappement de boucle ou de test à choix multiple)

break permet d'arrêter prématurément une et une seule instruction itérative (while, do ... while, for) ou une sélection sur choix (switch) avant la fin normale

```
while ( condition1 )
        {
        instruction1;
        if ( condition2 )
          {
          instruction2;
          break;
          }
        instruction3;
        }
```

dès que la condition2 est vraie, on effectue l'instruction2 et on sort de la boucle while quelque soit l'état de la condition1

exemple :

```
char  c;
printf( ″Taper une lettre :″);
c=getch() ;
switch ( c )
        {
        case       'a' :
        case       'e' :
        case       'i' :
        case       'o' :
        case       'u' :
        case       'y' :
          printf("voyelles\n");
          break;
        default :
          printf("consonnes\n");
        }
```

Exercices

Exercice 0 :

Ecrire ces instructions dans un programme :

```
int i=2;
float x;
x = 5/i;
```

Afficher à l'écran la valeur de x. Voyez vous quelque chose qui vous semble étrange ?
Si vous changez « x=5/I » dans votre programme par :

- x = 5.0/i;
- x = 5./i;
- x = 5/(float)i;

Que se passe-t'il ?

Exercice 1 :
Ecrire un programme qui déclare la variable constante ***pi*** et la variable ***r***.
Déclarer trois variables ***D***, ***P*** et ***S*** et affecter respectivement à ces variables les valeurs du diamètre, du périmètre et de la surface d'un cercle dont le rayon est ***r***. On affichera à l'écran le contenu de ces différentes variables selon le format suivant :
Un cercle de rayon r a pour diamètre D, pour circonférence P et pour surface S.
NB : Le rayon est une entrée du programme (entrer au clavier par l'utilisateur).

Exercice 2 :
Ecrire un programme qui affiche la moyenne d'une suite d'entiers positifs entrés au clavier.
On arrêtera la saisie quand le nombre -1 est entré, comme dans l'exemple suivant :

```
      Entrez un entier positif : 5
      Entrez un entier positif : 2
      Entrez un entier positif : 3
      Entrez un entier positif : -1
La moyenne de ces 3 entiers vaut 3.333333
```

Exercice 3 :
Ecrire des programmes permettant de trouver le minimum, le maximum ainsi que la valeur médiane de 3 entiers.

Exercice 4 :
Ecrire un programme permettant de résoudre une équation de second degré $ax^2+bx+c=0$ dont les coefficients sont entrés au clavier.

Exercice 5 :

a. Ecrire un programme permettant de calculer *n!*

b. Ecrire un programme qui calcule x^n (où x est un réel et n un entier)

Exercice 6 :
Créer un programme qui calcule le nombre de combinaisons d'un ensemble de n pièces prises m par m:

$$\binom{m}{n} = \frac{n!}{m! \cdot (n-m)!} = \frac{\prod_{i=0}^{m-1} n-i}{\prod_{i=1}^{m} i}$$

Exercice 7 :
Ecrire un programme qui affiche un triangle rempli d'étoiles, s'étendant sur un nombre de lignes entré au clavier, comme dans l'exemple suivant :

```
Nombre de lignes = 5
*
**
***
****
*****
```

Exercice 8 :

Ecrire un programme qui donne le numéro N de la suite de Fibonacci définie par :

N → Fibo(N) = 0 if N=0;
Fibo(N) = 1 if N=1;
Fibo(N) = Fibo(N-2) + Fibo(N-1), if N≥2.

Exercice 9 :
Ecrire une fonction Max qui reçoit deux entiers a et b et retourne le maximum d'entre eux.
Dans le programme principal :

- Saisir deux entiers x et y.
- Calculer et afficher les maximum de x et y.

Ecrire la même fonction Max, mais en utilisant l'opérateur #define.

#define **……..**(a, b) ((a > b) **….** a : b))

Exercice 10 :
En utilisant l'opérateur #define, définir les fonction suivantes :

- Signe(x) qui retourne -1 si x est strictement négative et 1 sinon.
- Abs(x) qui retourne la valeur absolue de x.
- Min(x,y) qui retourne le minimum de x et y.

Les mettre en œuvre dans le programme principal.

ANNEXES

COMPLEMENTS AU LANGAGE C

DIRECTIVES DE COMPILATION

Les directives de compilation (ou commandes du préprocesseur) permettent :

- de réaliser des substitutions de symboles,
- d'inclure automatiquement des fichiers "texte",
- de permettre la compilation conditionnelle.

Les directives de compilation sont toujours introduites par la #.

Substitution de symboles (#define)

Elle permet de substituer un symbole par un autre ou par une valeur constante.

Ansi, le programme suivant :

```
#define    NB         10                    /* NB remplace la valeur
10   */
#define    DEBUT      {                     /* DEBUT est équivalent à
'}' */
#define    FIN        }                     /* FIN est équivalent à
'{'   */
#define    ENTIER     int                   /*   le   type   int   est
renommé    */
#define    SOMXY      X + Y                 /*  synonyme  d'expression
X+Y  */
#define    MSG        "Bonjour !\n"         /*  chaine  de  caractères
*/

void main()
DEBUT
      ENTIER X, Y, Z;

      X = 5; Y = NB;

      printf(MSG);

      Z = SOMXY;
      X = 2 * SOMXY + NB;
FIN
```

est équivalent, après compilation, au code :

```
void main()
{
      int X, Y, Z;

      X = 5; Y = 10;

      printf("Bonjour !\n");

      Z = X + Y;
      X = 2 * X + Y + 10;
}
```

on remarque ici que les expressions (apparemment équivalentes)

```
#define    SOMXY      X + Y
```

et

```
#define    SOMXY      (X + Y)
```

ne donnent pas le même résultat pour la variable X. L'utilisation des () est donc une question importante quant au sens de l'expression recherchée.

Les macros

Une macro n'est autre qu'une définition de symbole de substitution dans laquelle on utilise des arguments (dont le type n'est pas précisé).

Exemple :

```
#define    PRODUIT(X,Y)          X * Y

void main()
{
      int i, j, k;
      float a, b, c, x;

      k = PRODUIT(i,j);
      c = PRODUIT(a,b);

      x = PRODUIT(5+i,b-a);
}
```

Là encore, la bonne utilisation des parenthèses est un problème important, En effet, dans l'exemple donné la macro PRODUIT donne les résultats suivants :

```
k = i * j;            /* soit (i*j)        */
c = a * b;            /* soit (a*b)        */

x = 5 + i * b - a;    /* soit (5+(i*b)-a) */
```

si les expressions de K et de c sont celles attendues, il n'en est pas de même pour x, on a en effet : x=(5+(i*b)-a) au lieu de x=((5+i)*(b-a)) comme attendu.

Pour y remédier, l'emploie des parenthèses est indispensable pour lever toute ambiguïté. Ainsi la macro PRODUIT est redéclarée de manière suivante :

```
#define    PRODUIT(X,Y)          ((X) * (Y))
```

Compilation conditionnelle

Les directives suivantes

```
#if defined
#ifdef
#ifndef
#if
#else
#elif
#endif
#undef
```

permettent d'incorporer ou d'exclure de la compilation des portions de codes d'un programme selon que l'évaluation de la condition donne 1 ou 0.

```
#define SYS1                        /* SYS1 est défini (par defaut = 1)
*/

void main()
{
     #ifdef SYS1                    /*  ce  bloc  est  compilé  si  SYS1
défini */
          . . .                     /*
*/
          . . .                     /*
*/
     #endif                         /*
*/

     #ifdef SYS1                    /*  ce  bloc  est  compilé  si  SYS1
défini */
          . . .                     /*
*/
          . . .                     /*
*/
     #else                          /* ou ce bloc dans le cas contraire
*/
          . . .                     /*
*/
          . . .                     /*
*/
     #endif                         /*
*/

     #ifndef SYS2                   /* bloc compilé si SYS2 non défini
*/
          . . .                     /*
*/
          . . .                     /*
*/
     #endif                         /*
*/

     #ifndef SYS2                   /* bloc compilé si SYS2 non défini
*/
          #define SYS2              /*              SYS2 est alors défini
*/
          #undef  SYS1              /*               et SYS1 devient non
défini */
     #endif                         /*
*/

     . . .
}
```

<u>Inclusion automatique de texte (#include)</u>

La directive de compilation #include permet d'inclure un fichier dans le corps du programme. On distingue deux écritures possibles :

```
#include "nom_fichier"          /* insère un fichier d'abord
recherché  */
                                /* dans le répertoire courant puis
dans */
                                /* les répertoires spécifiés dans
*/
                                /* l'option "Include directories"
*/


#include <nom_fichier>          /* insère un fichier recherché dans
les */
                                /* répertoires spécifiés dans
l'option  */
                                /* "Include directories"
*/
```

POINTEURS DE FONCTIONS

Les pointeurs permettent non seulement d'adresser une variable mais aussi toute fonction. Par exemple :

```
void fonct1(void)
{
     printf("Fonction n°1\n");
}

void fonct2(void)
{
     printf("Fonction n°2\n");
}

void fonct3(int val)
{
     printf("Fonction n°3 : val = %d\n", val);
}

int fonct4(int val)
{
     val++;
     printf("Fonction n°4 : val = %d\n", val);
     return val;
}


void main()
{
     int x = 0, y;         /* deux variables de type int
*/

     void (*g)();          /* pointeur sur une fonction sans
argument  */
                           /* en entrée et ne retournant rien
*/

     void (*h)(int);       /* pointeur sur une fonction qui accepte
```

```
un */
                              /* un entier et ne retournant rien *
/

      int (*k)(int);          /* pointeur sur une fonction qui accepte
un */
                              /* un entier et retourne un entier
*/


      /* différents appels possibles */

      g = fonct1;             /* g pointe sur la fonction fonct1() */
      (*g)();                 /* la fonction est exécutée          */

      g = fonct2;             /* g pointe maintenant sur fonct2()  */
      (*g)();                 /* la fonction est exécutée          */

      h = fonct3;             /* h pointe sur la fonction fonct3() */
      (*h)(x);                /* la fonction est exécutée          */

      k = fonct4;             /* k pointe sur la fonction fonct4() */
      y = (*k)(x);            /* la fonction est exécutée          */
}
```

TABLE DES CARACTERES ASCII

Car.	Code	Car.	Code	Car.	Code	Car.	Code	Car.	Code	Car.	Code
NUL	0	2	50	d	100	–	150	È	200	ú	250
	1	3	51	e	101	—	151	É	201	û	251
STX	2	4	52	f	102	˜	152	Ê	202	ü	252
ETX	3	5	53	g	103	™	153	Ë	203	ý	253
EOT	4	6	54	h	104	š	154	Ì	204	þ	254
	5	7	55	i	105	›	155	Í	205	ÿ	255
ACK	6	8	56	j	106	œ	156	Î	206		
BEL	7	9	57	k	107	•	157	Ï	207		
	8	:	58	l	108	ž	158	Ð	208		
	9	;	59	m	109	Ÿ	159	Ñ	209		
LF	10	<	60	n	110	espace	160	Ò	210		
	11	=	61	o	111	¡	161	Ó	211		
	12	>	62	p	112	¢	162	Ô	212		
CR	13	?	63	q	113	£	163	Õ	213		
	14	@	64	r	114	¤	164	Ö	214		
	15	A	65	s	115	¥	165	×	215		
	16	B	66	t	116	¦	166	Ø	216		
	17	C	67	u	117	§	167	Ù	217		
	18	D	68	v	118	¨	168	Ú	218		
	19	E	69	w	119	©	169	Û	219		
NAK	20	F	70	x	120	ª	170	Ü	220		
	21	G	71	y	121	"	171	Ý	221		
	22	H	72	z	122	¬	172	Þ	222		
	23	I	73	{	123	-	173	ß	223		
	24	J	74	|	124	®	174	à	224		
	25	K	75	}	125	¯	175	á	225		
	26	L	76	~	126	°	176	â	226		
	27	M	77	suppr	127	±	177	ã	227		
	28	N	78	€	128	²	178	ä	228		
	29	O	79	•	129	³	179	å	229		
	30	P	80	‚	130	´	180	æ	230		
	31	Q	81	ƒ	131	µ	181	ç	231		
Espace	32	R	82	„	132	¶	182	è	232		
!	33	S	83	…	133	·	183	é	233		
"	34	T	84	†	134	¸	184	ê	234		
#	35	U	85	‡	135	¹	185	ë	235		
$	36	V	86	ˆ	136	º	186	ì	236		
%	37	W	87	‰	137	"	187	í	237		
&	38	X	88	Š	138	¼	188	î	238		
'	39	Y	89	‹	139	½	189	ï	239		
(	40	Z	90	Œ	140	¾	190	ð	240		
)	41	[	91	•	141	¿	191	ñ	241		
*	42	\	92	Ž	142	À	192	ò	242		
+	43	]	93	•	143	Á	193	ó	243		
,	44	^	94	•	144	Â	194	ô	244		
-	45	_	95	'	145	Ã	195	õ	245		
.	46	`	96	'	146	Ä	196	ö	246		
/	47	a	97	"	147	Å	197	÷	247		
0	48	b	98	"	148	Æ	198	ø	248		
1	49	c	99	•	149	Ç	199	ù	249		

HOW TO USE DEV-C++

First steps: “hello world”.

How to use Bloodshed Dev C++ to compile and execute C programs, is explained in this section.

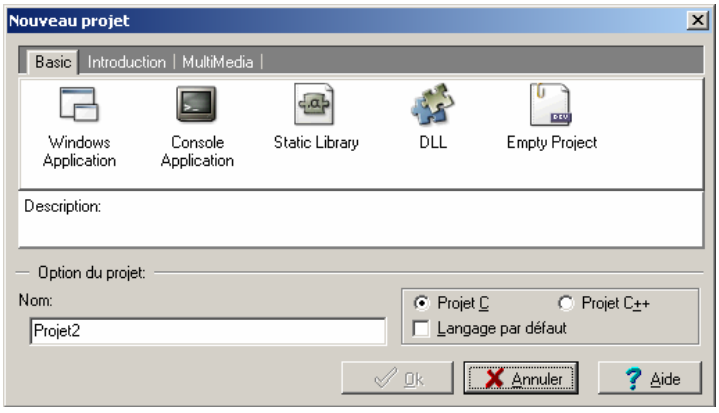
Step 0: Run Dev-C++.
Run Dev-C++ and follow the next steps.

Step 1: Configure Dev-C++.
We need to tell Dev-C++ where we plan to save files and modify one of the default settings.

- Go to the "`Options`" menu and select "`Compiler Options`".
- In the "`Directories`" tab, put check in the "`Add the directory below to be searched for include files:`" option, and enter the location(s) of where you plan to save your files. *Do not use directory names with spaces and make sure to separate multiple entries with semicolons.*
  As an example, mine is set to: `C:\PRG;C:\PRG\L3`.
- In the "`Linker`" tab, put check in the "`Generate debugging information`" option. This will allow you to use the debugger with your programs.

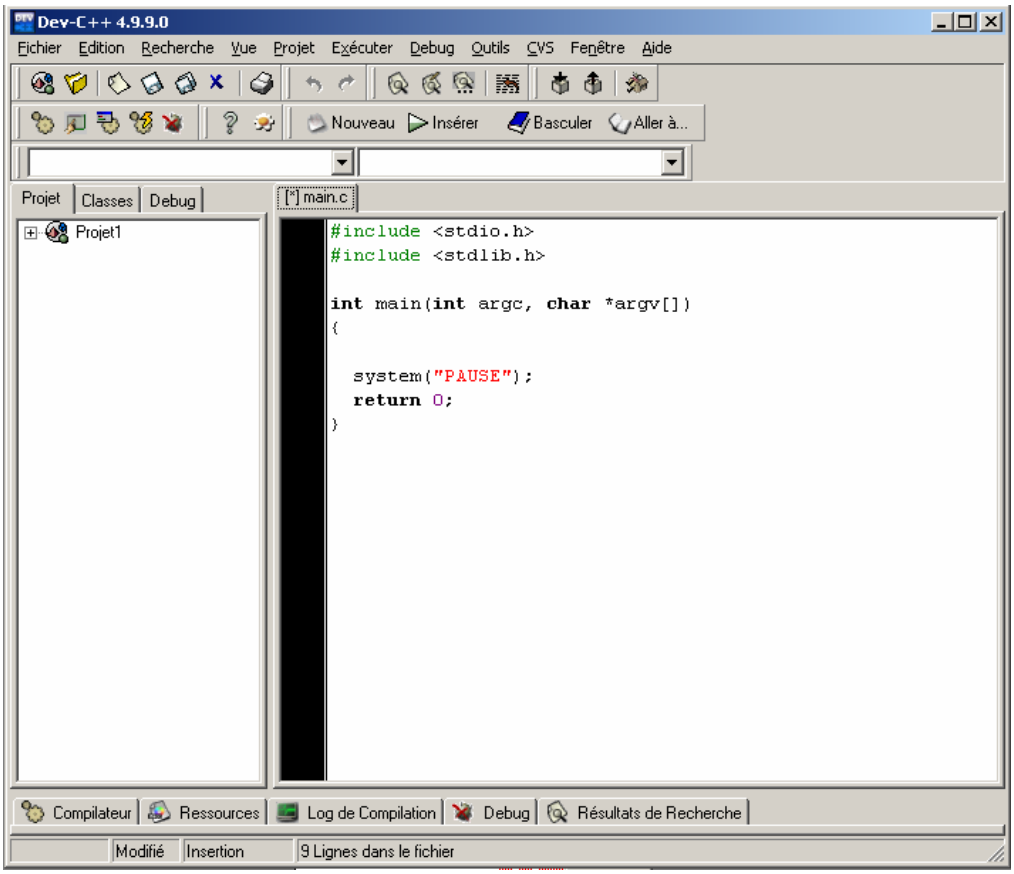
Step 2: Create a new project.
A "project" can be considered as a container that is used to store all the elements that are required to compile a program.

- Go to the "`File`" menu and select "`New Project...`" (or just press CTRL+N).



- Choose "`Console Application`", make sure "`C project`" is selected, and click "`OK`".
- At this point, Dev-C++ will ask you to give your project a name (like « projet1.dev ») in a directory...
  You can give your project any valid filename, but keep in mind that the name of your project will also be the name of your executable.

Step 3: Create source file(s).
You should now have a project with one empty and untitled source file. This would be sufficient if we were writing simple programs that relied exclusively on the standard library and/or precompiled object code.



Step 4: Compile.
Once you have entered all of your source code, you are ready to compile.

- Go to the "`Execute`" menu and select "`Compile`" (or just press CTRL+F9).

  It is likely that you will get some kind of compiler or linker error the first time you attempt to compile a project. Syntax errors will be displayed in the "`Compiler`" tab at the bottom of the screen. You can double-click on any error to take you to the place in the source code where it occurred. The "`Linker`" tab will flash if there are any linker errors. Linker errors are generally the result of syntax errors not allowing one of the files to compile.

Step 5: Execute.
Once your project successfully compiles, a dialog box appears with several options:

- "Continue", which will just take you back to Dev-C++.
- "Parameters", which will allow you to pass command-line parameters to your program.
- "Execute", which will execute your program.

Disappearing windows
If you execute your program (with or without parameters), you may notice something; a

console window will pop up, flash some text and disappear. The problem is that, if directly executed, console program windows close after the program exits.
You can solve this problem using MS-DOS command-prompt: use system(“pause”); at the end of program;
Instead of using Dev-C++ to invoke your program, you can just open an MS-DOS Prompt, go to the directory where your program was compiled (i.e. where you saved the project) and enter the program name (along with any parameters). The command-prompt window will not close when the program terminates.

Step 6: First Program : « Hello World ».

Use « printf » command to print « Hello World » on console window.