INF 102: PROGRAMMATION DE BASE

PINDRA Nadjime

18 mai 2020

TABLE DES MATIÈRES

CHAPITRE 1

INTRODUCTION

Sommaire

1.1 Historique

Le C a été conçu en 1972 par Dennis Ritchie et Ken Thompson, chercheurs aux Bell Labs, afin de développer un système d'exploitation UNIX sur un DEC PDP-11. En 1978, Brian Kernighan et Dennis Richie publient la définition classique du C dans le livre The C Programming language.

Le C devenant de plus en plus populaire dans les années 80, plusieurs groupes mirent sur le marché des compilateurs comportant des extensions particulières. En 1983, l'ANSI (American National Standards Institute) décida de normaliser le langage ; ce travail s'acheva en 1989 par la définition de la norme ANSI C. Celle-ci fut reprise telle quelle par l'ISO (International Standards Organization) en 1990. C'est ce standard, ANSI C, qui est décrit dans le présent document.

Un programme C est un **ensemble d'instructions** qui se saisit dans un **fichier .c** à l'aide d'un éditeur (ex. : BlocNote), ce type de fichier s'appelle une **source**. Les instructions qui y sont écrites s'appellent

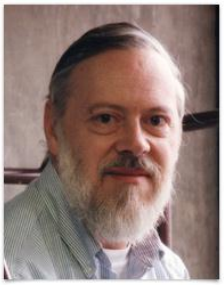


FIGURE 1.1 – Dennis Ritchie

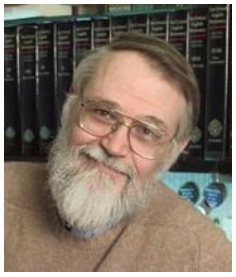


FIGURE 1.2 – Ken Thompson

1.2 Création d'un programme en langage C

La manière de développer et d'utiliser un programme en langage C dépend naturellement de l'environnement de programmation dans lequel vous travaillez. Nous vous fournissons ici quelques indications générales (s'appliquant à n'importe quel environnement) concernant ce que l'on pourrait appeler les grandes étapes de la création d'un programme, à savoir : édition du programme, compilation et édition de liens.

1.2.1 L'édition du programme

L'édition du programme (on dit aussi parfois « saisie ») consiste à créer, à partir d'un clavier, tout ou partie du texte d'un programme qu'on nomme « programme source ». les fichiers source porteront l'extension C.

1.2.2 La compilation

Elle consiste à traduire le programme source (ou le contenu d'un fichier source) en langage machine, en faisant appel à un programme nommé compilateur. En langage C, compte tenu de l'existence d'un préprocesseur, cette opération de compilation comporte en fait deux étapes :

- **traitement par le préprocesseur** : ce dernier exécute simplement les directives qui le concernent (il les reconnaît au fait qu'elles commencent par un caractère #). Il produit, en résultat, un programme source en langage C pur. Notez bien qu'il s'agit toujours d'un vrai texte, au même titre qu'un programme source : la plupart des environnements de programmation vous permettent d'ailleurs, si vous le souhaitez, de connaître le résultat fourni par le préprocesseur.

- **compilation** proprement dite, c'est-à-dire traduction en langage machine du texte en langage C fourni par le préprocesseur.
  Le résultat de la compilation porte le nom de module objet.

1.2.3 L'édition des liens

Le module objet créé par le compilateur n'est pas directement exécutable. Il lui manque, au moins, les différents modules objet correspondant aux fonctions prédéfinies (on dit aussi « fonctions standard ») utilisées par votre programme (comme printf , scanf , sqrt).
C'est effectivement le rôle de l'éditeur de liens que d'aller rechercher dans la bibliothèque standard les modules objet nécessaires. Notez que cette bibliothèque est une collection de modules objet organisée, suivant l'implémentation concernée, en un ou plusieurs fichiers.
Le résultat de l'édition de liens est ce que l'on nomme un programme exécutable, c'est-à-dire un ensemble autonome d'instructions en langage machine. Si ce programme exécutable est rangé dans un fichier, il pourra ultérieurement être exécuté sans qu'il soit nécessaire de faire appel à un quelconque composant de l'environnement de programmation en C. Certains environnements de développement servent d'éditeur, et prennent en charge la compilation et l'édition de liens **(codeblocks, netbeans, eclipse, dev-C++)**.[1]

1.3 Environnement de développement : Codeblocks [3]

Voir l'annexe *A* [3]

CHAPITRE 2

GÉNÉRALITÉS SUR LE LANGAGE C

Sommaire

Dans ce chapitre, nous vous proposons une première approche d'un programme en langage C, basée sur deux exemples commentés. Vous y découvrirez comment s'expriment les instructions de base (déclaration, affectation, lecture et écriture), ainsi que deux des structures fondamentales (boucle avec compteur, choix). Nous dégagerons ensuite quelques règles générales concernant l'écriture d'un programme. Enfin, nous vous montrerons comment s'organise le développement d'un programme en vous rappelant ce que sont l'édition, la compilation, l'édition de liens et l'exécution.

2.1 Un exemple de programme en langage C

Voici un exemple de programme en langage C, accompagné d'un exemple d'exécution. Avant d'en lire les explications qui suivent, essayez d'en percevoir plus ou moins le fonction-

nement.

Programme C :

```
#include <stdio.h>
#include <math.h>
#define NFOIS 5
main()
{ int i ;
float x ;
float racx ;
printf ("Bonjour\n") ;
printf ("Je vais vous calculer %d racines carrées\n", NFOIS) ;
for (i=0 ; i<NFOIS ; i++)
{ printf ("Donnez un nombre : ") ;
scanf ("%f", &x) ;
if (x < 0.0)
printf ("Le nombre %f ne possède pas de racine carrée\n", x) ;
else
{ racx = sqrt (x) ;
printf ("Le nombre %f a pour racine carrée : %f\n", x, racx) ;
}
}
printf ("Travail terminé - Au revoir") ;
}
```

Après exécution

```
Bonjour
Je vais vous calculer 5 racines carrées
Donnez un nombre : 4
Le nombre 4.000000 a pour racine carrée : 2.000000
Donnez un nombre : 2
Le nombre 2.000000 a pour racine carrée : 1.414214
Donnez un nombre : -3
Le nombre -3.000000 ne possède pas de racine carrée
Donnez un nombre : 5.8
Le nombre 5.800000 a pour racine carrée : 2.408319
Donnez un nombre : 12.58
Le nombre 12.580000 a pour racine carrée : 3.546829
Travail terminé - Au revoir
```

2.2 Structure d'un programme en langage C

2.2.1 Les directives à destination du préprocesseur

Les trois premières lignes de notre programme :

```
#include <stdio.h>
#include <math.h>
#define NFOIS 5
```

sont appelées des **directives**. Ces directives seront prises en compte avant la traduction (compilation) du programme. Contrairement au reste du programme, elles doivent être écrites à raison d'une par ligne et doivent obligatoirement commencer en début de ligne.
Les deux premières directives

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

demandent en fait d'introduire (avant compilation) des instrucions (en langage C) situées dans les fichiers stdio.h et math.h .
il est nécessaire d'incorporer de tels fichiers, nommés « fichiers en-têtes », qui contiennent des déclarations appropriées concernant cette fonction : stdio.h pour printf et scanf , math.h pour sqrt .
La troisième directive

```
#define NFOIS 5
```

demande simplement de remplacer systématiquement, dans toute la suite du programme, le symbole NFOIS par 5 . Autrement dit, le programme qui sera réellement compilé comportera ces instrutions :

```
printf ("Je vais vous calculer %d racines carrées\n", 5) ;
for (i=0 ; i<5 ; i++)
```

2.2.2 Programme principal

La ligne

```
main()
```

se nomme un « en-tête ». Elle précise que ce qui sera décrit à sa suite est en fait le programme principal (main).
Le programme (principal) proprement dit vient à la suite de cet en-tête. Il est délimité par les accolades « { » et « } ». On dit que les instructions situées entre ces accolades forment un « bloc ». Ainsi peut-on dire que la fonction main est constituée d'un en-tête et d'un bloc.

2.2.3 Déclaration

Les trois instructions :

```
int i ;
float x ;
float racx ;
```

sont des **déclarations**.
La première déclaration *int i* précise que la variable nommée i est de type int , c'est-à-dire qu'elle est destinée à contenir des nombres entiers (relatifs). Nous verrons qu'en C il existe plusieurs types d'entiers.
Les deux autres déclarations *float x* et *float racx* précisent que les variables x et racx sont de type float , c'est- à-dire qu'elles sont destinées à contenir des nombres flottants (approximation de nombres réels). Là encore, nous verrons qu'en C il existe plusieurs types flottants.
En C, **les déclarations des types des variables sont obligatoires et doivent être regroupées au début du programme**.

2.2.4 Pour écrire des informations : la fonction printf

L'instruction :

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

appelle en fait une fonction prédéfinie (fournie avec le langage, et donc que vous n'avez pas à écrire vous-même) nommée printf . Ici, cette fonction reçoit un argument qui est : "Bonjour $\backslash n$" Les guillemets servent à délimiter une « chaîne de caractères » (suite de caractères). La notation $\backslash n$ est conventionnelle : elle représente un caractère de fin de ligne, c'est-à-dire un caractère qui, lorsqu'il est envoyé à l'écran, provoque le passage à la ligne suivante.
L'instruction suivante :

```
printf ("Je vais vous calculer %d racines carrées\n", NFOIS) ;
```

ressemble à la précédente avec cette différence qu'ici la fonction printf reçoit deux arguments. Pour comprendre son fonctionnement, il faut savoir qu'en fait le premier argument de printf est ce que l'on nomme un « format »; il s'agit d'une sorte de guide qui précise comment afficher les informations qui sont fournies par les arguments suivants.
Ici, on demande à printf d'afficher suivant ce format : "Je vais vous calculer %d racines carrées $\backslash n$" la valeur de NFOIS , c'est-à-dire, la valeur 5 .

2.2.5 Pour lire des informations : la fonction scanf

L'instruction :

```
scanf ("%f", &x) ;
```

est un appel de la fonction prédéfinie scanf dont le rôle est de lire une information au clavier. Comme printf , la fonction scanf possède en premier argument un format exprimé sous forme d'une chaîne de caractères, ici : %f qui correspond à une valeur flottante.

2.2.6 Pour faire une répétition : l'instruction for

Comme nous le verrons, en langage C, il existe plusieurs façons de réaliser une répétition (on dit aussi une « boucle »). Ici, nous avons utilisé l'instruction for :

```
for (i=0 ; i<NFOIS ; i++)
```

Son rôle est de répéter le bloc (délimité par des accolades « { » et « } ») figurant à sa suite, en respectant les consignes suivantes :

- avant de commencer cette répétition, réaliser :
  $i = 0$
- avant chaque nouvelle exécution du bloc (tour de boucle), examiner la condition :
  $i < NFOIS$
  si elle est satisfaite, exécuter le bloc indiqué, sinon passer à l'instruction suivant ce bloc : à la fin de chaque exécution du bloc, réaliser :
  $i++$
  Il s'agit là d'une notation propre au langage C qui est équivalente à :
  $i = i + 1$

2.2.7 Pour faire des choix : l'instruction if

Les lignes :

```
if (x < 0.0)
printf ("Le nombre %f ne possède pas de racine carrée\n", x) ;
else
{ racx = sqrt (x) ;
printf ("Le nombre %f a pour racine carrée : %f\n", x, racx) ;
}
```

constituent une instruction de choix basée sur la condition $x < 0.0$. Si cette condition est vraie, on exécute l'instruction suivante, c'est-à-dire :

```
printf ("Le nombre \%f ne possède pas de racine carrée\n", x) ;
```

Si elle est fausse, on exécute l'instruction suivant le mot else , c'est-à-dire, ici, le bloc :

```
{ racx = sqrt (x) ;
printf ("Le nombre \%f a pour racine carrée : %f\n", x, racx) ;
}
```

Notez qu'il existe un mot *else* mais pas de mot *then* . La syntaxe de l'instruction if (notamment grâce à la présence de parenthèses qui encadrent la condition) le rend inutile.

2.3 Quelques règles d'écriture

Ce paragraphe expose un certain nombre de règles générales intervenant dans l'écriture d'un programme en langage C. Nous aborderons les éléments suivants :

- les identificateurs
- les mots-clés
- le format libre
- les séparateurs
- les commentaires

2.3.1 Les identificateurs

Les identificateurs servent à désigner les différents « objets » manipulés par le programme. Le rôle d'un identificateur est de donner un nom à une entité du programme. Plus précisément, un identificateur peut désigner :

— un nom de variable ou de fonction,
— un type défini par typedef, struct, union ou enum,
— une étiquette.

Un identificateur est une suite de caractères parmi :

— les lettres (minuscules ou majuscules, mais non accentuées),
— les chiffres,
— le "blanc souligné" (_).

2.3.2 Les mots-clés

Un certain nombre de mots, appelés mots-clefs, sont réservés pour le langage lui-même et ne peuvent pas être utilisés comme identificateurs. L'ANSI C compte 32 mots clés :

auto	const	double	float	int	short	struct	unsigned
break	continue	else	for	long	signed	switch	void
case	default	enum	goto	register	sizeof	typedef	volatile
char	do	extern	if	return	satitic	union	while

que l'on peut ranger en catégories :

- les spécificateurs de stockage
  auto register static extern typedef
- les spécificateurs de type
  char double enum float int long short signed struct union unsigned void
- les qualificateurs de type
  const volatile
- les instructions de contrôle
  break case continue default do else for goto if switch while

- divers
  return sizeof

2.3.3 Les séparateurs

Dans un programme, deux identificateurs successifs entre lesquels la syntaxe n'impose aucun signe particulier (tel que : , = ; * ( ) [ ] { }) doivent impérativement être séparés soit par un espace, soit par une fin de ligne. En revanche, dès que la syntaxe impose un séparateur quelconque, il n'est alors pas nécessaire de prévoir d'espaces supplémentaires (bien qu'en pratique cela améliore la lisibilité du programme). Ainsi, vous devrez impérativement écrire :

- int x,y
  et non :
- intx,y
  En revanche, vous pourrez écrire indifféremment :
- int n,compte,total,p
  ou plus lisiblement :
- int n, compte, total, p

2.3.4 Le format libre

Le langage C autorise une mise en page parfaitement libre. En particulier, une instruction peut s'étendre sur un nombre quelconque de lignes, et une même ligne peut comporter autant d'instructions que vous le souhaitez. Les fins de ligne ne jouent pas de rôle particulier, si ce n'est celui de séparateur, au même titre qu'un espace, sauf dans les « constantes chaînes » où elles sont interdites ; de telles constantes doivent impérativement être écrites à l'intérieur d'une seule ligne. Un identificateur ne peut être coupé en deux par une fin de ligne, ce qui semble évident.

2.3.5 Les commentaires

Comme tout langage évolué, le langage C autorise la présence de commentaires dans vos programmes source. Il s'agit de textes explicatifs destinés aux lecteurs du programme et qui n'ont aucune incidence sur sa compilation.
Ils sont formés de caractères quelconques placés entre les symboles /* et */ . Ils peuvent apparaître à tout endroit du programme où un espace est autorisé. En général, cependant, on se limitera à des emplacements propices à une bonne lisibilité du programme.
Voici quelques exemples de commentaires :

```
] /* programme de calcul de racines carrées */
/* commentaire fantaisiste &ç§{<>} ?%!!!!!! */
/* commentaire s'étendant
sur plusieurs lignes
de programme source
```

```
*/
/* ============================================
*
commentaire quelque peu esthétique
*
*
et encadré, pouvant servir,
*
*
par exemple, d'en-tête de programme
*
============================================ */
```

CHAPITRE 3

LES TYPES DE BASE DU LANGAGE C

Sommaire

Les types char , int et float que nous avons déjà rencontrés sont souvent dits « scalaires » ou « simples », car, à un instant donné, une variable d'un tel type contient une seule valeur. Ils s'opposent aux types « structurés » (on dit aussi « agrégés ») qui correspondent à des variables qui, à un instant donné, contiennent plusieurs valeurs (de même type ou non). Ici, nous étudierons en détail ce que l'on appelle les types de base du langage C.

3.1 La notion de type

La mémoire centrale est un ensemble de positions binaires nommées bits. Les bits sont regroupés en octets (8 bits), et chaque octet est repéré par ce qu'on nomme son adresse.
L'ordinateur, compte tenu de sa technologie actuelle, ne sait représenter et traiter que des informations exprimées sous forme binaire. Toute information, quelle que soit sa nature, devra être codée sous cette forme.
Par exemple l'octet 01001101 peut représenter l'entier 77 ou le caractère M dans le cas du code ASCII. Il peut également représenté une partie d'une instruction ou d'un nombre entier

codé sur 2 octets, ou d'un nombre réel codé sur 4 octets.
On comprend donc qu'il n'est pas possible d'attribuer une signification à une information binaire tant que l'on ne connaît pas la manière dont elle a été codée.
La notion de type sert à régler les problèmes que nous venons d'évoquer.
Les types de base du langage C se répartissent en trois grandes catégories en fonction de la nature des informations qu'ils permettent de représenter :

— **nombres entiers** (mot-clé int),
— **nombres flottants** (mot-clé float ou double),
— **caractères** (mot-clé char) ; nous verrons qu'en fait char apparaît (en C) comme un cas particulier de int.

3.2 Les types entiers

3.2.1 Leur représentation en mémoire

Le mot-clé int correspond à la représentation de nombres entiers relatifs. Pour ce faire : un bit est réservé pour représenter le signe du nombre (en général 0 correspond à un nombre positif) ; les autres bits servent à représenter la valeur absolue du nombre.

3.2.2 Les différents types d'entiers

Le C prévoit que, sur une machine donnée, on puisse trouver jusqu'à trois tailles différentes d'entiers, désignées par les mots-clés suivants :

— short int (qu'on peut abréger en short),
— int (c'est celui que nous avons rencontré dans le chapitre précédent),
— long int (qu'on peut abréger en long).

Chaque taille impose naturellement ses limites. Toutefois, ces dernières dépendent, non seulement du mot-clé considéré, mais également de la machine utilisée.
À titre indicatif, avec 16 bits, on représente des entiers s'étendant de -32 768 à 32 767 ; avec 32 bits, on peut couvrir les valeurs allant de -2 147 483 648 à 2 147 483 647 .

3.2.3 Notation des constantes entières

La façon la plus naturelle d'introduire une constante entière dans un programme est de l'écrire simplement sous forme décimale, avec ou sans signe, comme dans ces exemples :

$$+533, \qquad 48, \qquad -273$$

3.3 Les types flottants

3.3.1 Les différents types et leur représentation en mémoire

Les types flottants permettent de représenter, de manière approchée, une partie des nombres réels. Pour ce faire, ils s'inspirent de la notation scientifique (ou exponentielle) bien connue

qui consiste à écrire un nombre sous la forme $1.5\,10^{22}$ ou $0.472\,10^{-8}$.
Plus précisément, un nombre réel sera représenté en flottant en déterminant deux quantités M (mantisse) et E (exposant) telles que la valeur

$$M.B^E$$

représente une approximation de ce nombre. La base B est généralement unique pour une machine donnée (il s'agit souvent de 2 ou de 16) et elle ne figure pas explicitement dans la représentation machine du nombre.
Le C prévoit trois types de flottants correspondant à des tailles différentes :

— float
— double
— long double

3.3.2 Notation des constantes flottantes

Comme dans la plupart des langages, les constantes flottantes peuvent s'écrire indifféremment suivant l'une des deux notations :

- décimale,
- exponentielle.

La notation décimale doit comporter obligatoirement un point (correspondant à notre virgule). La partie entière ou la partie décimale peut être omise (mais, bien sûr, pas toutes les deux en même temps !). En voici quelques exemples corrects :

$$12.43, \qquad -0.38, \qquad -.38, \qquad 4., \qquad .27$$

En revanche, la constante 47 serait considérée comme entière et non comme flottante. Dans la pratique, ce fait aura peu d'importance, si ce n'est au niveau du temps d'exécution, compte tenu des conversions automatiques qui seront mises en place par le compilateur (et dont nous parlerons dans le chapitre suivant). La notation exponentielle utilise la lettre e (ou E) pour introduire un exposant entier (puis- sance de 10), avec ou sans signe.

$4.25E4 \quad 4.25e{+}4 \quad 42.5E3 \quad 54.27E{-}32 \quad 542.7E{-}33 \quad 5427e{-}34 \quad 48e13 \quad 48.e13 \quad 48.0E13$

Par défaut, toutes les constantes sont créées par le compilateur dans le type double . Il est toutefois possible d'imposer à une constante flottante :

- d'être du type float , en faisant suivre son écriture de la lettre F (ou f).
- d'être du type long double , en faisant suivre son écriture de la lettre L (ou l).

3.4 Les types caractères

3.4.1 La notion de caractère en langage C

Comme la plupart des langages, C permet de manipuler des caractères codés en mémoire sur un octet.

Le mot-clé char désigne un objet de type caractère. Un char peut contenir n'importe quel élément du jeu de caractères de la machine utilisée.
Une des particularités du type char en C est qu'il peut être assimilé à un entier : tout objet de type char peut être utilisé dans une expression qui utilise des objets de type entier. Par exemple, si c est de type char, l'expression $c + 1$ est valide. Elle désigne le caractère suivant dans le code ASCII. Ainsi, le programme suivant imprime le caractère 'B'.

```
main()
{
char c = 'A';
printf("%c", c + 1);
}
```

3.4.2 Notation des constantes caractères

Les constantes de type « caractère », lorsqu'elles correspondent à des caractères imprimables, se notent de façon classique, en écrivant entre apostrophes (ou quotes) le caractère voulu, comme dans ces exemples :

```
'a'  'Y'   '+'   '$'
```

Certains caractères non imprimables possèdent une représentation conventionnelle utilisant le caractère « \ », nommé « antislash ».

Voici la liste de ces caractères :

Notation en C	Signification
\ a	cloche ou bip
\ b	retour arrière
\ f	saut de page
\ n	saut de ligne
\ r	retour chariot
\ t	tabulation horizontal
\ v	tabulation verticale

3.4.3 Initialisations et constantes

1. Nous avons déjà vu que la directive # define permettait de donner une valeur à un symbole. Dans ce cas, le préprocesseur effectue le remplacement correspondant avant la compilation.
2. Par ailleurs, il est possible d'initialiser une variable lors de sa déclaration comme dans : int n = 15 ;
   Ici, pour le compilateur, n est une variable de type int dans laquelle il placera la valeur 15.
3. Il est possible de déclarer que la valeur d'une variable ne doit pas changer lors de l'exécution du programme. Par exemple, avec :
   const int n = 20 ;

on déclare n de type int et de valeur (initiale) 20 mais, de surcroît, les éventuelles instructions modifiant la valeur de n seront rejetées par le compilateur. On pourrait penser qu'une déclaration (par const) remplace avantageusement l'emploi de define. En fait, nous verrons que les choses ne sont pas aussi simples, car les variables ainsi déclarées ne pourront pas intervenir dans ce qu'on appelle des « expressions constantes » (notamment, elles ne pourront pas servir de dimension d'un tableau !).

CHAPITRE 4

LES OPÉRATEURS ET LES EXPRESSIONS EN LANGAGE C

Sommaire

Le langage C est certainement l'un des langages les plus fournis en opérateurs. Cette richesse se manifeste tout d'abord au niveau des opérateurs classiques (arithmétiques, relationnels, logiques) ou moins classiques (manipulations de bits). Mais, de surcroît, le C dispose d'un important éventail d'opérateurs originaux d'affectation et d'incrémentation. Ce dernier aspect nécessite une explication. En effet, dans la plupart des langages, on trouve, comme en C :

- d'une part, des expressions formées (entre autres) à l'aide d'opérateurs,
- d'autre part, des instructions pouvant éventuellement faire intervenir des expressions, comme, par exemple, l'instruction d'affectation :
  y = a * x +b ;
  ou encore l'instruction d'affichage :
  printf ("valeur %d", n + 2*p) ;
  dans laquelle apparaît l'expression n + 2 * p .

4.1 L'affectation

En C, l'affectation est un opérateur à part entière. Elle est symbolisée par le signe =. Sa syntaxe est la suivante :
variable = expression
Le terme de gauche de l'affectation peut être une variable simple, un élément de tableau mais pas une constante. Cette expression a pour effet d'évaluer expression et d'affecter la valeur obtenue à variable. De plus, cette expression possède une valeur, qui est celle expression. Ainsi, l'expression i = 5 vaut 5.
L'affectation effectue une conversion de type implicite : la valeur de l'expression (terme de droite) est convertie dans le type du terme de gauche. Par exemple, le programme suivant

```
main()
{
int i, j = 2;
float x = 2.5;
i = j + x;
x = x + i;
printf("\n %f \n",x);
}
```

imprime pour x la valeur 6.5 (et non 7), car dans l'instruction i = j + x ;, l'expression j + x a été convertie en entier.

4.1.1 Notion de Lvalue

Ce terme désigne une « valeur à gauche », c'est-à-dire tout ce qui peut apparaître à gauche d'un opérateur d'affectation.
Certes, pour l'instant, vous pouvez trouver que dire qu'à gauche d'un opérateur d'affectation doit apparaître une lvalue n'apporte aucune information. En fait, d'une part, nous verrons qu'en C d'autres opérateurs que = font intervenir une lvalue ; d'autre part, au fur et à mesure que nous rencontrerons de nouveaux types d'objets, nous préciserons s'ils peuvent être ou non utilisés comme lvalue.
Pour l'instant, les seules lvalue que nous connaissons restent les variables de n'importe quel type de base déjà rencontré.

4.1.2 L'opérateur d'affectation possède une associativité de droite à gauche

Contrairement à tous ceux que nous avons rencontrés jusqu'ici, cet opérateur d'affectation possède une associativité de droite à gauche. C'est ce qui permet à une expression telle que : i = j = 5 d'évaluer d'abord l'expression j = 5 avant d'en affecter la valeur (5) à la variable j . Bien entendu, la valeur finale de cette expression est celle de i après affectation, c'est-à-dire 5 .

4.2 Les opérateurs arithmétiques

Les opérateurs arithmétiques classiques sont l'opérateur unaire - (changement de signe) ainsi que les opérateurs binaires

```
+ addition
- soustraction
* multiplication
/ division
% reste de la division (modulo)
```

Ces opérateurs agissent de la façon attendue sur les entiers comme sur les flottants. Leurs seules spécificités sont les suivantes :

- Contrairement à d'autres langages, le C ne dispose que de la notation / pour désigner à la fois la division entière et la division entre flottants. Si les deux opérandes sont de type entier, l'opérateur / produira une division entière (quotient de la division). Par contre, il délivrera une valeur flottante dès que l'un des opérandes est un flottant. Par exemple,
  ```
  float x;
  x = 3 / 2;
  affecte à x la valeur 1. Par contre
  x = 3 / 2.;
  affecte à x la valeur 1.5.
  ```
- L'opérateur % ne s'applique qu'à des opérandes de type entier. Si l'un des deux opérandes est négatif, le signe du reste dépend de l'implémentation, mais il est en général le même que celui du dividende.

Notons enfin qu'il n'y a pas en C d'opérateur effectuant l'élévation à la puissance. De façon générale, il faut utiliser la fonction pow(x,y) de la librairie math.h pour calculer x^y.

4.3 Les opérateurs relationnels

```
> strictement supérieur
>= supérieur ou égal
< strictement inférieur
<= inférieur ou égal
== égal
!= différent
```

Leur syntaxe est :

```
expression-1 op expression-2
```

Les deux expressions sont évaluées puis comparées. La valeur rendue est de type int (il n'y a pas de type booléen en C) ; elle vaut 1 si la condition est vraie, et 0 sinon. Attention à ne pas confondre l'opérateur de test d'égalité == avec l'opérateur d'affection =. Ainsi, le programme

```
main()
{
int a = 0;
int b = 1;
if (a = b)
printf("\n a et b sont egaux \n");
else
printf("\n a et b sont differents \n");
}
imprime à l'écran a et b sont egaux !
```

4.4 Les opérateurs de logiques booléens

```
&& et logique
|| ou logique
! négation logique
```

Comme pour les opérateurs de comparaison, la valeur retournée par ces opérateurs est un int qui vaut 1 si la condition est vraie et 0 sinon. Dans une expression de type

```
expression-1 op-1 expression-2 op-2 ...expression-n
```

l'évaluation se fait de gauche à droite et s'arrête dès que le résultat final est déterminé. Par exemple dans

```
int i;
int p[10];
if ((i >= 0) && (i <= 9) && !(p[i] == 0))
...
la dernière clause ne sera pas évaluée si i n'est pas entre 0 et 9.
```

4.5 Les opérateurs d'affectation composée

Les opérateurs d'affectation composée sont

```
+=   -=  *=  /=   %=  &=  ^=  |=  <<=  >>=

Pour tout opérateur op, l'expression
expression-1 op= expression-2
est équivalente à
expression-1 = expression-1 op expression-2

Toutefois, avec l'affection composée, expression-1 n'est évaluée qu'une seule fois.
```

4.6 Les opérateurs d'incrémentation et de décrémentation

Dans des programmes écrits dans un langage autre que C, on rencontre souvent des expressions (ou des instructions) telles que :

```
i = i + 1
n = n - 1
```

qui incrémentent ou qui décrémentent de 1 la valeur d'une variable (ou plus généralement d'une lvalue).
En C, ces actions peuvent être réalisées par des opérateurs « unaires » portant sur cette lvalue. Ainsi, l'expression :

```
++i
```

a pour effet d'incrémenter de 1 la valeur de i , et sa valeur est celle de i après incrémentation. Là encore, comme pour l'affectation, nous avons affaire à une expression qui non seulement possède une valeur, mais qui, de surcroît, réalise une action (incrémentation de i).
Il est important de voir que la valeur de cette expression est celle de i après incrémentation.

```
Ainsi, si la valeur de i est 5 , l'expression :
n = ++i - 5
affectera à i la valeur 6 et à n la valeur 1 .
```

En revanche, lorsque cet opérateur est placé après la lvalue sur laquelle il porte, la valeur de l'expression correspondante est celle de la variable avant incrémentation.

```
Ainsi, si i vaut 5 , l'expression :
n = i++ - 5
affectera à i la valeur 6 et à n la valeur 0
 (car ici la valeur de l'expression i++ est 5 ).
```

On dit que ++ est :

- un opérateur de préincrémentation lorsqu'il est placé à gauche de la lvalue sur laquelle il porte,
- un opérateur de postincrémentation lorsqu'il est placé à droite de la lvalue sur laquelle il porte.

Bien entendu, lorsque seul importe l'effet d'incrémentation d'une lvalue, cet opérateur peut être indifféremment placé avant ou après. Ainsi, ces deux instructions (ici, il s'agit bien d'instructions car les expressions sont terminées par un point-virgule, leur valeur se trouve donc inutilisée) sont équivalentes :

```
i++ ;
++i ;
```

De la même manière, il existe un opérateur de décrémentation noté −− qui, suivant les cas, sera :

- un opérateur de prédécrémentation lorsqu'il est placé à gauche de la lvalue sur laquelle il porte,
- un opérateur de postdécrémentation lorsqu'il est placé à droite de la lvalue sur laquelle il porte.

4.7 L'opérateur virgule

Une expression peut être constituée d'une suite d'expressions séparées par des virgules
expression-1, expression-2, ... , expression-n
Cette expression est alors évaluée de gauche à droite. Sa valeur sera la valeur de
l'expression de droite. Par exemple, le programme

```
main()
{
int a, b;
b = ((a = 3), (a + 2));
printf("\n b = %d \n",b);
}
```

imprime b = 5.
La virgule séparant les arguments d'une fonction ou les déclarations de variables n'est pas l'opérateur virgule.

4.8 L'opérateur conditionnel ternaire

L'opérateur conditionnel ? est un opérateur ternaire. Sa syntaxe est la suivante :
condition ? expression-1 : expression-2
Cette expression est égale à expression-1 si condition est satisfaite, et à
 expression-2
sinon. Par exemple, l'expression

```
x >= 0 ? x : -x
```

correspond à la valeur absolue d'un nombre. De même l'instruction

```
m = ((a > b) ? a : b);
```

affecte à m le maximum de a et de b.

4.9 L'opérateur de conversion de type

L'opérateur de conversion de type, appelé cast, permet de modifier explicitement
le type
d'un objet. On écrit

```
(type) objet
```

Par exemple,

```
main()
{
```

```
int i = 3, j = 2;
printf("%f \n",(float)i/j);
}
retourne la valeur 1.5.
```

4.10 L'opérateur adresse

L'opérateur d'adresse & appliqué à une variable retourne l'adresse-mémoire de cette variable. La syntaxe est:

```
&objet
```

4.11 L'opérateur sizeof

L'opérateur sizeof , dont l'emploi ressemble à celui d'une fonction, fournit la taille en octets (n'oubliez pas que l'octet est, en fait, la plus petite partie adressable de la mémoire). Par exemple, dans une implémentation où le type int est représenté sur 2 octets et le type double sur 8 octets, si l'on suppose que l'on a affaire à ces déclarations :

```
int n ;
double z ;
 l'expression sizeof(n) vaudra 2 ,
 l'expression sizeof(z) vaudra 8 .
```

Cet opérateur peut également s'appliquer à un type de nom donné. Ainsi, dans l'implémentation précédemment citée :

```
 sizeof(int) vaudra 2 ,
 sizeof(double) vaudra 8 .
```

Quelle que soit l'implémentation, sizeof(char) vaudra toujours 1 (par définition, en quelque sorte). Cet opérateur offre un intérêt :

- lorsque l'on souhaite écrire des programmes portables dans lesquels il est nécessaire de connaître la taille exacte de certains objets,
- pour éviter d'avoir à calculer soi-même la taille d'objets d'un type relativement complexe pour lequel on n'est pas certain de la manière dont il sera implémenté par le compilateur. Ce sera notamment le cas des structures.

Exercice :

1) Soit les déclarations suivantes :

```
int n = 10 , p = 4 ;
long q = 2 ;
float x = 1.75 ;
```

Donner le type et la valeur de chacune des expressions suivantes :

(a) $n + q$

(b) $n + x$

(c) $n\%p + q$

(d) $n < p$

(e) $n \geqslant p$

(f) $n > q$

(g) $q + 3 * (n > p)$

(h) $q\&\&n$

(i) $(q - 2)\&\&(n - 10)$

(j) $x * (q == 2)$

(k) $x * (q = 5)$

2) Écrire plus simplement l'instruction suivante :

$$z = (a > b?a : b) + (a <= b?a : b)$$

3) n étant de type int , écrire une expression qui prend la valeur :

```
-1 si n est négatif,
0 si n est nul,
1 si n est positif.
```

CHAPITRE 5

LES ENTRÉES-SORTIES CONVERSATIONNELLES

Sommaire

Jusqu'ici, nous avons utilisé de façon intuitive les fonctions printf et scanf pour afficher des informations à l'écran ou pour en lire au clavier. Nous vous proposons maintenant d'étudier en détail les différentes possibilités de ces fonctions, ce qui nous permettra de répondre à des questions telles que :

- quelles sont les écritures autorisées pour des nombres fournis en données ? Que se passe-t-il lorsque l'utilisateur ne les respecte pas ?
- comment organiser les données lorsque l'on mélange les types numériques et les types caractères ?
- que se produit-il lorsque, en réponse à scanf , on fournit trop ou trop peu d'informations ?
- comment agir sur la présentation des informations à l'écran ?

5.1 Les possibilités de la fonction printf

Nous avons déjà vu que le premier argument de printf est une chaîne de caractères qui spécifie à la fois :

- des caractères à afficher tels quels,
- des codes de format repérés par %. Un code de conversion (tel que c , d ou f) y précise le type de l'information à afficher.

5.1.1 Les principaux codes de conversion

- **%c** char : caractère
- **%d** int : entier (convient aussi à char)
- **%f** double ou float : (compte tenu des conversions systématiques float→double) écrit en notation décimale avec six chiffres après le point (par exemple : 1.234500 ou 123.456789)
- **%e** double ou float (compte tenu des conversions systématiques float→double) écrit en notation exponentielle (mantisse entre 1 inclus et 10 exclu) avec six chiffres après le point décimal, sous la forme x.xxxxxxe+yyy ou x.xxxxxx-yyy pour les nombres positifs et -x.xxxxxxe+yyy ou -x.xxxxxxe-yyy pour les nombres négatifs
- **%s** chaîne de caractères dont on fournit l'adresse (notion qui sera étudiée ultérieurement)

5.1.2 La fonction d'écriture printf

La fonction printf est une fonction d'impression formatée, ce qui signifie que les données sont converties selon le format particulier choisi. Sa syntaxe est

```
printf("chaîne de contrôle ",expression-1, ..., expression-n);
```

La chaîne de contrôle contient le texte à afficher et les spécifications de format correspondant à chaque expression de la liste. Les spécifications de format ont pour but d'annoncer le format des données à visualiser. Elles sont introduites par le caractère %, suivi d'un caractère désignant le format d'impression. Les formats d'impression en C sont donnés à la table 1.5. En plus du caractère donnant le type des données, on peut éventuellemnt préciser certains paramètres du format d'impression, qui sont spécifiés entre le % et le caractère de conversion dans l'ordre suivant :

- largeur minimale du champ d'impression : %10d spécifie qu'au moins 10 caractères seront réservés pour imprimer l'entier.
- précision : %.12f signifie qu'un flottant sera imprimé avec 12 chiffres après la virgule. De même %10.2f signifie que l'on réserve 12 caractères (incluant le caractère .) pour imprimer le flottant et que 2 d'entre eux sont destinés aux chiffres après la virgule. Lorsque la précision n'est pas spécifiée, elle correspond par défaut à 6 chiffres après la virgule. Pour une chaîne de caractères, la précision correspond au nombre de caractères imprimés : %30.4s signifie que l'on réserve un champ de 30 caractères pour imprimer la chaîne mais que seulement les 4 premiers caractères seront imprimés (suivis de 26 blancs).

Exemple :

```
#include <stdio.h>
main()
{
int i = 23674;
int j = -23674;
long int k = (1l << 32);
```

```
double x = 1e-8 + 1000;
char c = 'A';
char *chaine = "chaine de caracteres";
printf("impression de i: \n");
printf("%d \t %u \t %o \t %x",i,i,i,i);
printf("\nimpression de j: \n");
printf("%d \t %u \t %o \t %x",j,j,j,j);
printf("\nimpression de k: \n");
printf("%d \t %o \t %x",k,k,k);
printf("\n%ld \t %lu \t %lo \t %lx",k,k,k,k);
printf("\nimpression de x: \n");
printf("%f \t %e \t %g",x,x,x);
printf("\n%.2f \t %.2e",x,x);
printf("\n%.20f \t %.20e",x,x);
printf("\nimpression de c: \n");
printf("%c \t %d",c,c);
printf("\nimpression de chaine: \n");
printf("%s \t %.10s",chaine,chaine);
printf("\n");
}
Ce programme imprime à l'écran :
impression de i:
23674 23674 56172 5c7a
impression de j:
-23674 4294943622 37777721606 ffffa386
impression de k:
0   0   0
4294967296 4294967296 40000000000 100000000
impression de x:
1000.000000 1.000000e+03 1000
1000.00 1.00e+03
1000.00000001000000000000 1.000000000001000000000e+03
impression de c:
A      65
impression de chaine:
chaine de caracteres chaine de
```

5.2 La fonction de saisie scanf

Nous avons déjà rencontré quelques exemples d'appels de scanf . Nous y avons notamment vu la nécessité de recourir à l'opérateur & pour désigner l'adresse de la variable (plus généralement de la lvalue) pour laquelle on souhaite lire une valeur.

La fonction scanf permet de saisir des données au clavier et de les stocker aux adresses spécifiées par les arguments de la fonctions.
Exemple :

```
scanf("chaîne de contrôle",argument-1,...,argument-n)
```

La chaîne de contrôle indique le format dans lequel les données lues sont converties. Elle ne contient pas d'autres caractères (notamment pas de
n). Comme pour printf, les conversions de format sont spécifiées par un caractère précédé du signe %. Les formats valides pour la fonction scanf diffèrent légèrement de ceux de la fonction printf.
Les données à entrer au clavier doivent être séparées par des blancs ou des ¡RETURN¿ sauf s'il s'agit de caractères. On peut toutefois fixer le nombre de caractères de la donnée à lire. Par exemple %3s pour une chaîne de 3 caractères, %10d pour un entier qui s'étend sur 10 chiffres, signe inclus. Exemple :

```
#include <stdio.h>
main()
{
int i;
printf("entrez un entier sous forme hexadecimale i = ");
scanf("%x",&i);
printf("i = %d\n",i);
}
Si on entre au clavier la valeur 1a, le programme affiche i = 26.
```

5.3 Impression et lecture de caractères

Les fonctions getchar et putchar permettent respectivement de lire et d'imprimer des caractères. Il s'agit de fonctions d'entrées-sorties non formatées.

```
L'expression :
putchar (c)
joue le même rôle que :
printf ("%c", c)
```

La fonction getchar retourne un int correspondant au caractère lu. Pour mettre le caractère lu dans une variable caractère, on écrit
caractere = getchar() ;

```
L'expression :
c = getchar()
joue le même rôle que :
scanf ("%c", &c)
```

L'exécution de putchar et getchar est plus rapide puisque ne faisant pas appel au mécanisme d'analyse d'un format. En toute rigueur, getchar est une macro (comme putchar) dont les instructions figurent dans stdio.h . Là encore, l'omission d'une instruction #include appropriée conduit à une erreur à l'édition de liens.

Exercice :

1) Quels seront les résultats fournis par ce programme ?

```
#include <stdio.h>
main ()
{
int n = 543 ;
int p = 5 ;
float x = 34.5678;
printf ("A : %d %f\n", n, x) ;
printf ("B : %4d %10f\n", n, x) ;
printf ("C : %2d %3f\n", n, x) ;
printf ("D : %10.3f %10.3e\n", x, x) ;
printf ("E : %*d\n", p, n) ;
printf ("F : %*.*f\n", 12, 5, x) ;
}
```

2) Saisir deux variables et les permuter avant de les afficher.

3) Écrire un programme demandant à de la façon suivante :

Noms des variables	A	B	C	D
Valeurs avant la permutation	1	2	3	4
valeurs avant la permutation	3	4	1	2

4) Affectez le caractère 'a' à une variable de type char, affichez ce caractère ainsi que son code ASCII.

5) Ecrivez un programme qui saisit un caractère miniscule et qui l'affiche en majuscule.

6) Une adresse IP est constituée de 4 valeurs de 0 à 255 séparées par des points, par exemple 192.168.0.1, chacun de ces nombres peut se coder sur 1 octet. Comme une variable de type long occupe 4 octets en mémoire, il est possible de s'en servir pour stocker une adresse IP entière. Ecrivez un programme qui saisit dans des unsigned short les 4 valeurs d'une adresse IP, et place chacune d'elle sur un octet d'une variable de type long. Ensuite vous extrairez de cet entier les 4 nombres de l'adresse IP et les afficherez en les séparant par des points.

CHAPITRE 6

LES INSTRUCTIONS DE CONTRÔLE

Sommaire

On appelle instruction de contrôle toute instruction qui permet de contrôler le fonctionnement d'un programme. Parmi les instructions de contrôle, on distingue les *instructions de branchement et les boucles*.
A priori, dans un programme, les instructions sont exécutées séquentiellement, c'est-à-dire dans l'ordre où elles apparaissent. Or la puissance et le « comportement intelligent » d'un programme proviennent essentiellement :

- de la possibilité d'effectuer des choix, de se comporter différemment suivant les circonstances (celles-ci pouvant être, par exemple, une réponse de l'utilisateur, un résultat de calcul...),
- de la possibilité d'effectuer des boucles, autrement dit de répéter plusieurs fois un ensemble donné d'instructions.

6.1 Bloc d'instructions

Un bloc est une suite d'instructions placées entre { et } . Les instructions figurant dans un bloc sont absolument quelconques. Il peut s'agir aussi bien d'instructions simples (terminées

par un point-virgule) que d'instructions structurées (choix, boucles) lesquelles peuvent alors à leur tour renfermer d'autres blocs.

6.2 Instruction *if*

La syntaxe est la suivante :

```
if (expression)
     instruction_1
else
     instruction_2
```

La forme la plus générale est celle-ci :

```
if ( expression-1 )
instruction-1
else if ( expression-2 )
instruction-2
...
else if ( expression-n )
instruction-n
else
instruction
```

avec un nombre quelconque de else if (...). Le dernier else est toujours facultatif. La forme la plus simple est

```
if ( expression )
   instruction
```

Chaque instruction peut être un bloc d'instructions.
Exemple :

```
#define TAUX_TVA 18.6
main()
{
double ht, ttc, net, tauxr, remise ;
printf("donnez le prix hors taxes : ") ;
scanf ("%f", &ht) ;
ttc = ht * ( 1. + TAUX_TVA/100.) ;
if ( ttc < 1000.)
tauxr=0;
   else if ( ttc < 2000 )
    tauxr=1
      else if ( ttc < 5000 )
```

```
        tauxr=3
 else
 tauxr=5

remise = ttc * tauxr / 100. ;
net = ttc - remise ;
printf ("prix ttc %10.2f\n", ttc) ;
printf ("remise  %10.2f\n", remise) ;
printf ("net à payer %10.2f\n", net) ;
}
```

6.3 Instruction *switch*

Sa forme la plus générale est celle-ci :

```
switch ( expression )
{
case constante-1:
liste d'instructions 1
break;
case constante-2:
liste d'instructions 2
break;
...
case constante-n:
liste d'instructions n
break;
default:
liste d'instructions
break;
}
```

Si la valeur de expression est égale à l'une des constantes, la liste d'instructions correspondant est exécutée. Sinon la liste d'instructions correspondant à default est exécutée. L'instruction default est facultative.
Exemple :

```
main()
{
int n ;
printf ("donnez un entier : ") ;
scanf ("%d", &n) ;
switch (n)
```

```
{ case 0 : printf ("nul\n") ;
break ;
case 1 : printf ("un\n") ;
break ;
case 2 : printf ("deux\n") ;
break ;
default : printf ("grand\n") ;
}
printf ("au revoir\n") ;
}
```

6.4 Les boucles

Les boucles permettent de répéter une série d'instructions tant qu'une certaine condition n'est pas vérifiée.

6.4.1 Boucle *while*

La synthaxe de while est la suivante :

```
while ( expression )
instruction
```

Tant que expression est vérifiée (i.e., non nulle), instruction est exécutée. Si expression est nulle au départ, instruction ne sera jamais exécutée. instruction peut évidemment être une instruction composée. Par exemple, le programme suivant imprime les entiers de 1 à 9.

```
i = 1;
while (i < 10)
{
printf("\n i = %d",i);
i++;
}
```

6.4.2 Boucle *do—while*

Il peut arriver que l'on ne veuille effectuer le test de continuation qu'après avoir exécuté l'instruction. Dans ce cas, on utilise la boucle do—while. Sa syntaxe est

```
do
instruction
while ( expression );
```

Ici, instruction sera exécutée tant que expression est non nulle. Cela signifie donc que instruction est toujours exécutée au moins une fois. Par exemple, pour saisir au clavier un entier entre 1 et 10 :

```
int a;
do
{
printf("\n Entrez un entier entre 1 et 10 : ");
scanf("%d",&a);
}
while ((a <= 0) || (a > 10));
```

6.4.3 Boucle *for*

La syntaxe de for est :

```
for ( expr 1 ; expr 2 ; expr 3)
instruction
```

Une version équivalente plus intuitive est :

```
expr 1;
while ( expr 2 )
{
instruction
expr 3;
}
```

Par exemple, pour imprimer tous les entiers de 0 à 9, on écrit :

```
for (i = 0; i < 10; i++)
printf("\n i = %d",i);
```

A la fin de cette boucle, i vaudra 10. Les trois expressions utilisées dans une boucle for peuvent être constituées de plusieurs expressions séparées par des virgules. Cela permet par exemple de faire plusieurs initialisations à la fois. Par exemple, pour calculer la factorielle d'un entier, on peut écrire :

```
int n, i, fact;
for (i = 1, fact = 1; i <= n; i++)
fact *= i;
printf("%d ! = %d \n",n,fact);
On peut également insérer l'instruction fact *= i; dans la boucle for ce qui donne :
int n, i, fact;
for (i = 1, fact = 1; i <= n; fact *= i, i++);
printf("%d ! = %d \n",n,fact);
```

On évitera toutefois ce type d'acrobaties qui n'apportent rien et rendent le programme difficilement lisible.

6.5 Les instructions de branchement non conditionnel

6.5.1 L'instruction *break*

Elle permet d'interrompre le déroulement de la boucle, et passe à la première instruction qui suit la boucle. En cas de boucles imbriquées, break fait sortir de la boucle la plus interne. Par exemple, le programme suivant :

```
main()
{
int i;
for (i = 0; i < 5; i++)
{
printf("i = %d\n",i);
if (i == 3)
break;
}
printf("valeur de i a la sortie de la boucle = %d\n",i);
}
imprime à l'écran
i = 0
i = 1
i = 2
i = 3
valeur de i a la sortie de la boucle = 3
```

6.5.2 Instruction *continue*

L'instruction continue permet de passer directement au tour de boucle suivant, sans exécuter les autres instructions de la boucle. Ainsi le programme

```
main()
{
int i;
for (i = 0; i < 5; i++)
{
if (i == 3)
continue;
printf("i = %d\n",i);
}
printf("valeur de i a la sortie de la boucle = %d\n",i);
}
imprime
i = 0
i = 1
```

```
i = 2
i = 4
valeur de i a la sortie de la boucle = 5
```

6.5.3 L'instruction *goto*

L'instruction goto permet d'effectuer un saut jusqu'à l'instruction etiquette correspondant. Elle est à proscrire de tout programme C digne de ce nom.

Exercice :

1) Soit le petit programme suivant :

```
#include <stdio.h>
main()
{
int i, n, som ;
som = 0 ;
for (i=0 ; i<4 ; i++)
{ printf ("donnez un entier ") ;
scanf ("%d", &n) ;
som += n ;
}
printf ("Somme : %d\n", som) ;
}
```

Écrire un programme réalisant exactement la même chose, en employant, à la place de l'instruction for :
- une instruction while,
- une instruction do... while.

2) Calculer la moyenne de notes fournies au clavier avec un dialogue de ce type :

```
note 1 : 12
note 2 : 15.25
note 3 : 13.5
note 4 : 8.75
note 5 : -1
moyenne de ces 4 notes : 12.37
```

3) $ax^2 + bx + c = 0$
Saisir les coefficients a, b et c, afficher la solution de l'équation $ax^2 + bx + c = 0$

4) Calculer une approximation de e^x à l'aide de la série $e^x = \sum_{k=0}^{+\infty} \frac{x^k}{k!}$

5) Ecrire un programme qui saisit une valeur n et qui affiche le carré suivant ($n = 5$ dans l'exemple) :

```
n = 5
X X X X X
X X X X X
X X X X X
X X X X X
X X X X X
```

CHAPITRE 7

LES FONCTIONS

Sommaire

Comme dans la plupart des langages, on peut en C découper un programme en plusieurs fonctions. Une seule de ces fonctions existe obligatoirement ; c'est la fonction principale appelée main. Cette fonction principale peut, éventuellement, appeler une ou plusieurs fonctions secondaires. De même, chaque fonction secondaire peut appeler d'autres fonctions secondaires ou s'appeler elle-même (dans ce dernier cas, on dit que la fonction est récursive).

7.1 Définition d'une fonction

La définition d'une fonction est la donnée du texte de son algorithme, qu'on appelle corps de la fonction. Elle est de la forme

```
type nom-fonction ( type-1 arg-1,..., type-n arg-n)
{
[ déclarations de variables locales ]
liste d'instructions
}
```

La première ligne de cette définition est l'en-tête de la fonction. Dans cet en-tête, type désigne le type de la fonction, c'est-à-dire le type de la valeur qu'elle retourne. Contrairement à d'autres langages, il n'y a pas en C de notion de procédure ou de sous-programme. Une fonction qui ne

renvoie pas de valeur est une fonction dont le type est spécifié par le mot-clé void. Les arguments de la fonction sont appelés paramètres formels, par opposition aux paramètres effectifs qui sont les paramètres avec lesquels la fonction est effectivement appelée. Les paramètres formels peuvent être de n'importe quel type. Leurs identificateurs n'ont d'importance qu'à l'intérieur de la fonction. Enfin, si la fonction ne possède pas de paramètres, on remplace la liste de paramètres formels par le mot-clé void.
Le corps de la fonction débute éventuellement par des déclarations de variables, qui sont locales à cette fonction. Il se termine par l'instruction de retour à la fonction appelante, return, dont la syntaxe est :

```
return(expression);
```

La valeur de expression est la valeur que retourne la fonction. Son type doit être le même que celui qui a été spécifié dans l'en-tête de la fonction. Si la fonction ne retourne pas de valeur (fonction de type void), sa définition s'achève par

```
return;
```

7.2 Appel d'une fonction

L'appel d'une fonction se fait par l'expression :

```
nom-fonction(para-1,para-2,...,para-n)
```

L'ordre et le type des paramètres effectifs de la fonction doivent concorder avec ceux donnés dans l'en-tête de la fonction. Les paramètres effectifs peuvent être des expressions. La virgule qui sépare deux paramètres effectifs est un simple signe de ponctuation ; il ne s'agit pas de l'opérateur virgule. Cela implique en particulier que l'ordre d'évaluation des paramètres effectifs n'est pas assuré et dépend du compilateur.

7.3 Déclaration d'une fonction

Le C n'autorise pas les fonctions imbriquées. La définition d'une fonction secondaire doit donc être placée soit avant, soit après la fonction principale main. Toutefois, il est indispensable que le compilateur "connaisse" la fonction au moment où celle-ci est appelée. Si une fonction est définie après son premier appel (en particulier si sa définition est placée après la fonction main), elle doit impérativement être déclarée au préalable. Une fonction secondaire est déclarée par son prototype, qui donne le type de la fonction et celui de ses paramètres, sous la forme :

```
type nom-fonction(type-1,...,type-n);
```

Les fonctions secondaires peuvent être déclarées indifféremment avant ou au début de la fonction main. Par exemple, on écrira

```
int puissance (int, int );
int puissance (int a, int n)
{
if (n == 0)
return(1);
return(a * puissance(a, n-1));
}
main()
{
int a = 2, b = 5;
printf("%d\n", puissance(a,b));
}
```

7.4 Cas des fonctions sans valeur de retour ou sans arguments

Quand une fonction ne renvoie pas de résultat, on le précise, à la fois dans l'en-tête et dans sa déclaration, à l'aide du mot-clé void.
Par exemple, voici l'en-tête d'une fonction recevant un argument de type int et ne fournissant aucune valeur :

```
void sansval (int n)
et voici quelle serait sa déclaration :
void sansval (int) ;
```

Naturellement, la définition d'une telle fonction ne doit, en principe, contenir aucune instruction return.
Voici l'en-tête d'une fonction ne recevant aucun argument et renvoyant une valeur de type float (il pourrait s'agir, par exemple, d'une fonction fournissant un nombre aléatoire !) :

```
float tirage (void)
Sa déclaration serait très voisine
(elle ne diffère que par la présence du point-virgule !) :
float tirage (void) ;
```

Enfin, rien n'empêche de réaliser une fonction ne possédant ni arguments ni valeur de retour. Dans ce cas, son en-tête sera de la forme :

```
void message (void)
et sa déclaration sera :
void message (void) ;
```

Voici un exemple illustrant deux des situations évoquées. Nous y définissons une fonction $affiche_carres$ qui affiche les carrés des nombres entiers compris entre deux limites fournies en arguments et une fonction erreur qui se contente d'afficher un message d'erreur (il s'agit de notre premier exemple de programme source contenant plus de deux fonctions).

```
#include <stdio.h>
main()
{ void affiche_carres (int, int) ; /* prototype de affiche_carres */
void erreur (void) ; /* prototype de erreur */
int debut = 5, fin = 10 ;
.....
affiche_carres (debut, fin) ;
.....
if (...) erreur () ;
}
void affiche_carres (int d, int f)
{ int i ;
for (i=d ; i<=f ; i++)
printf ("%d a pour carré %d\n", i, i*i) ;
}
void erreur (void)
{ printf ("*** erreur ***\n") ; }
```

7.5 Durée de vie des variables

Les variables manipulées dans un programme C ne sont pas toutes traitées de la même manière. En particulier, elles n'ont pas toutes la même durée de vie. On distingue deux catégories de variables.

- Une variable permanente occupe un emplacement en mémoire qui reste le même durant toute l'exécution du programme. Cet emplacement est alloué une fois pour toutes lors de la compilation. La partie de la mémoire contenant les variables permanentes est appelée segment de données. Par défaut, les variables permanentes sont initialisées à zéro par le compilateur. Elles sont caractérisées par le mot-clé static.
- Les variables temporaires se voient allouer un emplacement en mémoire de façon dynamique lors de l'exécution du programme. Elles ne sont pas initialisées par défaut. Leur emplacement en mémoire est libéré par exemple à la fin de l'exécution d'une fonction secondaire.

7.5.1 Les variables globales

On appelle variable globale une variable déclarée en dehors de toute fonction. Une variable globale est connue du compilateur dans toute la portion de code qui suit sa déclaration. Les variables globales sont systématiquement permanentes. Dans le programme suivant, n est une variable globale :

```
int n;
void fonction();
void fonction()
```

```
{
n++;
printf("appel numero %d\n",n);
return;
}
main()
{
int i;
for (i = 0; i < 5; i++)
fonction();
}
```

La variable n est initialisée à zéro par le compilateur et il s'agit d'une variabl permanente.
En effet, le programme affiche

```
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

7.5.2 Les variables locales

On appelle variable locale une variable déclarée à l'intérieur d'une fonction (ou d'un bloc d'instructions) du programme. Par défaut, les variables locales sont temporaires. Quand une fonction est appelée, elle place ses variables locales dans la pile. A la sortie de la fonction, les variables locales sont dépilées et donc perdues. Les variables locales n'ont en particulier aucun lien avec des variables globales de même nom. Par exemple, le programme suivant

```
int n = 10;
void fonction();
void fonction()
{
int n = 0;
n++;
printf("appel numero %d\n",n);
return;
}
main()
{
int i;
for (i = 0; i < 5; i++)
fonction();
}
affiche
```

```
appel numero 1
appel numero 1
appel numero 1
appel numero 1
appel numero 1
```

Les variables locales à une fonction ont une durée de vie limitée à une seule exécution de cette fonction. Leurs valeurs ne sont pas conservées d'un appel au suivant.
Il est toutefois possible de créer une variable locale de classe statique en faisant précéder sa déclaration du mot-clef static : static type nom-de-variable ; Une telle variable reste locale à la fonction dans laquelle elle est déclarée, mais sa valeur est conservée d'un appel au suivant. Elle est également initialisée à zéro à la compilation. Par exemple, dans le programme suivant, n est une variable locale à la fonction secondaire fonction, mais de classe statique.

```
int n = 10;
void fonction();
void fonction()
{
static int n;
n++;
printf("appel numero %d\n",n);
return;
}
main()
{
int i;
for (i = 0; i < 5; i++)
fonction();
}
Ce programme affiche
appel numero 1
appel numero 2
appel numero 3
appel numero 4
appel numero 5
```

Exercice :

1) Écrire :
   - une fonction, nommée f1, se contentant d'afficher "bonjour" (elle ne possèdera aucun argument ni valeur de retour),
   - une fonction, nommée f2, qui affiche "bonjour" un nombre de fois égal à la valeur reçue en argument (int) et qui ne renvoie aucune valeur,

- une fonction, nommée f3, qui fait la même chose que f2 , mais qui, de plus, renvoie la valeur (int) 0.

Écrire un petit programme appelant successivement chacune de ces trois fonctions, après les avoir convenablement déclarées sous forme d'un prototype.

2) Qu'affiche le programme suivant ?

```
int n=5 ;
main()
{
void fct (int p) ;
int n=3 ;
fct(n) ;
}
void fct(int p)
{
printf("%d %d", n, p) ;
}
```

3) Écrire une fonction récursive calculant la valeur de la « fonction d'Ackermann » A définie pour $m > 0$ et $n > 0$ par :
$A(m, n) = A(m - 1, A(m, n - 1))$ pour $m > 0$ et $n > 0$
$A(0, n) = n + 1$ pour $n > 0$
$A(m, 0) = A(m - 1, 1)$ pour $m > 0$

4) Soient a et b deux entiers strictement positifs. a est un diviseur strict de b si a divise b et $a \neq b$. Par exemple, 3 est un diviseur strict de 6. Mais 6 n'est pas un diviseur strict de 6. a et b sont des nombres amis si la somme des diviseurs stricts de a est b et si la somme des diviseurs de b est a. Le plus petit couple de nombres amis connu est 220 et 284.
   a) Ecrire une fonction **int sommeDiviseursStricts(int n)**, elle doit renvoyer la somme des diviseurs stricts de n.
   b) Ecrire une fonction **int sontAmis(int a, int b)**, elle doit renvoyer 1 si a et b sont amis, 0 sinon.

5) Un nombre n est un nombre de **Kaprekar** en base 10, si la représentation décimale de n^2 peut être séparée en une partie gauche u et une partie droite v tel que $u + v = n$.
$45^2 = 2025$, comme $20 + 25 = 45$, 45 est aussi un nombre de **Kaprekar**. $4879^2 = 23804641$, comme $238 + 04641 = 4879$ (le 0 de 046641 est inutile, on a juste placé pour éviter toute confusion), alors 4879 est encore un nombre de **Kaprekar**.
   a) Est-ce que 9 est un nombre de **Kaprekar** ?
   b) Ecrire la fonction int **sommeParties(int n, int p)** qui découpe n est deux nombres dont le deuxième comporte p chiffres, et qui retourne leur somme.
   Par exemple, **sommeParties(12540, 2) = 125 + 40 = 165**
   c) Ecrire la fonction **int estKaprekar(int n)**

CHAPITRE 8

LES TABLEAUX

Sommaire

Comme tous les langages, C permet d'utiliser des tableaux. On nomme ainsi un ensemble d'éléments de même type désignés par un identificateur unique ; chaque élément est repéré par un indice précisant sa position au sein de l'ensemble.

8.1 Les tableaux à un indice

Un tableau est un ensemble fini d'éléments de même type, stockés en mémoire à des adresses contiguës.
La déclaration d'un tableau à une dimension se fait de la façon suivante :

```
type nom-du-tableau[nombre-éléments];
```

où *nombre-éléments* est une expression constante entière positive[2]. Par exemple, la déclaration

```
int tab[10];
```

indique que *tab* est un tableau de 10 éléments de type int. Cette déclaration alloue donc en mémoire pour l'objet tab un espace de 10×4 octets consécutifs. Pour plus de clarté, il est recommandé de donner un nom à la constante *nombre-éléments* par une directive au préprocesseur, par exemple

```
#define nombre-éléments 10
```

On accède à un élément du tableau en lui appliquant l'opérateur []. Les éléments d'un tableau sont toujours numérotés de 0 à *nombre-éléments -1*. Le programme suivant imprime les éléments du tableau *tab* :

```
#define N 10
main()
{
int tab[N];
int i;
...
for (i = 0; i < N; i++)
printf("tab[%d] = %d\n",i,tab[i]);
}
```

8.1.1 Quelques règles

Un élément de tableau est une lvalue. Il peut donc apparaître à gauche d'un opérateur d'affectation comme dans :
$tab[2] = 5$ Il peut aussi apparaître comme opérande d'un opérateur d'incrémentation, comme dans :
$tab[3] + +$ et $- - tab[i]$
En revanche, il n'est pas possible, si $t1$ et $t2$ sont des tableaux d'entiers, d'écrire $t1 = t2$.

Les indices

Un indice peut prendre la forme de n'importe quelle expression arithmétique de type entier (ou caractère, compte tenu des règles de conversion systématique). Par exemple, si n, p, k et j sont de type int , ces notations sont correctes :

```
t[n-3]
t[3*p-2*k+j%l]
Il en va de même, si c1 et c2 sont de type char , de :
t[c1+3]
t[c2-c1]
```

La dimension d'un tableau

La dimension d'un tableau (son nombre d'éléments) ne peut être qu'une constante ou une expression constante. Ainsi, cette construction :

```
#define N 50
.....
int t[N] ;
float h[2*N-1] ;
```

est correcte. En revanche, elle ne le serait pas (en C) si N était une constante symbolique définie par **const int N=50**, les expressions N et $2 * N - 1$ n'étant alors plus calculables par le compilateur (elle sera cependant acceptée en C++).

8.2 Les tableaux à deux indices

8.2.1 Leur déclaration

Comme tous les langages, C autorise les tableaux à plusieurs indices (on dit aussi à plusieurs dimensions). Par exemple, la déclaration :

```
int tab[5][3]
```

réserve un tableau de 15 (5 x 3) éléments. Un élément quelconque de ce tableau se trouve alors repéré par deux indices comme dans ces notations :

```
tab[3][2]
tab[i][j]
tab[i-3][i+j]
```

De manière similaire, on peut déclarer un tableau à plusieurs dimensions. Par exemple, pour un tableau à deux dimensions :

```
type nom-du-tableau[nombre-lignes][nombre-colonnes]
```

En fait, un tableau à deux dimensions est un tableau unidimensionnel dont chaque élément est lui-même un tableau. On accède à un élément du tableau par l'expression *tableau*[*i*][*j*].

8.2.2 Arrangement en mémoire des tableaux à deux indices

Les éléments d'un tableau sont rangés suivant l'ordre obtenu en faisant varier le dernier indice en premier. Ainsi, le tableau *tab* déclaré précédemment verrait ses éléments ordonnés comme suit :

```
tab[0][0]
tab[0][1]
tab[0][2]
tab[1][0]
tab[1][1]
tab[1][2]
....
tab[4][0]
tab[4][1]
tab[4][2]
```

8.3 Initialisation des tableaux

Comme les variables scalaires, il est possible d'initialiser (partiellement ou totalement) un tableau lors de sa déclaration. Les valeurs fournies devront obligatoirement être des expressions constantes, et cela quelle que soit la classe d'allocation du tableau concerné (alors que les variables scalaires automatiques pouvaient être initialisées avec des expressions quelconques). Voici quelques exemples vous montrant comment initialiser un tableau.

8.3.1 Initialisation de tableaux à un indice

La déclaration :

```
int tab[5] = { 10, 20, 5, 0, 3 } ;
```

place les valeurs 10 , 20 , 5 , 0 et 3 dans chacun des cinq éléments du tableau tab.
Il est possible de ne mentionner dans les accolades que les premières valeurs, comme dans ces exemples : La déclaration :

```
int tab[5] = { 10, 20 } ;
int tab[5] = { 10, 20, 5 } ;
```

Les valeurs manquantes seront, suivant la classe d'allocation du tableau, initialisées à zéro (statique) ou aléatoires (automatique).
De plus, il est possible d'omettre la dimension du tableau, celle-ci étant alors déterminée par le compilateur par le nombre de valeurs énumérées dans l'initialisation. Ainsi, la première déclaration de ce paragraphe est équivalente à :

```
int tab[] = { 10, 20, 5, 0, 3 } ;
```

Remarque :
La construction suivante :

```
#define N 10
...
int tab[5] = { 2*N-1, N-1, N, N+1, 2*N+1}
```

est correcte. Elle ne le serait pas, en revanche, si l'on définissait **N** comme une constante symbolique entière par **const int N = 10**.

8.3.2 Initialisation de tableaux à deux indices

Voyez ces deux exemples équivalents (nous avons volontairement choisi des valeurs consécutives pour qu'il soit plus facile de comparer les deux formulations) :

```
int tab [3] [4] = { { 1, 2, 3, 4 } ,
{ 5, 6, 7, 8 },
{ 9,10,11,12 }
}
int tab [3] [4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 } ;
```

La première forme revient à considérer notre tableau comme formé de trois tableaux de quatre éléments chacun. La seconde, elle, exploite la manière dont les éléments sont effectivement rangés en mémoire et elle se contente d'énumérer les valeurs du tableau suivant cet ordre. Là encore, à chacun des deux niveaux, les dernières valeurs peuvent être omises. Les déclarations suivantes sont correctes (mais non équivalentes) :

```
int tab [3] [4] = { { 1, 2 } , { 3, 4, 5 } } ;
int tab [3] [4] = { 1, 2 , 3, 4, 5 } ;
```

Exercice :

1) Écrire, de deux façons différentes, un programme qui lit 10 nombres entiers dans un tableau avant d'en rechercher le plus grand et le plus petit
2) Écrire une fonction qui ne renvoie aucune valeur et qui détermine la valeur maximale et la valeur minimale d'un tableau d'entiers (à un indice) de taille quelconque. Il faudra donc prévoir 4 arguments : le tableau, sa dimension, le maximum et le minimum. Écrire un petit programme d'essai.
3) Écrire une fonction permettant de trier par ordre croissant les valeurs entières d'un tableau de taille quelconque (transmise en argument). Le tri pourra se faire par réarrangement des valeurs au sein du tableau lui-même.
4) Écrire une fonction calculant la somme de deux matrices dont les éléments sont de type double. Les adresses des trois matrices et leurs dimensions (communes) seront transmises en argument.
5) Ecrire un programme qui lit deux tableaux A et B et leurs dimensions N et M au clavier et qui ajoute les éléments de B à la fin de A.
6) Ecrire une fonction **int somme(int T[], int n)** retournant la somme des n éléments du tableau T.
7) Ecrire un programme qui lit deux matrices A et B de dimensions N et M respectivement M et P au clavier et qui effectue la multiplication des deux matrices. Le résultat de la multiplication sera affecté à la matrice C, qui sera ensuite affichée.

Appendices

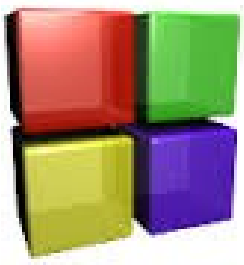
ANNEXE A

CODEBLOCKS

Code::Blocks

Table des matières

Code::Blocks est un environnement de développement intégré libre et multiplate-forme. Il est écrit en C++ grâce à la bibliothèque wxWidgets. Pour le moment, Code::Blocks est orienté C et C++, mais il supporte d'autres langages comme le D.



1. Introduction

Pour développer des programmes, il faut un logiciel qui permette :

- d'éditer du code (écrire le programme)
- de faire appel aux outils de compilation pour réaliser l'exécutable
- de débuguer le programme lors de son exécution

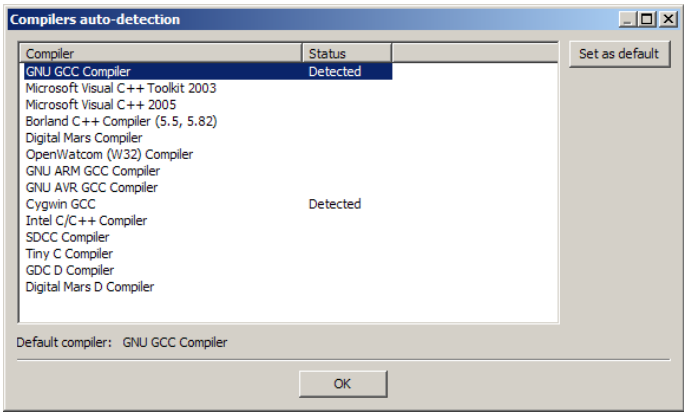
Un tel programme comme Code::Blocks s'appelle un IDE (Integrated Development Environment) :

- il est libre
- il est multi-plateformes (version pour Windows ou Linux)
- il est de taille raisonnable (installateur < 35Mo avec les outils de compilation)

Code::Blocks est capable de générer des applications écrites en C ou en C++ (en mode console ou non). Pour pouvoir utiliser Code::Blocks, il faut lui associer un compilateur tel MinGW[1]. Ce compilateur utilise les outils de compilations « libres » disponibles sur quasiment toutes les plateformes du marché.

2. Première exécution de Code::Blocks

Lors de la première exécution de Code::Blocks, si plusieurs compilateurs sont installés sur la machine, il faut choisir celui à utiliser par défaut. Le compilateur installé avec Code::Blocks est GNU[2] GCC[3] Compiler.



1 Minimalist GNU for Windows
2 Acronyme récursif : GNU is Not Unix
3 GNU C Compiler

3. Premier programme

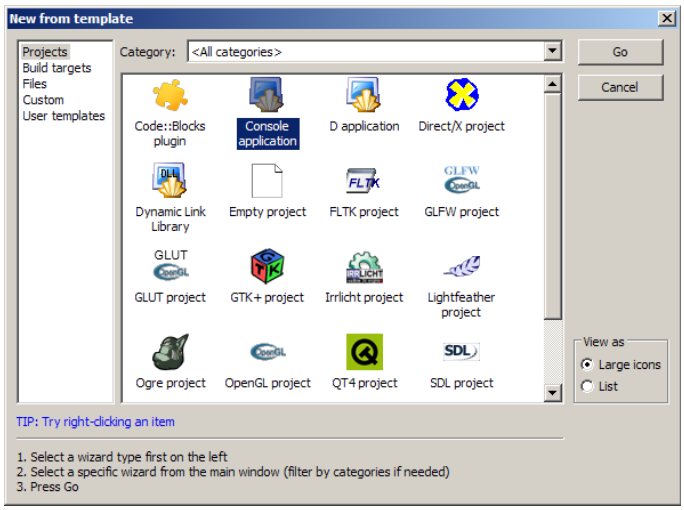
3.1. Créer un projet

Un projet contient tous les éléments nécessaires pour compiler un programme.

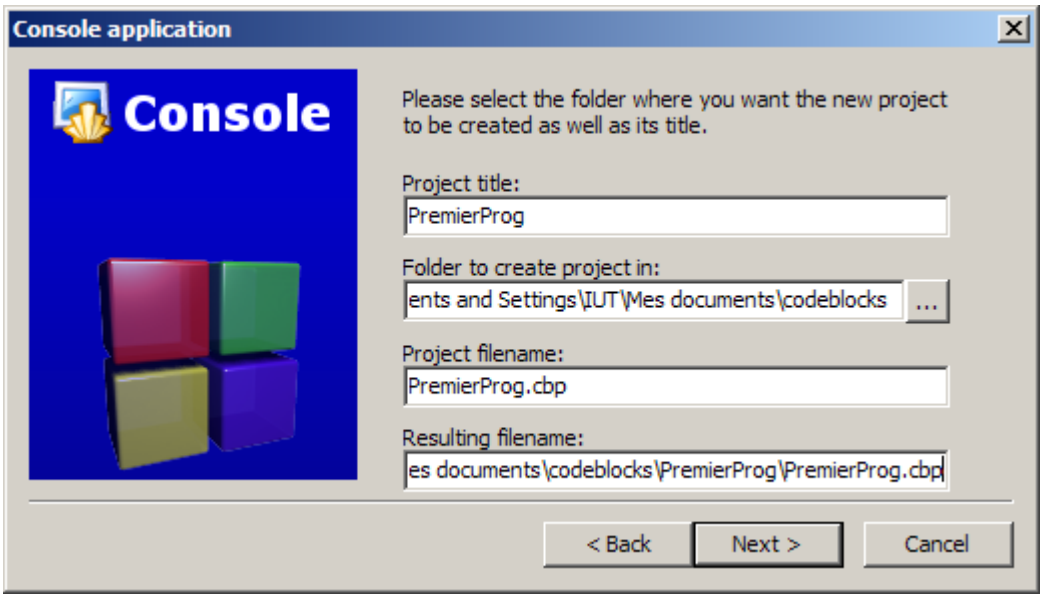
Vous pouvez créer un nouveau projet soit par le menu File → New → Project...

Vous pouvez aussi simplement cliquer sur Create a new project au milieu de l'écran d'accueil.

Une boîte de dialogue s'ouvre alors pour vous permettre de choisir le type de projet que vous souhaitez créer. Dans un premier temps, il est plus facile de créer des projets du type Console Application. Sélectionnez donc Console Application puis cliquez sur Go.



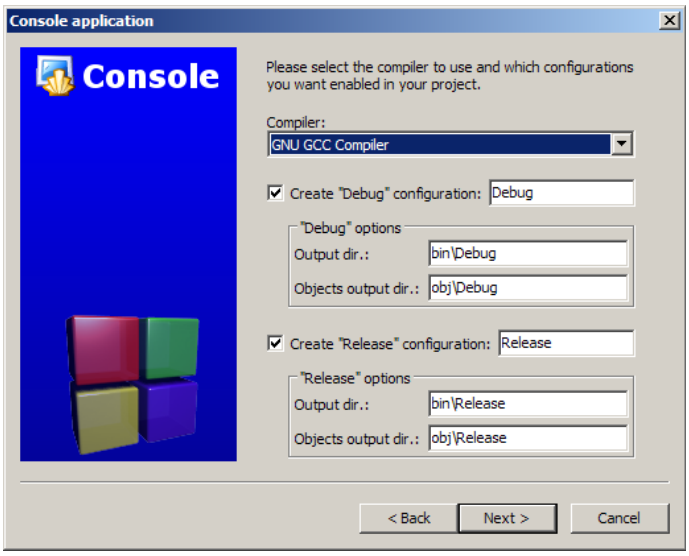
Vous devez maintenant choisir le nom du projet et son répertoire de sauvegarde.



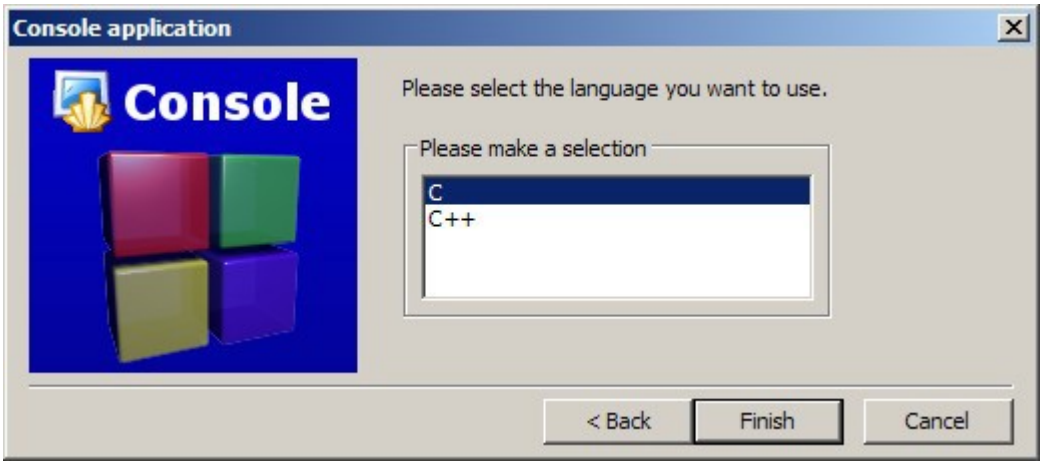
Code::Blocks place automatiquement tous les fichiers du projet dans un répertoire qui porte le nom du projet. Dans l'exemple ci-dessous, le répertoire du projet est Mes Documents\codeblocks et le nom du projet est PremierProg alors tous les fichiers du projets seront placés dans le répertoire Mes Documents\codeblocks\PremierProg

Pour le nom du projet, il est préférable de n'utiliser que des lettres et des chiffres et d'éviter tous les caractères spéciaux, en particulier les espaces.

Vous devez ensuite choisir le compilateur, il doit être automatiquement sélectionné à GNU GCC Compiler, et la création d'une version de débogage (Debug) et d'une version finale (Release) de votre programme.



Il reste à choisir un développement du programme en langage C.



Un clic sur Finish va maintenant créer le projet.

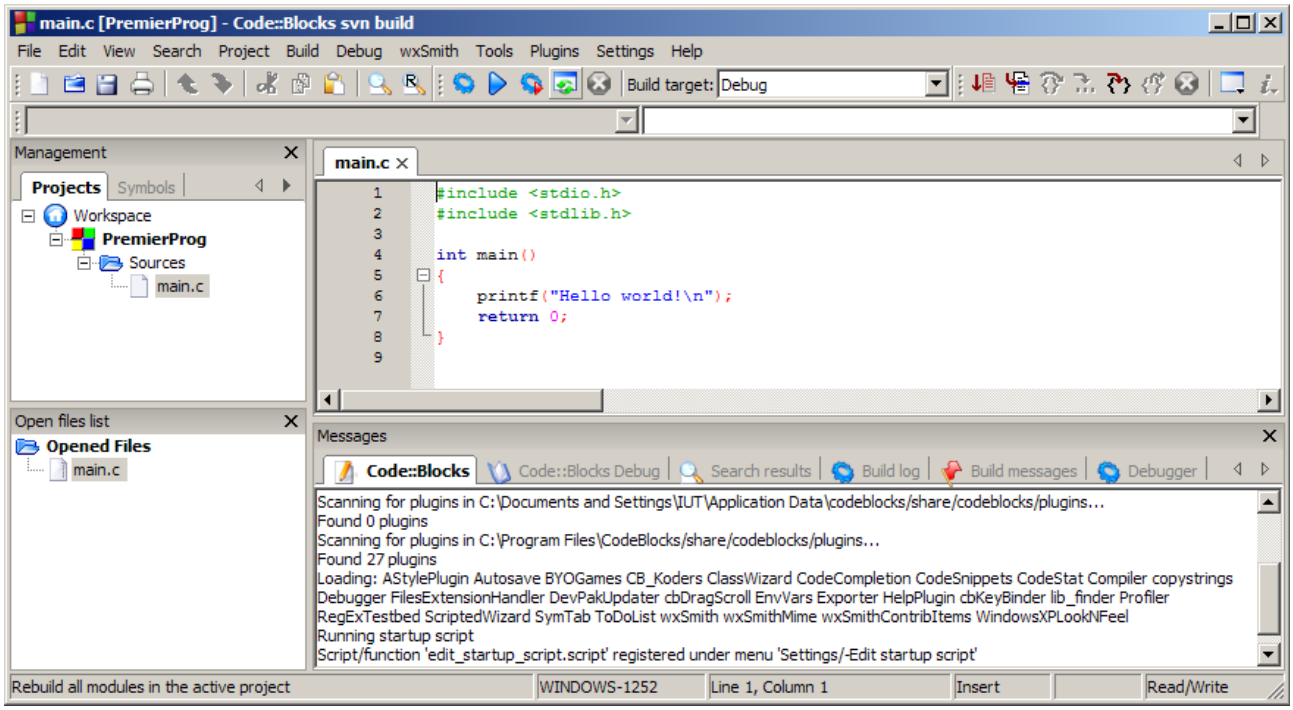
Dans le répertoire Mes Documents\codeblocks\PremierProg, deux fichiers ont été créés :

- PremierProg.cbp fichier du projet dont le format est propre à Code::Blocks et d'extension cbp pour Code::Blocks Project.
- main.c fichier en langage C.

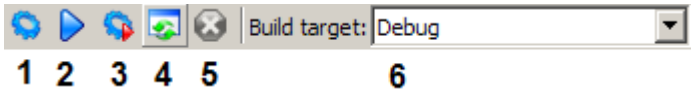
3.2. Construire et exécuter un projet

Vous pouvez éditer le fichier main.c en allant à gauche dans la fenêtre Management dans Sources → main.c. Ce fichier comporte la fonction main, fonction principale du programme.

Le fichier main.c contient les instructions pour afficher Hello world! à l'exécution. Pour pouvoir exécuter le programme, il faut tout d'abord construire (build) le projet. Cette opération peut être réalisée soit par le menu Build, soit par la barre d'outils Compiler que l'on peut activer ou désactiver par View → Toolbars → Compiler.



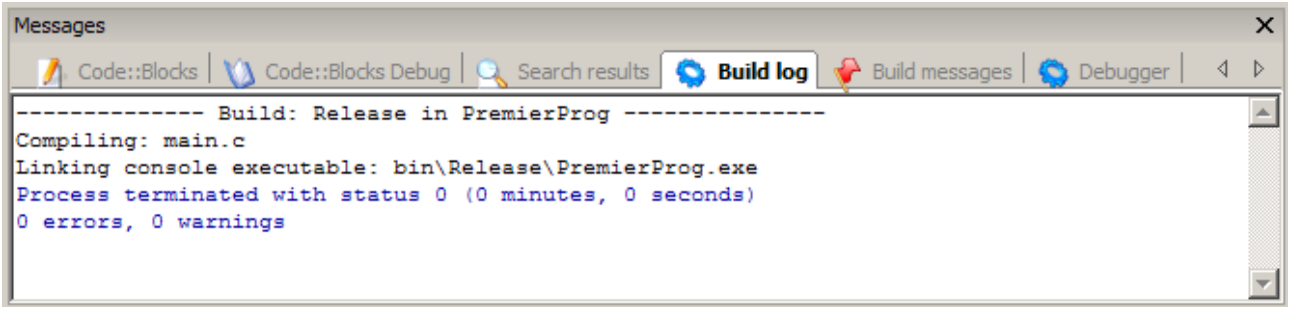
La barre d'outils Compiler :



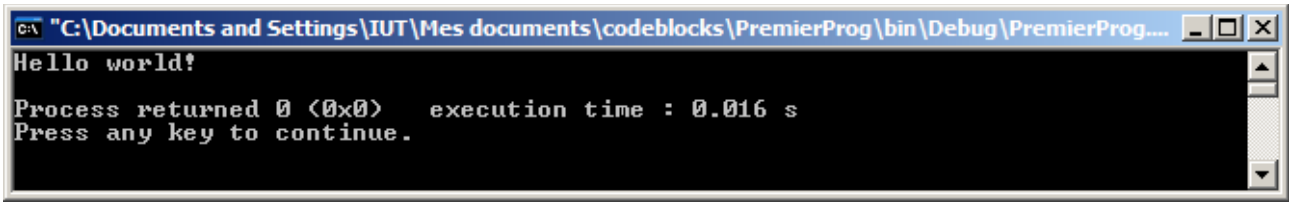
1. Construction du projet (Build)
2. Exécution du projet (Run)
3. Construction, puis exécution du projet (Build and run)
4. Tout reconstruire (Rebuild)
5. Termine l'exécution en cours (Abort)
6. Choix du type de cible, débogage (Debug) ou version finale (Release)

La version finale donne un fichier exécutable plus petit et généralement plus efficace mais elle n'autorise pas le débogage. Les fichiers sources compilés sont rangés dans le sous-répertoire obj du projet. Le programme exécutable se trouve dans le sous-répertoire bin\Debug pour la version de débogage et dans le sous-répertoire bin\Release pour la version finale. Sous Windows, le nom du programme correspond au nom du projet avec l'extension exe.

On obtient l'affichage du résultat d'un Build dans l'onglet Build log de la fenêtre Messages.

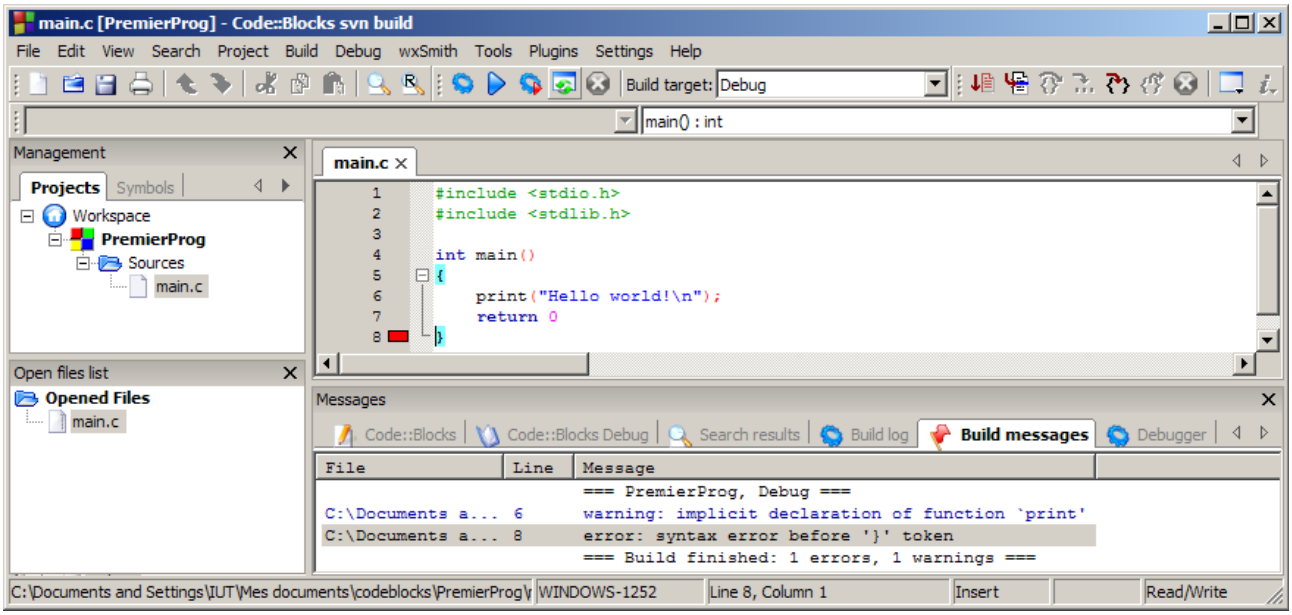


Si on lance le programme (Run), une fenêtre s'ouvre et affiche le résultat de l'exécution.



4. Erreur à la compilation

Les erreurs lors de la compilation du projet sont recensées dans l'onglet Build messages de la fenêtre Messages. Un simple clic sur l'erreur envoie le curseur dans le code à l'endroit où l'erreur est détectée. Un marqueur rouge dans la marge du code indique la ligne concernée. Attention, souvenez vous qu'en langage C, un point-virgule oublié en fin de ligne entraîne une erreur sur la ligne suivante. À titre d'exemple, créons volontairement deux erreurs dans le programme précédent.



Écrivons print à la place de printf et supprimons le point virgule après le return 0. Lors du Build, nous obtenons alors l'affichage ci-contre : l'absence de point-virgule apparaît comme une erreur.

À la compilation, l'usage de la fonction inconnue print (à la place de printf) génère un warning.

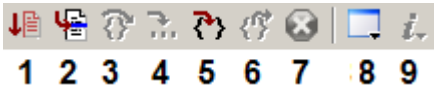
Elle aurait déclenché une erreur à l'édition de lien.

5. Utiliser le débogueur

5.1. Les commandes du débogueur

Il est possible d'utiliser le débogueur soit à partir du menu Debug, soit grâce à la barre d'outils Debugger. On peut activer ou désactiver cette barre d'outils par View → Toolbars → Debugger.

La barre d'outils Debugger :



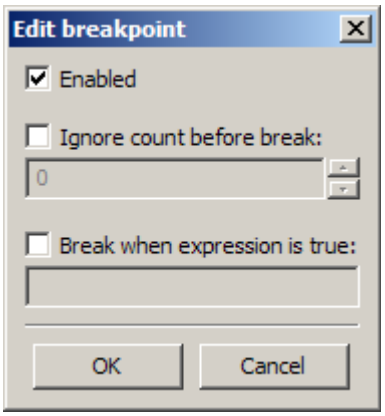
1. Exécute en mode débogueur (Debug/Continue)

   Attention, si aucun point d'arrêt n'est placé, le programme s'exécute normalement.
2. Exécute jusqu'au curseur placé dans le code (Run to cursor)
3. Exécute une ligne sans entrer dans les fonctions (Next line)
4. Exécute une instruction en assembleur (Next instruction)
5. Exécute une ligne en entrant dans les fonctions (Step into)
6. Exécute jusqu'à la fin de la fonction en cours (Step out)
7. Termine l'exécution en cours (Abort)
8. Menu d'ouverture des fenêtres de débogage
9. Menu d'informations

5.2. Les points d'arrêt (Breakpoint)

Pour placer ou enlever un point d'arrêt (Breakpoint) dans le programme, il faut cliquer dans la marge juste après le numéro de la ligne. Un rond rouge doit apparaître. On peut aussi utiliser le menu Debug → Toggle breakpoint ou la touche F5. Lors de l'exécution en mode débogage, le programme s'arrêtera sur les points d'arrêt.

Il est possible de désactiver un point d'arrêt (sans le supprimer), de lui associer un nombre de passages avant arrêt ou une condition logique pour stopper. Un clic droit sur le point d'arrêt permet d'accéder à un menu d'édition du point d'arrêt (Edit breakpoint). Une boîte de dialogue s'ouvre alors et on peut choisir d'ignorer un certain nombre de passages avant de prendre en compte le point d'arrêt ou de ne s'arrêter que pour une condition logique particulière exprimée en langage C.



Lors d'un arrêt en mode débogage, une flèche jaune indique l'avancée de l'exécution du programme.

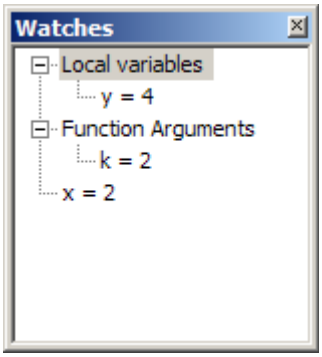
5.3. Visualiser l'état des variables (Watches)

Lors du fonctionnement en mode débogage, Code::Blocks analyse automatiquement les valeurs des variables locales et des arguments des fonctions. Ces informations se trouvent dans la fenêtre

Watches.

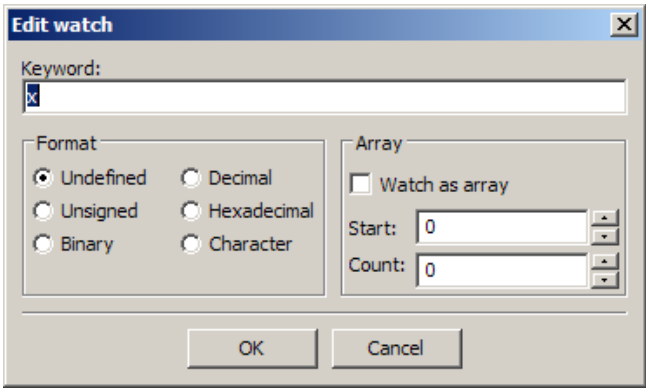
Vous pouvez ouvrir cette fenêtre grâce au menu Debug → Debugging windows → Watches. Vous pouvez aussi utiliser la barre d'outils Debugger et l'icône d'ouverture des fenêtre de débogage. La fenêtre Watches comprend par défaut deux listes, une pour les variables locales et une pour les arguments des fonctions. Les variables dans ces listes sont ajoutées et retirées de manière automatique en fonction du déroulement du programme. Un arrêt dans la fonction suivante :

```
double fct(double k)
{
      double y;

      y = k * x;

      return y;
}
```



où x est une variable globale donne l'affichage ci-dessus dans la fenêtre Watches (x a été ajouté manuellement).

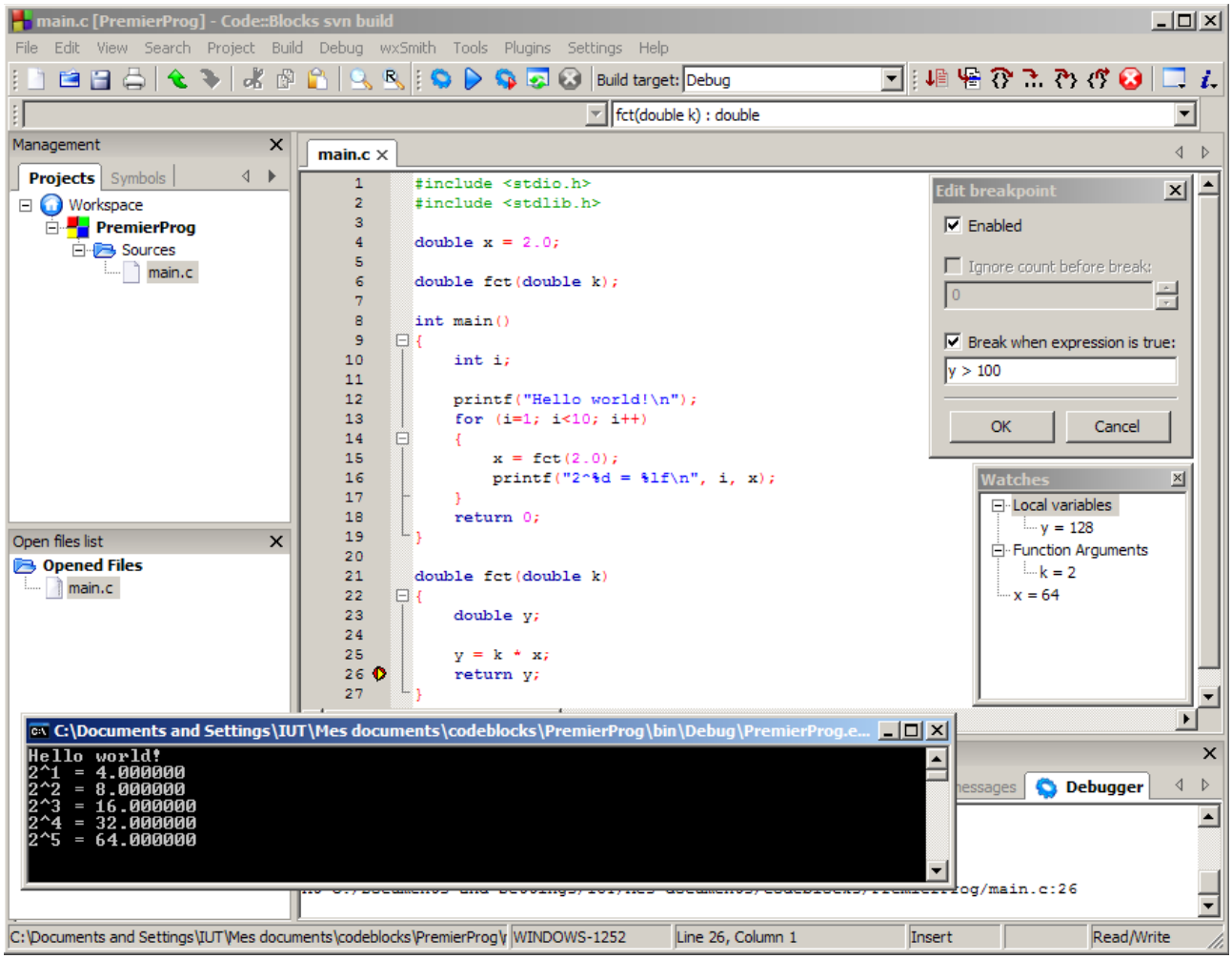
Un clic droit dans la fenêtre Watches permet d'accéder à un menu pour ajouter une variable (Add watch). Un clic droit sur une variable dans la fenêtre Watches permet de changer sa valeur (Change value) ou de modifier sa représentation (Edit watch).



Un clic droit sur une variable dans le code pendant une session de débogage permet d'obtenir un menu pour ajouter la visualisation de la variable.

5.4. Exemple d'une session de débogage

Le programme est arrêté par un point d'arrêt avec condition :



6. Autres fonctionnalités

6.1. Installer un fichier d'aide

Il est pratique d'obtenir rapidement de l'aide sur les fonctions des librairies standard du langage C. Pour cela, il faut associer un fichier d'aide à un appui sur la touche F1 quand le curseur est placé sur une fonction.

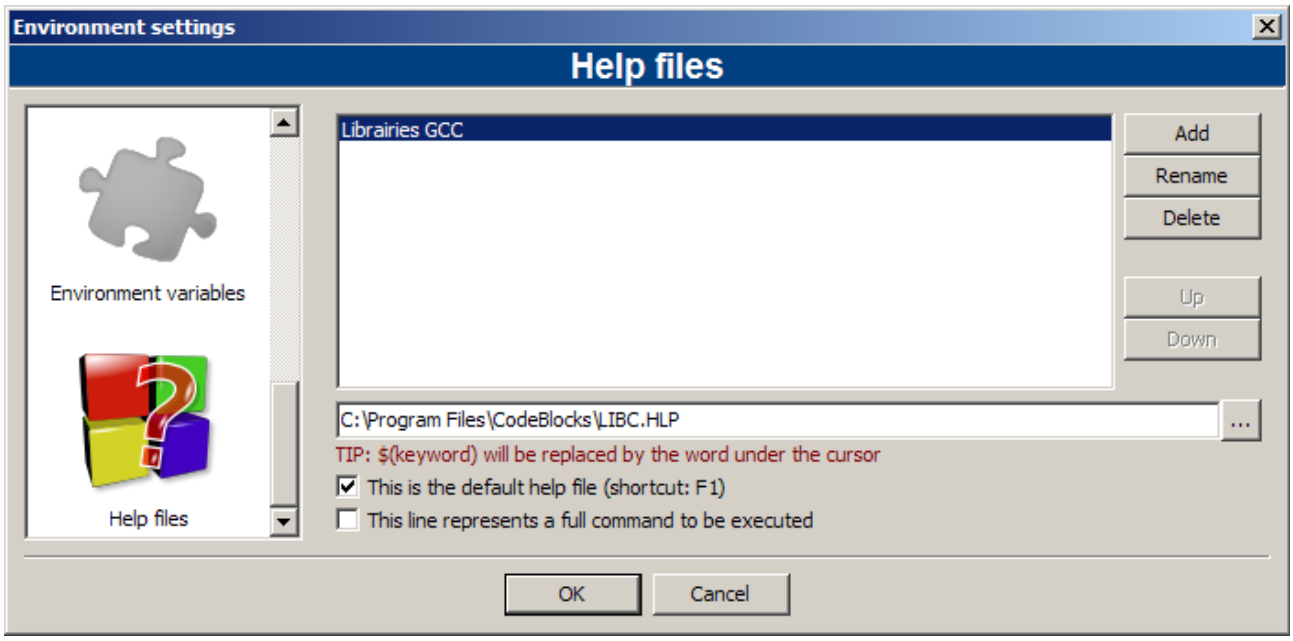
1. Allez dans le menu

   Settings → Environment...

2. Dans la fenêtre qui s'ouvre, faites défiler la partie gauche jusqu'à trouver Help files et cliquez sur Help files.

3. Cliquez sur le bouton Add et entrez un nom pour le fichier d'aide (par exemple Librairies GCC).

4. Cliquez sur Oui pour parcourir (browse) l'arborescence des fichiers.
5. Recherchez le fichier LIBC.HLP qui se trouve à l'emplacement C:\Program Files\CodeBlocks\LIBC.HLP pour une installation standard.
6. Cochez l'option This is the default help file (shortcut: F1) pour activer l'usage de la touche F1.
7. Terminez en cliquant sur OK.



Pour vérifier le fonctionnement, placez le curseur sur la fonction main et appuyez sur F1. Une fenêtre d'aide doit s'ouvrir pour vous donnez des informations sur la fonction main en langage C.

6.2. Formater automatiquement son code

Le langage C n'impose aucune contrainte d'écriture du code. Pour ceux qui écrivent en langage C comme des porcs, Code::Blocks intègre un outils de mise en forme automatique du code : AStyle. Pour l'utiliser, il suffit d'aller dans le menu Plugins → Source code formatter (AStyle).

On peut configurer la mise en forme du code en allant dans le menu Settings → Editor... puis en sélectionnant sur la gauche de la fenêtre Source formatter.

Avant AStyle

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

double x = 1.0;
void fct(void);

int main()
{
int i;

printf("Hello world!\n");
for (i=1; i<10; i++) {
fct();
```

Après AStyle

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

double x = 1.0;
void fct(void);

int main()
{
      int i;

      printf("Hello world!\n");
      for (i=1; i<10; i++)
      {
```

```
printf("2^%d = %lf\n", i, x);
}

return 0;
}

void fct(void){
x = 2 * x;
}
```

Pour en savoir plus, cliquer ici 

```
            fct();
            printf("2^%d = %lf\n", i, x);
      }

      return 0;
}

void fct(void)
{
      x = 2 * x;
}
```

BIBLIOGRAPHIE

[1] *Programmer en langage C*. Eyrolles, 2009.

[2] Anne Canteaut. Programmation en langage c. Technical report, INRIA, •.

[3] http ://projet.eu.org/pedago/sin/tutos/. Sciences informatique et numérique, •.