EiJV
ÉCOLE D'INGÉNIEURS

Université de Picardie Jules Verne
Ecole d'Ingénieurs Jules Verne
Filière Cybersécurité

UNIVERSITÉ de Picardie Jules Verne

Language C

C. Drocourt

cyril.drocourt@u-picardie.fr

SOMMAIRE

Chapitre 1 - Premiers pas en C

Table des matières

1 - Présentation du langage C, ses atouts.

Le langage C est né dans les 1970 à la seule fin de réécrire un système d'exploitation, Unix, alors écrit en assembleur.

Le Langage C est dérivé du langage B, lui même inspiré du langage BCPL. Ses deux principaux inventeurs sont Brian Kernighan et Dennis Ritchie (on parle souvent du C de Kernighan et Ritchie), mais il ne faut pas oublier Ken Thompson, qui lui fût l'inventeur du langage B.

Lors de son apparition, les autres langages disponibles étaient plutôt limités :

- Fortran : dédié au calcul scientifique,
- Cobol : dédié à la gestion,
- Assembleur : Compliqué et non portable,

Le but à l'époque du langage C était :

- être indépendant du matériel,
- être rapide,
- être un langage de haut niveau,
- posséder des types de données,
- être portable.

Évidemment, la notion de langage de haut niveau a aujourd'hui évolué, et le langage C est inversement à notre époque considéré comme un langage bas niveau.

La majorité des systèmes d'exploitations actuels ont au moins une partie en langage C/C++ notamment :

- Windows
- Linux
- Mac OS X

2 - Evolution du langage

Le langage C a connu quelques améliorations au fur et à mesure du temps :

- C K&R (1970) : La base,
- AINSI C ou C89 (1989) ou C90 (1990) : Standardisation et définitions de bibliothèques standards,
- C99 (1999) : Nouveaux types (_Bool, _Complex, …), fonctions « inline », commentaires « // », ...
- C11 (2011) : Threads, Unicode, tableaux dynamiques,...

La norme la plus courante dans les compilateurs est un mélange de C99 et de C89, en effet, la majorité des compilateurs à inclut des éléments de C99 mais pas forcément tout, ce qui rend par moment difficile la portabilité d'un programme au sens strict.

Le langage C++ est un langage objet, il conserve la majorité des éléments de syntaxe du langage C mais les ajouts en font un langage à part.

Voici le premier programme donné directement par Kernighan et Ritchie en 1978 :

```
main()
{
    printf("hello, world\n");
}
```

Comme on peut le constater pour commencer :

- Le point d'entrée d'un programme s'appelle « main », il est donc oblogatoire,
- Un début de bloc est matérialisé par un caractère '{',
- Une fin de bloc est caractérisé par un caractère '}',
- L'affichage d'un texte est effectué à l'aide de la fonction « printf »,
- Chaque ligne se termine par un caractère ' ;',

3 - Les fichiers sources (.c, .h).

Dans un ensemble de programmes écrits en langage C on trouve habituellement deux types de fichiers :

- **Les fichiers « .c »** : qui contiennent du code C,
- **Les fichiers « .h »** : qui contiennent des entêtes que l'on inclus dans le code C,

En effet, le langage C ne contient de base aucune fonction, que ce soit pour l'affichage, le calcul scientifique, l'enregistrement de fichiers, … Ainsi, il est nécessaire d'utiliser des bibliothèques externes, mais il faut au préalable indiquer en début de programme les entêtes de ces librairies pour informer le compilateur de l'existence de certaines fonctions.

On retrouve donc en début de programme des lignes du type :

```
#include <XXX.h>
```

Si on reprend notre premier programme, l'utilisation de la fonction « printf » nécessite d'indiquer l'entête qui contient la définition de cette fonction, notre programme serait donc :

```
#include <stdio.h>

main()
{
    printf("hello, world\n");
}
```

4 - Structure générale d'un programme.

Un programme en langage C est structuré de la manière suivante :

- **Directives de pré-compilation** : Elles sont traitées par une première passe du compilateur, et ne représente pas du code, elles débutent par le caractère « # »,
- **Définitions globales** : On y retrouve la définition d'éléments qui doivent être utilisés par l'ensemble du programme, que ce soit des types de données ou des variables.
- **Fonctions et/ou prototypes** : Définition des fonctions qui seront utilisés par le programme avec le code associé,
- **Partie principale du programme** : Le point d'entrée du programme appelé « main » auquel on associe le code principal, ce point d'entrée doit retourner un entier.

Notre programme en suivant la nome C89/Ansi serait donc :

```
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
    return 0;
}
```

Nous pouvons remarquer que :

- Un type (int : entier) a été ajouté à gauche du « main » indiquant que le programme retourne un entier,
- Dans les arguments du « main » on trouve le mot « void » qui indique en C soit vide soit non typé, c'est le premier cas ici,
- A la fin de notre main, l'instruction « return 0 » indique que le programme retourne donc l'entier 0.

5 - Les commentaires

En utilisant la norme C89/Ansi les commentaires doivent se trouver entre les caractères « /* » et « */ », ainsi notre programme serait :

```
/* hello.c : Premier programme  */
/* Par : C. Drocourt */
/* Date : 12/05/1980 */
#include <stdio.h>

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

Il est également possible d'utiliser les commentaires sur plusieurs lignes :

```
/* Le commentaire se trouve
    sur deux lignes */
```

A partir de la norme C99, il est possible également d'utiliser le format de commentaires introduits par le C++ à l'aide des caractères « // » :

```
// hello.c : Premier programme
// Par : C. Drocourt
// Date : 12/05/1980
```

La différence fondamentale est que cette syntaxe ne met en commentaire **QUE** le reste de la ligne suivant ces deux caractères, alors que la première syntaxe permet de mettre en commentaire plusieurs lignes.

Il est possible de mixer les deux types de commentaires dans un même programme suivant les besoins.

6 - Utilisation élémentaire de la chaîne de production

Afin de transformer un fichier en langage C en un programme exécutable, plusieurs étapes sont nécessaires. A chacune de ces étapes, il est possible de générer le fichier intermédiaire correspondant, ainsi en partant d'un fichier source `fichier.c`, il faut :

- passer le pré-processeur,
- générer un fichier assembleur,
- faire l'assemblage de ce fichier en fichier objet,
- faire l'édition de liens avec les librairies utiles et générer l'exécutable,

Code source → **Assembleur** → **Objet** → **Exécutable**

Il est évidemment possible de :

- Passer directement de « fichier.c » au fichier exécutable,
- Nommer l'exécutable comme souhaité, « a.out » étant le nom par défaut sous Unix par exemple.

7 - Les environnements d'édition, de compilation et d'exécution

7.1 - Linux

Le langage C est né sous Unix, il paraît donc logique que Linux soit un environnement de prédilection pour développer dans ce langage. Le compilateur sous Linux est le GNU C Compiler, ou « gcc ». Les principales options du compilateur sont:

- `-O` : optimisation demandée,
- `-E` : ne passe que le pré-processeur,
- `-S` : s'arrêter au fichier assembleur `.s`,
- `-c` : s'arrêter au fichier objet `.o`,
- `-o NAME` : nom du fichier final, « `a.out` » par défaut,
- `-g` : générer les informations pour le debugger,
- `-L <catalogue>` : Recherche des librairies dans le catalogue indiqué,
- `-static` : Force l'édition de lien à utiliser des librairies statiques (dynamique par défaut),
- `-lzzz` : ajouter la librairie « `zzz` » lors de l'édition de liens.

Exemples de compilation :

Soit un programme nommé « myprog.c », que l'utilisateur souhaite compiler pour obtenir un exécutable se nommant « myprog » :

```
# en une seul directive
gcc -o myprog myprog.c
```

```
# en deux étapes
gcc -c myprog.c
gcc -o myprog myprog.o
```

Pour exécuter le programme :

```
./myprog
```

Autres options intéressantes :

- `-Wall` : Demande au compilateur d'afficher également toutes les messages informatifs liés à la compilation (Warning All), ce qui permet de détecter des oublis, comme par exemple des variables non utilisées ou non initialisées,
- `-ansi` : Permet de demander au compilateur de se placer en mode de compatibilité « ansi », c'est à dire à la compatibilité C90,
- `-std=<norme>` : Permet de demander au compilateur de se placer dans le mode de compatibilité demandé (c90, c99, c11),

La chaîne de compilation sera donc la suivante :

Code source		**Assembleur**		**Objet**		**Exécutable**
Fichier.c	→	Fichier.s	→	Fichier.o	→	a.out

Environnements :

Sous Linux, les principaux éditeurs/environnements ont :

- vi/vim : Editeur historique sous Unix,
- Emacs : Idem,
- Geany : Editeur multi-langage, léger et simple d'utilisation,
- Code::Blocks : IDE C/C++ multi-plateforme,
- Eclipse (d'IBM) : Historiquement pour Java mais permet également le développement en Langage C avec l'environnement CDT (C/C++ Development Tools),
- Qt Creator : Solution de Nokia orientée sur l'utilisation de la librairie QT,
- Anjuta DevStudio : Dédié plus particulièrement aux applications GNOME,
- Kdevelop : Dédié aux application KDE,

7.2 - Sous Windows

Compilateur « gcc »

Pour programmer en langage C sous Windows de la même manière que sous Linux, il faut installer le compilateur « gcc » afin de retrouver le même environnement de développement. Pour ce faire, plusieurs solutions existent :

- **MinGW** (Minimalist GNU for Windows) : La solution la plus simple, qui permet d'installer le compilateur ainsi que l'ensemble des composants minimum pour le faire fonctionner (répertoire « c:\mingw-w64 » par défaut),
- **CYGWIN** : Solution plus complète qui permet la même chose que précédemment, mais aussi de compiler des programmes spécifiques à Linux, utilisant notamment des appels systèmes spécifiques,

Les options sont les mêmes que sous Linux, la seule différence notable étant le nom des fichiers par défaut :

Code source		**Assembleur**		**Objet**		**Exécutable**
Fichier.c	→	Fichier.s	→	Fichier.obj	→	a.exe

Compilateur Microsoft

Avec Visual Studio, plusieurs outils sont disponibles pour réaliser le processus de compilation :

- **cl.exe** : Le compilateur Microsoft, disponible depuis l'invite de commande développeur de Visual Studio,
- **link.exe** : L'éditeur de liens de Microsoft, disponible depuis l'invite de commande développeur de Visual Studio, appelé par défaut par la compilateur,

Les principales options du compilateur « cl.exe » sont :

- `/c` : Demande au compilateur de ne pas invoquer automatiquement l'éditeur de liens, et de générer uniquement le fichier « objet »,
- `/link` : Permet de placer des options spécifiques qui seront destinées à l'éditeur de liens.
- `/E` (ou `/P`) : Demande de ne réaliser uniquement que la phase de préprocesseur,

- `/Zg` : Pour générer les prototypes de fonctions,
- `/Zs` : Pour vérifier la syntaxe,
- `/Fo` : Pour préciser le nom du fichier de sortie,

Exemples de compilation :

Soit un programme nommé « myprog.c », que l'utilisateur souhaite compiler pour obtenir un exécutable se nommant « myprog.exe ». En une seule commande :

```
cl.exe myprog.c
```

De manière plus explicite :

```
cl.exe myprog.c /Fe:prog.exe
```

En deux étapes :

```
cl.exe /c myprog.c
link.exe /OUT:myprog.exe myprog.obj
```

Pour exécuter le programme :

```
myprog.exe
```

IDE

- **Dev-C++** : Environnement de développement qui embarque MinGW par défaut, toutefois l'utilisation de CYGWIN reste possible,
- **Visual Studio** : Environnement de développement qui s'appuie sur le compilateur « cl » de Microsoft, plutôt utilisé pour les applications spécifiques Windows,
- **Code::Blocks** : IDE C/C++ multi-plateforme (***disponible avec le compilateur***),
- **Eclipse** (d'IBM) : Historiquement pour Java mais permet également le développement en Langage C avec l'environnement CDT (C/C++ Development Tools),
- **Autres** : Tout éditeur de texte comme « Notepad » permettent d'éditer des fichiers en langage C,

8 - La syntaxe de base du langage

Une instruction se termine par un « ; » :

```
i+2;
```

On peut placer plusieurs instructions sur une seule ligne, mais c'est en général déconseillé pour la lisibilité :

```
13;i+2;
```

Une instruction peur contenir une opération :

```
i=13+1;
```

On peut regrouper des instructions en blocs par les caractères « { » et « } » :

```
{ i=2+1;c=i+3;}
```

Une déclaration de variable commence par la définition de son type :

```
int i;
```

9 - Les types de données et les constantes de base

9.1 - Les types principaux

Les types de bases sont :

- **`char`** : Un caractère sur un taille de un octet (8 bits),
- **`int`** : Un entier, habituellement sur 4 octets mais peut changer suivant les compilateurs et les plateformes et être sur 2 octets,
- **`short`** (ou short int) : Un entier sur 2 octets (16 bits),
- **`long`** (ou long int) : Un entier sur 4 octets (32 bits),
- **`long long`** (ou long long int) : Suivant les compilateurs et architectures, un entier sur 8 octets (64 bits),
- **`float`** : Un flottant sur 4 octets (32 bits),
- **`double`** : Un flottant sur 8 octets (64 bits),
- **`long double`** : Un flottant sur 10 octets (80 bits),

De plus, un type entier est signé par défaut (signed) c'est à dire qu'il peut prendre des valeurs positives ou négatives. Dans le cas contraire il faut le préciser par le terme « unsigned », par exemple pour un entier court : **unsigned short**.

9.2 - Plages de valeurs

Les types sont signés par défaut sauf pour le type « char », ainsi un type « short » peut contenir une valeur de -32768 à 32767 (-2^{15} à $2^{15}-1$), pour utiliser des valeurs non signées on utilise le mot « unsigned », par exemple « unsigned short » peut contenir une valeur de 0 à 65535 ($2^{16}-1$).

Voici un tableau récapitulatif des plages de valeurs :

Type	Taille	Plage de valeur
(unsigned) **char**	1 octet – 8 bits	0 à 255 (2^8-1)
signed char	1 octet – 8 bits	-128 (-2^7) à +127 ($+2^7-1$)
unsigned short	2 octets – 16 bits	0 à +65535 ($2^{16}-1$)
(signed) **short**	2 octets – 16 bits	-32768 (-2^{15}) à +32767 ($+2^{15}-1$)
unsigned long	4 octets – 32 bits	0 à +4 294 967 294 ($2^{32}-1$)
(signed) **long**	4 octets – 32 bits	-2^{31} à $+2^{31}-1$ (2.147.483.647)
unsigned long long	8 octets – 64 bits	0 à $2^{64}-1$ ($18*10^{18}-1$)
(signed) **long long** (int)	8 octets – 64 bits	-2^{63} à $+2^{63}-1$

Dans le fichier d'inclusion « limits.h », il est possible d'utiliser les constantes prédéfinies permettant d'obtenir les valeurs minimales et maximales de ces types :

Type	Minimum	Maximum
(unsigned) **char**	0	UCHAR_MAX
signed char	SCHAR_MIN	SCHAR_MAX
unsigned short	0	USHRT_MAX
(signed) **short**	SHRT_MIN	SHRT_MAX
unsigned long	0	UINT_MAX
(signed) **int**	INT_MIN	INT_MAX
unsigned int	0	ULONG_MAX
(signed) **long**	LONG_MIN	LONG_MAX
unsigned long long	0	ULONGLONG_MAX
(signed) **long long** (int)	LONGLONG_MIN	LONGLONG_MAX

Il faut noter que la norme C99 amène de nouveaux types plus explicites qui correspondent aux précédents dans le fichier « stdint.h », cependant peu de développeurs ont pris l'habitude de les utiliser :

- `int8_t, int16_t, int32_t, int64_t` : Entiers signés respectivement sur 8, 16, 32 et 64 bits,
- `uint8_t` : Entiers non signés respectivement sur 8, 16, 32 et 64 bits,

De la même manière, des constantes sont définies pour les plages de valeurs minimales et maximales :

- `INT8_MIN, INT16_MIN, INT32_MIN, INT64_MIN` : valeurs minimales signées
- `INT8_MAX, INT16_MAX, INT32_MAX, INT64_MAX` : valeurs maximales signées,
- `UINT8_MAX, INT16_MAX, INT32_MAX, INT64_MAX` : valeurs maximales non signées,

9.3 - Les réels

Les réels possèdent une représentation bien spécifiques, en effet, un réel n'est pas stocké sous la forme « X,Y », mais sous la forme $s.m.b^e$ définie par la norme IEEE754 avec :

- s : Signe,
- m : Mantisse qui contient le nombre,
- b : base, dans la norme c'est 2,
- e : exposant, l'exposant de la base, qui permet de positionner la position de la virgule,

Nous avons donc les codages suivants :

Type	Taille mantisse	Taille exposant	Exposant min	Exposant max
float	23 bits	8 bits	2^{-126}	2^{127}
double	52 bits	11 bits	2^{-1022}	2^{1023}
long double	64	15	2^{-16382}	2^{16383}

Ce qui est important de comprendre ici, c'est que l'on ne peut pas uniquement spécifier un type réel en fonction de sa valeur minimale et maximale, mais aussi et surtout en fonction de sa précision :

Type	Taille	Intervalle de valeurs	Précision
float	4 octets	$1,17*10^{-38}$ à $3,40*10^{38}$	6 chiffres
double	8 octets	$2,22*10^{-308}$ à $1,79*10^{308}$	15 chiffres
long double	10 octets	$3,36*10^{-4932}$ à $1,19*10^{4932}$	19 chiffres

La valeur minimale ici indique le plus petit chiffre possible en valeur absolu, idem pour la valeur maximale, ce qu'on peut schématiser par :

[-MAX ;-MIN],0,[MIN;MAX]

Le programme va réaliser des approximations dès que le réel en question ne peut pas être représenté, ce qui peut engendrer d'énormes écarts au final. Il conviendra donc de choisir convenablement ses types de réels.

De la même manière que pour les entiers, certaines constantes sont prédéfinies dans le fichier d'inclusion spécifique « float.h » :

- `FLT_MIN` et `FLT_MAX` : Pour les valeurs minimales et maximales d'un type « float » en valeur absolue,
- `DBL_MIN` et `DBL_MAX` : Pour les valeurs minimales et maximales d'un type « double » en valeur absolue,
- `LDBL_MIN` et `LDBL_MAX` : Pour les valeurs minimales et maximales d'un type « long double » en valeur absolue,

9.4 - Initialisation

Lorsque l'on déclare une variable, cette dernière ne possède aucune valeur et son contenu est donc **indéfini**. Il est donc primordial de lui affecter une valeur d'initialisation afin de pouvoir l'utiliser :

```
int nombre ;
nombre = 3,
```

ou plus court :

```
int nombre = 3 ;
```

Il est possible de déclarer plusieurs variables du même type au même moment :

```
int nombre=3, total = 0 ;
```

9.5 - Constantes

Il existe deux manière de déclarer une constante :

- En précédent la déclaration par le mot « const », la variable n'est plus modifiable :

```
const int mavar=3 ;
```

- En utilisant la directive de précompilation « #define » :

```
#define MAVAR 3
```

Les différences sont :

- Dans le premier cas, la variable occupe de l'espace mémoire, pas dans le deuxième,
- La deuxième solution est plus générale car permet de remplacer toute chaîne par une autre, pas seulement des variables ou constantes,
- Dans le premier cas il s'agit réellement d'une variable, et elle est donc typée, pas dans le deuxième,

Valeurs de constantes

Une constante peut prendre la valeur suivantes :

- Un chiffrer entier : `12948`
- Un chiffre octal commençant par un 0 : `034`
- Un chiffre hexadécimal commençant par « 0x » : `0x3F`
- Une chaine de caractère entre double guillemets : `"bonjour à tous"`
- Un caractère unique entre simple guillemets : `'c'`
- Un entier long : `234L`
- Un entier long non signé : `234UL`
- Un double : `12.34`
- Un flottant : `12.34F`

Certains codes de caractères ont une symbolique spécifique :

- `'\n'` : Nouvelle ligne,
- `'\t'` : tabulation,
- `'\b'` : Backspace ou retour arrière,

D'une manière générale le caractère '\' permet de placer des caractères spéciaux en indiquant :

- Le code octal du caractère : \33
- Le code hexadécimal du caractère : \x2F
- Le caractère ? : \ ?
- Le caractère " : \"

10 - Variables globales et locales

Une variable possède une portée en fonction de la position où elle est déclarée :

- Au début de programme : c'est une variable globale,
- Au début de fonction : c'est une variable locale à la fonction,
- Au début d'un bloc : sa porté est le bloc dans laquelle elle est déclarée,

Par exemple :

```
int var1;

int main(void)
{
  int var 2;
  ...
}
```

11 - Entrées/sorties formatées

11.1 - Fonction « printf »

Afin de pouvoir développer nos premiers programmes nous auront au moins besoin d'afficher des informations. Pour cela comme nous l'avons déjà vu, nous utilisons la fonction « printf » :

```
printf("format",expr1,expr2,...) ;
```

- La chaîne « format » contient une chaîne de caractères avec potentiellement des formateurs de variables,
- Les termes « expr1 », « expr2 », … représentent les variables ou expressions qui seront utilisé dans les formateurs,

Exemple :

```
int nombre = 3;
printf("Chiffre : %d\n",nombre);
```

Les formateurs sont les suivants :

- **%d** ou **%i** : entier relatif (int),
- **%u** : entier naturel (unsigned int)
- **%o** : entier exprimé en octal (int)
- **%x** : entier exprimé en hexadécimal (int)
- **%c** : caractère
- **%f** : rationnel en notation décimale (double ou float)
- **%e** : rationnel en notation scientifique (double ou float)
- **%s** : chaîne de caractères

On peut donc avoir par exemple :

```
double M = 12.123456789;
int d = 4;
printf("Reel:%f, Reel:%e, calcul:%d\n", M, M, d*3);
```

Certains formateurs additionnels peuvent être ajoutés au formateur principal pour modifier la longueur du type indiqué :

- **`%hh`** : entier naturel 8 bits (char),
- **`%h`** : entier naturel court (short int),
- **`%l`** : entier naturel long (long int),
- **`%ll`** : entier naturel long long (long long int),
- **`%L`** : réel de type long (long double),
- **`%z`** : type taille (size_t)

Il est possible de personnaliser le formatage en utilisant la syntaxe suivante :

```
%[flag][largeur][.precision][modif]type
```

Avec :

- `[flag]` : Indicateur de signe et d'alignement, « - » : alignement à gauche, « + » : ajout du signe + ou -, « espace » : ajout du signe si négatif et espace si positif,
- `[largeur]` : Le nombre de caractères minimums à utiliser pour l'affichage comblés par des espaces,
- `[.precision]` : La précision après la virgule ou pour les chiffres entiers le nombre minimum de chiffres comblés par des « 0 »,
- `[modif]` : pour modifier l'interprétation de l'expression, « h » : short, « l » : long pour entier ou double sinon, « L » : Long double,

Exemple :

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

int main(void) {
  int a=3;
  double b=3.2;
  printf("(%+5.4d),(%7.4f)\n",a,b);
  return(0);
}
```

Va donner :

```
(+0003),( 3.2000)
```

11.2 - Fonction « scanf »

Pour saisir des informations depuis le clavier, plusieurs fonctions existent, qui seront détaillées dans un prochain chapitre, la plus importante étant la fonction « scanf » :

```
scanf("format",expr1,expr2,…) ;
```

Cette dernière fonctionne d'une manière similaire à la fonction « printf », c'est à dire qu'il faut passer par une chaîne de formatage puis préciser les variables sous forme d'adresse, par exemple pour saisir un entier :

```
int a;
printf("Saisissez un entier : ");
scanf("%d",&a);
printf("Entier :%d",a);
```

Le format d'un formateur est le suivant :

```
%[larg][modif]type
```

Avec :

- `[larg]` : Indique le nombre maximal de caractères à lire,
- `[modif]` : pour modifier l'interprétation de l'expression, « h » : short, « l » : long pour entier ou double sinon, « L » : Long double,
- `type` : Le type qui sont globalement identiques à ceux utilisés par la fonction d'affichage « printf »,

Il est possible de demander plusieurs entrées en une seule fois, séparées par des espaces, ou tout autre caractère qui devra alors être utilisé pour donner les valeures.

Exemple :

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

int main(void) {
  int a=3;
  float b=3.2;
  printf("Entrez un entier et un flottant :");
  scanf("%d %f",&a,&b);
  printf("Valeurs : (%d) (%f)\n",a,b);
  return(0);
}
```

Donnera :

```
Entrez un entier et un flottant :3 4.56
Valeurs : (3) (4.560000)
```

Chapitre 2 - Opérateurs et expressions

Table des matières

1 - Opérateurs arithmétiques

En langage C, il existe 5 opérateurs arithmétiques :

- `+` : L'addition,
- `-` : La soustraction,
- `*` : La multiplication,
- `/` : La division,
- `%` : Le reste de la division ou modulo,

Les trois derniers sont prioritaires sur les deux premiers comme en mathématique, mais il est possible d'utiliser des parenthèses, c'est d'ailleurs conseillé :

- `3 * 2 + 4 / 2` → donne 8
- `3 *( 2 + 4) / 2` → donne 9

2 - Incrémentation et décrémentation

Dans le langage C, des opérations spécifiques d'incrémentation et de décrémentations existent sous la forme de notation suivante :

- -- : pour la décrémentation,
- ++ : pour l'incrémentation,

Comme nous le verrons après, ces opérations sont prioritaires sur le reste de la chaîne de traitement. De plus, la position de ces opérations change en fonction de la précédence des symboles, ainsi si i vaut 1 :

- i++ : possède comme valeur 1, et « i » vaut 2, l'incrémentation étant réalisée après,
- ++i : possède comme valeur 2, et « i » vaut 2, l'incrémentation étant réalisée avant,

3 - Opérateurs d'affectation

En langage C, il existe 6 opérateurs d'affectation :

- = : Affecte la partie de droite à la variable de gauche,
- += : Ajoute le contenu de la partie de droite à la variable de gauche,
- -= : Soustrait le contenu de la partie de droite à la variable de gauche,
- *= : Multiplie le contenu de la partie de droite à la variable de gauche,
- /= : Soustrait le contenu de la partie de droite à la variable de gauche,
- %= : Prend le module de la partie de gauche par la partie de droite et stocke le reste dans la variable de gauche,

Ainsi en terme de résultat, il est équivalent d'écrire `i=i+1`, `i+=1` et `i++`.

De plus, une affectation possède également une valeur qui est égale à la partie de gauche, et qui peut donc être réutiliser :

```
a=b=c*3+8
```

4 - Opérateurs de comparaison

Il existe un certain nombre d'opérateurs permettant de comparer des expressions :

- `==` : Partie de gauche égale à la partie de droite,
- `!=` : Partie de gauche différente de la partie de droite,
- `<` : Partie de gauche strictement inférieure à la partie de droite,
- `>` : Partie de gauche strictement supérieure à la partie de droite,
- `<=` : Partie de gauche inférieure ou égale à la partie de droite,
- `>=` : Partie de gauche supérieure ou égale à la partie de droite,

Ainsi, si i=4 :

- `3>5` : Est vraie,
- `i>=3` : Est vraie,
- `i!=4` : Est faux,

5 - Opérateurs logiques : ET, OU, négation

Il est possible de cumuler plusieurs expressions conditionnelles ou d'utiliser la négation :

- `&&` : Représente le ET conditionnel,
- `||` : Représente le OU conditionnel,
- `!` : Représente la négation,

Exemples avec a=2, b=3 et c=4 :

- `(a + 4) != 3` : (a + 4) différent de 3, vrai ici
- `(a < b) && (b < c) : b` supérieur à a et inférieur à c, vrai ici,
- `a < b && b < c` : même chose car < est prioritaire sur &&
- `!(b < c)` :b n'est pas inférieur à c, faux ici

6 - Mécanismes de fonctionnement des expressions logiques

Une expression logique peut donc contenir une ou plusieurs conditions, dont le résultat est vrai ou faux. Par contre, en langage C le type booléen n'existe pas de base, la valeur de l'expression est donc :

- 0 : Si le résultat est faux,
- 1 (ou différent de 0) : Si le résultat est vrai,

Ainsi, le morceau de programme suivant va afficher « 1 » :

```
int A = 3;
int B;
B = A<4;
printf("%d\n",B);
```

7 - Expressions logiques dans les instructions

Les expressions conditionnelles ne sont en général pas utilisées directement dans les opérations d'affectation, mais utilisées dans des expressions logiques, notamment dans le « SI ».

Par exemple pour réaliser un test logique basé sur une expression conditionnelle, on pourra utiliser :

```
if((a < b) && (b < c))
{
  /* Code correspondant au test */
}
```

Le chapitre spécifique nommé « Structures de contrôles » détaillera l'ensemble des instructions liées aux expressions logiques.

8 - Opérateurs binaires

En langage C il est possible de manipuler directement les objets au niveai binaire à l'aide des opérateurs suivants :

- & : ET binaire,
- | : OU binaire
- ^ : OU EXCLUSIF binaire (XOR),
- << : Décalage binaire à gauche, remplit avec des 0,
- >> : Décalage binaire à droite, remplit avec des 0 ou des 1 suivant le signe,
- ~ : Complément à 1,

Il est à noter que les opérations binaires doivent être réalisées sur des objets adéquats, comme les entiers, et non sur des doubles ou des flottants.

Ainsi pour faire un masque binaire, il est possible d'écrire :

```
b = a & 0x0F ;
```

Et pour un décalage de deux bits :

```
b = a << 2 ;
```

Certaines opérations binaires sont très utilisées pour tester la présence de certains bits dans des appels systèmes. Ici, le test est vrai si la variable « result » contient bien les bits présents dans le « MASK ».

```
if((result&MASK)==MASK)
```

Dernier exemple ou on place les 6 bits de droite à 0 (complément à 1) :

```
b = a & ~077
```

9 - Précédence et associativité des opérateurs

Un certain nombre d'opérateurs possèdent les mêmes caractéristiques de priorité que celles utilisés classiquement en mathématique : + - * / …

Pour d'autres, la priorité n'est pas toujours évidente, la liste est donc donnée après mais d'une manière générale, il est toujours préférable de parenthèser les expressions.

De plus, les opérateurs possèdent un sens d'évaluation, appelée associativité, qui est en général de gauche à droite, mais certains opérateurs sont évalués dans le sens inverse, c'est le cas par exemple de l'opérateur d'affectation « = ».

Les opérateurs sont classés de la manière suivante par ordre décroissant de priorité :

- `() []` : Gauche vers Droite
- `! ~ ++ -- -` : **Droite vers gauche**
- `(conversion)` : **Droite vers gauche**
- `* / %` : Gauche vers Droite
- `+ -` : Gauche vers Droite
- `<< >>` : Gauche vers Droite
- `< <= > >=` : Gauche vers Droite
- `== !=` : Gauche vers Droite
- `&` : Gauche vers Droite
- `^` : Gauche vers Droite
- `|` : Gauche vers Droite
- `&&` : Gauche vers Droite
- `||` : Gauche vers Droite
- `?:` : **Droite vers gauche**
- `= += -= *= /= %= >>= <<= &= ^= |=` : **Droite vers gauche**

10 - Opérateur ternaire

L'opérateur ternaire est une spécificité du langage C, et représente en fait une affectation liée à une condition de manière condensée :

```
(condition) ? (valeur si vraie) : (valeur si faux)
```

Par exemple, pour récupérer le maximum de deux valeurs, on peut utiliser directement l'expression suivante :

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

Cette manière d'écrire permet de condenser des tests simples, mais il ne faut pas en abuser sous peine de perdre en visibilité …

11 - Autres considérations

11.1 - Conversions implicites

Lorsqu'une expression est réalisée en langage C, il est conseillé d'utiliser le même type d'objets pour l'ensemble des opérandes. Toutefois, dans le cas ou cela n'est pas possible ou souhaité, le compilateur intègre la notion de conversion implicite.

La règle alors est que les opérandes sont converties par rapport à celui étant le plus grand, et les signés en non signés, par exemple :

```
int i=2;
float j=3.0,total;
total=j/i;
```

Le compilateur va analyser les deux opérandes « i » et « j », il va donc convertir « i » en flottant et réaliser la division, le total sera égal à 1,5.

Autre exemple :

```
int i=2,j=3;
float total;
total=j/i;
```

Ici les deux opérandes sont de types entiers, il va donc réaliser la division entière et placer ensuite le résultat dans la variable total qui vaudra ici 1,0.

Il faut ajouter à cela que les conversions implicites peuvent poser également des problèmes pour des valeurs non signées comparées avec des valeurs signées :

```
int i=-2;
unsigned int j=3;
float total;
total=j*i;
```

11.2 - Conversions explicites

Comme nous l'avons vu précédemment, il est préférable lorsque l'on doit utiliser la conversion, d'utiliser la conversion explicite.

Pour cela, il suffit d'utiliser l'opérateur de conversion caractérisé par les symboles « (*type*) » devant la valeur, on parle également de « cast ».

Si on reprend notre exemple de conversion implicite de division d'entiers, cela nous donnerait :

```
int i=2,j=3;
float total;
total=(float)j/i;
```

Ici, la conversion s'applique uniquement à la variable « j », ce qui introduit une opérande de type « float », ce qui du coup convertit l'autre opérande en type « float ».

Chapitre 3 - Structures de contrôle

Table des matières

1 - Notion de blocs

Il est possible de regrouper plusieurs instructions dans un bloc définit par des accolades :

```
int b;

{
  int a;
  b = a + 2;
}
```

Les particularités des blocs sont les suivantes :

- Un bloc est considéré comme une seule instruction, ce qui est utile dans les structures de contrôles,
- Les variables déclarées à l'intérieure d'un bloc ne sont pas visibles à l'extérieur, dans notre exemple la variable « a »,
- Syntaxiquement, toute instruction peut être remplacée par un bloc d'instructions,

2 - Structure conditionnelle « if »

Le langage C permet évidemment d'utiliser une structure conditionnelle « si » caractérisée par la syntaxe suivante :

```
if (<condition>) <instruction>;
```

Si contre-condition :

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

où :

- `<condition>` : Indique la condition ou l'ensemble des conditions devant satisfaire au test, comme décrit dans le chapitre précédent, si la condition est vraie le test s'applique,
- `<instruction>` : désigne l'instruction ou le bloc d'instruction qui sera exécuté,

Exemple 1 : Si a est plus petit que 2 on l'incrémente, sinon on le décrémente.

```
if (a<2) a++ ;
else a-- ;
```

Exemple 2 : Si plusieurs instructions il faut utiliser des blocs.

```
if (a<2) { a++; printf("a :%d\n",a); }
else a--;
```

Exemple 3 : On préférera en général la syntaxe suivante.

```
if (a<2) {
  a++;
  printf("a :%d\n",a);
}
else a--;
```

Exemple 4 : Il est possible d'imbriquer les « if ».

```
if(a<2) a++ ;
else if(a>4) a-- ;
```

Exemple 5 : Attention aux « else » dans les « if » imbriqués, ici le « else » s'applique au deuxième « if ».

```
if (a<2)
if (a>0) a++;
else a--;
```

Exemple 5 : On préférera utiliser des blocs et indenter pour éviter les ambiguïtés.

```
if (a<2) {
  if (a>0) a++;
  else a--;
}
```

3 - Structure de choix multiple

Lorsqu'il est nécessaire de comparer une expression avec de multiples valeurs, il est plus simple d'utiliser la structure de choix multiple « switch » dont la syntaxe est la suivante :

```
switch(<expression>) {
  case <constante> : <suite d'instructions> break ;
  case <constante> : <suite d'instructions> break ;
  ...
  default : <suite d'instructions>
}
```

Plusieurs remarques :

- Le cas « default » n'est pas obligatoire,
- L'instruction « break » permet de stopper le « switch » et d'en sortir,
- L'instruction « break » n'est pas syntaxiquement obligatoire, dans ce cas la suite des « case » est exécutée,

Exemple 1 : La variable « a » est de type « int »

```
switch(a) {
  case 0 : a++;break;
  case 1 : a--;break;
  default : printf("Erreur a : %d\n",a);
}
```

Exemple 2 : La variable « touche » est de type « char » et on teste donc si la valeur est q, Q, e ou E sans utiliser de « break » :

```
switch(touche) {
  case 'q' :
  case 'Q' :
  case 'e' :
  case 'E' : exit(0);
}
```

4 - Les structures de boucles « while »

En langage C il n'existe pas de boucle « jusqu'à » mais deux formes de boucles « tant que » qui sont les suivantes :

```
while(<condition>) <instruction>;
do <instruction>; while(<conditions>) ;
```

où :

- `<condition>` : Indique la condition ou l'ensemble des conditions devant satisfaire au test, comme décrit dans le chapitre précédent, si la condition est vraie le test s'applique,
- `<instruction>` : désigne l'instruction ou le bloc d'instruction qui sera exécuté,

La différence fondamentale entre les deux est que dans le deuxième cas, la boucle sera exécutée au moins 1 fois.

Exemple 1 : Affichage d'un tableau

```
while(i<MAX) printf("%d\n",tab[i]);
```

Exemple 2 : Attente que l'utilisateur tape la touche 'q'

```
do touche=getchar(); while(touche!='q');
```

Exemple 3 : Attente que l'utilisateur tape la touche 'q' avec 3 essais

```
while((i<=3)&&(touche!='q')) {
  touche=getchar();
  i++;
}
```

5 - La structure de boucle « for »

Le structure de la boucle « pour » en langage C est la suivante :

```
for(<initialisations>;<conditions>;<opérations>) <instruction>
```

Avec :

- <initialisation> : L'initialisation de la variable, ou des variables,
- <conditions> : La condition ou l'ensemble de conditions pour continuer la boucle,
- <opérations> : Les opérations à réaliser à la fin de chaque itération,
- <instruction> : L'instruction ou le bloc d'instructions à exécuter,

La boucle « for » peut se traduire en boucle « while » de la manière suivante :

```
<initialisations>
while(<conditions>) {
  <instruction>
  <opérations>
}
```

Exemple 1 : Une boucle simple

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

Exemple 2 : Avec déclaration de variable et double conditions

```
for(int i=0 ; i<10 && tablo[i]!=4 ; i+=2) tablo[i]=0;
```

Exemple 3 : Plusieurs déclarations, conditions et opérations

```
for(int i=0,j=10 ; i!=13 && tablo[j]!=-1 ; i++,j+=2) {
  printf("%d\n", tablo[j]) ;
  tablo[j]=0 ;
}
```

6 - Instructions de contrôle de boucles

Nous avons déjà rencontré l'instruction « break » dans la structure de contrôle « switch ». Cette instruction est généralisable à l'ensemble des structures de contrôle et permet de sortir de cette dernière. Exemple d'une sortie immédiate de la boucle si la valeur est trouvée :

```
for(i=0;i<10;i++) {
  if(tab[i]==MAX) break;
  printf("Indice %d ne convient pas\n",i);
}
```

De la même manière, l'instruction « continue » permet de passer à l'itération suivante d'une boucle sans exécuter la suite :

```
for(i=0;i<10;i++) {
  if(tab[i]==0) continue; /* on n'execute pas printf */
  printf("Indice %d ne convient pas\n",i);
}
```

Chapitre 4 - Tableaux, pointeurs et chaînes de caractères

Table des matières

1 - Les tableaux

Pour déclarer un tableau en Langage C il faut utiliser l'opérateur « [] » qui contiendra la taille de ce dernier, par exemple pour un tableau de 10 entiers :

```
int t_int[10];
```

On pourra ensuite accéder à chaque élément du tableau indépendamment les uns des autres, avec un indice de 0 à 9 :

```
t_int[0] = 3 ;
t_int[2]=t_int[0]+2 ;
```

Pour un tableau à deux dimensions, ici 5 par 3 :

```
float t_float[5][3] ;
```

Qu'on utilisera de la manière suivante :

```
t_float[2][3] = 5 ;
```

Il est possible d'initialiser le contenu des tableaux lors de leurs déclarations :

```
int t_int[10] = {1,2,3,4,5,6,7,8,9,10 } ;
float t_float[3][2] = { {0, 1}, {1, 0}, {2, 5}};
```

Il faut noter par contre, que pour recopier le contenu d'un tableau dans un autre il n'est pas possible d'utiliser le symbole d'égalité :

```
int s[10] ;
int v[10] ;
s = v     /* NON */
```

En effet, une variable représentant un tableau contient en fait l'adresse du premier élément de ce dernier. Pour recopier un tableau, il est nécessaire de passer par une ou plusieurs boucles. Par contre on a donc le droit d'écrire :

```
int * p = s ;
```

2 - Les pointeurs

En langage C, il est est possible de déclarer des variables pouvant contenir des données, mais aussi de manipuler les adresses d'objets déclarés, appelés pointeurs.

Le Langage C introduit donc deux opérateurs spécifiques :

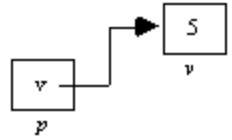
- * : Opérateur de contenu (pointé par une adresse),
- & : Opérateur d'adresse (d'un objet en mémoire),

Ainsi, pour obtenir l'adresse d'une variable existante « v », on utilisera la notation suivante : « &v ».

Inversement, pour obtenir le contenu de la zone mémoire pointée par la variable « p », on utilisera la notation : « * p ».

Dans ce premier exemple, nous déclarons une variable « v » en mémoire, et un pointeur « p », qui va contenir l'adresse de cette variable :

```
int v=5 ;
int * p = &v ;
```



Ainsi, les expressions suivantes sont équivalentes :

- *p + 3
- v + 3

3 - Calculs sur les pointeurs

Il est possible d'utiliser certaines opérations sur les pointeurs comme l'addition et la soustraction, ainsi les instructions suivantes sont valides :

```
p = p+1 ;
p-- ;
```

En fait, dans le premier cas, le pointeur est décalé à droite en mémoire de la taille de l'objet sur quoi il est censé pointé (int, float, ..), alors que dans le deuxième cas, il est décalé à gauche. Exemple :

```
int i[10] ;
int * p = i ;
p++ ;
*p = 3 ;
```

Ici, le pointeur « p » pointe initialement sur le premier élément du tableau, puis après incrémentation, sur le deuxième, qu'on initialise, soit l'équivalent de « i[1]=3 ».

Deuxième exemple :

```
int i[10] ;
int * p = &(i[3]) ;
*(p+2) = 4 ;
```

Ici, on fait pointer « p » sur la 4ème case du tableau « i », puis on initialise la 6ème à l'aide de « *(p+2) ».

Initialisation

Un pointeur, comme toute variable, possède un contenu indéfini. Il est donc primordial de l'initialiser afin d'éviter de modifier des espaces mémoires involontairement. Si on ne connaît pas encore l'emplacement de ce dernier, on utilisera le terme « NULL » (constante prédéfinie) :

```
int *p = NULL ;
```

4 - Les tableaux dynamiques

La taille d'un tableau n'est pas forcément connu au démarrage d'un programme, et il peut être nécessaire d'allouer dynamiquement de la mémoire, pour cela on utilisera la fonctions « malloc », « realloc » et « calloc » :

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

Exemple :

```
int * p ;
p = (int *)malloc(sizeof(int)*20) ;
p[2]=3 ;
```

Ici, on déclare un pointeur « p », qu'on affecte de la valeur reçu par la fonction « malloc », à qui on a demandé d'allouer un espace contigu de 10 entiers. On remarquera qu'ensuite, l'espace s'utilise comme un tableau.

Il est important de préciser que cet espace mémoire ne sera jamais détruit automatiquement, il est nécessaire de réaliser l'opération par un appel à la fonction « free » :

```
#include <stdlib.h>
void free(void *ptr);
```

Ainsi nous aurons :

```
free(p) ;
```

Pour les autres fonctions d'allocation dynamique :

- `calloc(nmemb, size)` : Utilise « Malloc » pour allouer « nmemb » de taille « size »,
- `realloc(ptr, size)` : Ré-alloue l'espace mémoire pointé par « ptr » avec une nouvelle taille « size ».

5 - Pointeurs doubles

Nous avons vu qu'un pointeur contenait l'adresse d'un élément en mémoire, qui pouvait donc contenir une donnée. Il se peut donc également, que la donnée elle même soit un pointeur, et nous avons donc à faire ici à un double pointeur.

Un tel pointeur se note donc avec deux caractères « * », par exemple pour un pointeur double sur un entier : « int ** p ».

Un tel pointeur peut être utilisé par exemple :

- Pour passer l'adresse d'un pointeur à une fonction au cas où il doit être modifiée,
- Pour déclarer un tableau dynamique de dimension 2,
- ...

Un tableau à deux dimensions peut être vu comme un pointeur sur un tableau à une dimension, contenant pour chaque case des adresses sur un autre tableau :

```
#include <stdlib.h>

int main() {
  int x=10,y=20,i,j;
  int ** tab;

  tab=(int **)malloc(sizeof(int *)*x);
  for(i=0;i<x;i++) tab[i]=(int *)malloc(sizeof(int)*y);

  for(j=0;j<y;j++) {
    for(i=0;i<x;i++) {
      /* Travail */
    }
  }

  for(i=0;i<x;i++) free(tab[i]);
  free(tab);
  return(0);
}
```

6 - Chaînes de caractères

Dans le langage C, une chaîne de caractère n'existe pas et une succession de caractères est donc représentée par un tableau, ainsi pour déclarer une chaîne de 10 caractères :

```
char chaine1[10] ;
```

On peut initialiser la chaîne au moment de sa déclaration :

```
char chaine1[100]="coucou";
char chaine2[7]="coucou";
char chaine3[]="coucou";
char * chaine4="coucou";
```

Dans les deux premiers cas, il y a une demande explicite de la taille du tableau, alors que dans les deux derniers, l'espace est réservé automatiquement.

Il faut noter qu'une chaîne de caractère doit se terminer par « \0 » (le zéro binaire) pour être traitée correctement. En effet, le système doit savoir quand se termine la chaîne en question. Il faut donc toujours ajouter 1 octet lors de la déclaration d'une chaîne de caractère.

Pour connaître la taille d'une chaîne de caractère, il faut donc parcourir le tableau jusqu'à trouver le caractère « \0 », ou utiliser la fonction :

```
#include <string.h>
size_t strlen(const char *s);
```

Ainsi, dans l'exemple précédent, en ajoutant la ligne suivante, le programme afficherait « 6 » :

```
printf(" Longueur :%ld\n",strlen(chaine4)) ;
```

ATTENTION : Ceci est la taille de la chaîne, et non la taille du tableau !!

7 - Manipulation de chaînes de caractères

7.1 - Copie

Comme les chaînes de caractères ne sont pas des types mais des tableaux, les opérateurs classiques ne peuvent pas fonctionner, comme l'affectation :

```
chaine1=chaine2 ;
```

Il faut donc passer par les fonctions spécialisées suivantes :

```
#include <string.h>
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
```

Dans le premier cas, l'intégralité de la chaîne « src » est recopiée dans la chaîne « dst », dans le deuxième cas c'est uniquement les « n » premiers caractères. La valeur retournée est le pointeur sur la destination. Il faudrait donc écrire :

```
strcpy(chaine1,chaine2) ;
```

7.2 - Comparaison

De la même manière que pour la copie, il est nécessaire d'utiliser des fonctions spécifiques pour comparer des chaînes de caractères :

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
```

Les deux fonctions permettent respectivement de comparer les chaînes complètes, ou les « n » premiers caractères.

Ces fonctions retournent « 0 » si la comparaison est identique, inférieur à « 0 » ou supérieur à « 0 » si elles sont différentes :

```
If (strcmp(chaine1,chaine2)==0) printf("Identiques!!\n ») ;
```

7.3 - Concaténation

Des fonctions spécifiques sont également disponibles pour concaténer des chaînes de caractères :

```
#include <string.h>
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
```

La première fonction va concaténer l'intégralité de la chaîne « src » à la suite de la chaîne « dst », alors que la deuxième ne va concaténer que les « n » premiers caractères.

7.4 - Autres fonctions

Il existe un multitude d'autres fonctions liées aux chaînes de caractères que nous ne pourrons détailler ici : strcasecmp, strchr, strcoll, strcspn, strdup, strfry, strncasecmp, strpbrk, strrchr, strsep, strspn, strstr, strtok, strxfrm, index, rindex, ...

7.5 - Traitement dans les chaînes de caractères

Il existe des fonctions spécifiques de saisie et d'affichage qui se réalisent dans une chaîne de caractère et non sur la sortie standard, elles sont les suivantes :

```
#include <stdio.h>
int sprintf(char *str, const char *format, ...);
int sscanf(const char *str, const char *format, ...);
```

Ces fonctions ont un comportement identique aux fonctions classiques « printf » et « scanf » sauf qu'elles effectuent leur sortie/entrée dans la chaîne pointé par le premier argument.

8 - Les chaînes de caractères Unicode

8.1 - Wide Character Type

Historiquement, les chaînes de caractères n'étaient prévus que pour fonctionner avec la table ASCII, c'est à dire sur un espace de 256 caractères.

Toutefois, les chaînes de caractères étant aujourd'hui codées avec un type de représentation, la norme C90 à implémenté le type « wchar_t » qui signifie Wide Character Type.

Le problème est que :

- Les types de chaînes de caractères ne sont plus les mêmes,
- Les fonctions ne sont plus les mêmes,
- Les constantes de type chaîne de caractères ne sont plus les mêmes,

Exemple :

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

int main(void) {
  wchar_t chaine[256];

  fwide(stdout,1);
  do {
    wprintf(L"Entrez une chaine : ");
    wscanf(L"%ls",chaine);
    wprintf(L"%d:%ls\n",wcslen(chaine),chaine);
  } while(wcscmp(chaine,L"fin")!=0);
  return(0);
}
```

8.2 - Fonctionnement par défaut

Normalement, sur les systèmes actuels, l'encodage et le décodage des caractères est réalisé de manière native suivant la configuration du système, par exemple :

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

int main(void) {
  char chaine[256];

  do {
    printf("Entrez une chaine : ");
    scanf("%255s",chaine);
    printf("%d:%s\n",(int)strlen(chaine),chaine);
  } while(strcmp(chaine,"fin")!=0);
  return(0);
}
```

Voici une exemple d'utilisation :

```
Entrez une chaine : 10$
3:10$
Entrez une chaine : 10€
5:10€
Entrez une chaine : fin
3:fin
```

Chapitre 5 - Les structures

Table des matières

1 - Définition

Une structure est un groupement de variable, réunis sous la forme d'une entité, permettant de les manipuler plus facilement comme un ensemble.

On utilise le terme « stuct » suivi du nom de la structure :

```
struct <nomstruct> {
  <type1> <nom1>
  <type1> <nom1>
  ...
};
```

Pour ensuite déclarer un objet de ce type :

```
struct <nomstruct> <nom>;
```

Il est également possible de définir la structure et de déclarer un objet du type en une seule opération :

```
struct <nomstruct> {
  <type1> <nom1>
  <type2> <nom2>
  ...
} <nom>;
```

Pour accéder aux éléments, on utilise l'opérateur « `.` » s'il s'agit d'une structure et l'opérateur « `->` » s'il s'agit d'un pointeur de structure :

- `<nom>.<nom1>` : Pour la variable `<nom1>`, `<nom>` est une structure,
- `<ptnom>-><nom1>` : Pour la variable `<nom1>`, `<ptnom>` est un pointeur,

Il est possible également d'initialiser les champs de la structure lors de la déclaration d'un élément de cette dernière.

Exemple 1 : Création d'un type complexe.

```
#include <stdio.h>

int main() {
  struct complexe {
    int reel;
    int imag;
  };
  struct complexe c1,c2;
  c1.reel=3;
  c1.imag=4;
  c2=c1;
  printf("%d, %d\n",c2.reel,c2.imag);
  return 0;
}
```

Remarque : On remarquera que l'opérateur d'affectation fonctionne nativement, et réalise l'affectation champ à champ.

Exemple 2 : Création d'une structure personne avec 3 champs.

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

int main() {
  struct personne {
    char nom[128];
    char prenom[128];
    int age;
  };
  struct personne pers1;
  strcpy(pers1.nom,"Martin");
  strcpy(pers1.prenom,"Pierre");
  pers1.age=35;
  printf("%s, %s, %d\n",pers1.nom, pers1.prenom, pers1.age);
  return 0;
}
```

Exemple 3 : Le même exemple mais en initialisant directement la variable « pers1 » avec les valeurs.

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

int main() {
  struct personne {
    char nom[128];
    char prenom[128];
    int age;
  };
  struct personne pers1 = {"Martin","Pierre",35};
  printf("%s, %s, %d\n",pers1.nom, pers1.prenom, pers1.age);
  return 0;
}
```

2 - Structures imbriquées

Il est possible de définit une structure à l'intérieur d'une autre :

```
struct date {
  short jour;
  short mois;
  short annee;
};
struct personne {
  char nom[128];
  char prenom[128];
  struct date naissance;
  int age;
};
struct personne pers1;
strcpy(pers1.nom,"Martin");
strcpy(pers1.prenom,"Pierre");
pers1.age=35;
pers1.naissance.jour=13;
pers1.naissance.mois=3;
pers1.naissance.annee=1983;
```

De la même manière, il est possible d'initialiser directement :

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

int main() {
  struct date {
    short jour;
    short mois;
    short annee;
  };
  struct personne {
    char nom[128];
    char prenom[128];
    int age;
    struct date naissance;
  };
  struct personne pers1={"Martin","Pierre",35, {13,3,1983}};
  printf("%s, %s, %d, Naissance : %d/%d/%d\n",pers1.nom
              ,pers1.prenom, pers1.age, pers1.naissance.jour
              ,pers1.naissance.mois, pers1.naissance.annee);
  return 0;
}
```

3 - Nouveaux types avec « typedef »

Un type souvent utilisé peut parfois se montrer rébarbatif au niveau de la quantité de caractères à insérer dans le programme, en particulier pour les types de structures.

Le langage C permet donc de définit des nouveaux types à l'aide du mot clef « typedef », qui prend en argument le type d'origine, puis le nom associé au nouveau type.

Par exemple pour définir un nouveau type nommé « ulint » :

```
typedef unsigned long int ulint;
```

Pour définir un nouveau type nommé « s_complexe » basé sur une structure :

```
struct complexe { int reel, imag; };
typedef struct complexe s_complexe;
```

Ou plus simplement :

```
typedef struct complexe { int reel, imag; } s_complexe;
```

Encore plus simplement :

```
typedef struct { int reel, imag; } s_complexe;
```

Pour les utiliser, on fera simplement :

```
ulint var1;
s_complexe c1;
```

4 - Pointeurs sur structures

De la même manière que pour les types standards, il peut être nécessaire de manipuler des pointeurs sur les structures plutôt que les structures elles-mêmes. Dans ce cas, l'opérateur « `->` » est utilisé :

```
#include <stdio.h>

int main() {
  typedef struct complexe {
    int reel;
    int imag;
  } s_complexe;

  s_complexe c1 = { 2, 5 };
  s_complexe * p_c2;
  p_c2=&c1;
  printf("%d, %d\n",p_c2->reel,p_c2->imag);
  return 0;
}
```

5 - Les énumérations

Les énumérations « enum » permettent de déclarer des variables dont les valeurs sont censées être choisies parmi les valeurs données, qui sont en fait représentées en interne par des types « int » ayant une valeur de 0 à n-1, par exemple le programme suivant affichera « 2 » :

```
#include <stdio.h>

int main() {
  enum couleur { ROUGE, VERT, BLEU };
  enum couleur coul1 = BLEU;
  printf("%d\n",coul1);
  return(0);
}
```

Il est possible de spécifier les valeurs prises lors de la déclaration :

```
enum couleur { ROUGE=10, VERT=20, BLEU=30 };
```

6 - Les unions

Lorsque que l'on souhaite pouvoir utiliser différents types de valeurs (int, float, ...), mais pas au même moment, il est possible d'utiliser le type « `union` ».

En effet, ce dernier est composé de plusieurs variables, à la manière d'une structure, mais n'utilise en mémoire que la taille du plus grand élément la composant, il n'est donc possible de ne définir qu'un seul de ces éléments au même moment. Voici un exemple tiré du système Unix :

```
union sigval {
  int   sival_int;
  void *sival_ptr;
};
```

Ici, il n'est donc possible d'utiliser au même moment que :

- L'entier « sival_int »,
- Le pointeur « sival_ptr »,

```
#include <stdio.h>

union sigval {
               int   sival_int;
               void *sival_ptr;
           };

int main() {
  union sigval test;
  test.sival_int = 0;
  printf("%ld %ld %ld\n"
         ,sizeof(union sigval),sizeof(int),sizeof(void *));
  return(0);
}
```

Lorsque le choix du type utilisé ne peut pas être connu a priori, il sera nécessaire d'utiliser une variable supplémentaire à l'extérieur de l'union pour le préciser.

7 - Les champs de bits

Dans une optique d'économiser la mémoire, le langage C permet dans un structure de définir la taille des entiers la composant en nombre de bits.

```
#include <stdio.h>

int main() {
  struct paquet {
    unsigned int flag : 1;
    unsigned int proto : 3;
    unsigned int header : 12;
  };
  struct paquet pa1;
  pa1.flag=0;
  pa1.proto=7;
  pa1.header=5;
  pa1.proto+=2;
  printf("%d %d %d\n",pa1.flag,pa1.proto,pa1.header);
  return(0);
}
```

Dans l'exemple précédent, l'ajout du chiffre 2 au champ proto va faire passer sa valeur à 1.

Il est globalement déconseillé d'utiliser ce mécanisme et il faut savoir que :

- Il n'est possible d'utiliser que des « `int` » et des « `unsigned int` » avec une préférence pour ces derniers,
- Il est impossible d'utiliser l'opérateur « & » avec des champs de bits,
- L'organisation mémoire n'est pas connu, et dépend du compilateur,

Chapitre 6 - Les fonctions

Table des matières

1 - Définition d'une fonction

Une fonction est un morceau de programme nommé qu'il est possible d'appeler une ou plusieurs fois, à partir d'un autre point du programme. Une fonction doit être définie avant d'être appelée

Une fonction se définit comme suit :

```
<type retourné> <Nom fonction>(<parametres>) {
<code fonction>
}
```

Le <type retourné> peut être :

- Un type classique (int, char, …),
- Un pointeur (float *, struct fiche *, …) simple, double ou plus,
- le terme « void » sinon,
- MAIS pas un tableau ...

Pour les paramètres de la fonction :

- Il peut n'y en avoir aucun, on peut mettre une liste vide ou « void » dans ce cas,
- Il peut y en avoir un ou plusieurs défini,
- Il peut y en avoir plusieurs dont le nombre est indéfini,

Exemple 1 : Ici, aucun paramètres d'appels ni de retour, on précise donc « void » comme type de retour, et vide comme paramètres :

```
void usage() {
  printf("Syntaxe du programme :\n");
  printf("./superprog <options> Nombre"\n");
}
```

Exemple 2 : Fonction qui calcule ici la somme de deux entiers et retourne la valeur comme résultat.

```
int somme(int a, int b) {
  return(a+b) ;
}
```

Exemple 3 : Fonction récursive de calcul de factoriel.

```
unsigned long int factoriel(unsigned long int nombre) {
  if(nombre>1) return(nombre*factoriel(nombre-1));
  return 1;
}
```

2 - portée des variables

La portée d'une variable dépend de l'endroit où elle est déclarée :

- **Les variables locales** : Déclarées dans la fonction, sa portée est limitée à cette dernière car allouée sur la pile au moment de l'appel de fonction

```
int somme(int a,int b) {
  int tmp ; /* Variable locale */
  tmp = a + b ;
  return(tmp) ;
}
```

- **Les variables globales** : Déclarées en début de programme, sa portée concerne tout le programme, allouée dans la zone de donnée,

```
int global = 0; /* Variable Globale */
int somme(int a) {
  int tmp; /* Variable locale */
  tmp = a + global;
  return(tmp);
}
```

- **Les variables « static »** : Déclarées avec le mot « static », sa portée reste la fonction dans laquelle elle est déclarée (sauf si on connaît son adresse), mais sa durée de vie est la même que celle du programme, puisque cette variable est allouée dans la zone de donnée.

```
void appel() {
  static int nbappel=0;
  nbappel++;
  printf("%d\n",nbappel);
}
```

3 - Appel d'une fonction

Une fonction s'appelle classiquement par son nom et ses paramètres, ici une fonction somme appelée avec 2 entiers :

```
#include <stdio.h>

int somme(int a, int b) {
  return(a+b);
}

int main() {
  int resultat;
  resultat=somme(3,2);
  printf("%d\n",resultat);
  return(0);
}
```

4 - Passage de paramètres

Lorsque l'on réalise un appel de fonction, on lui transmet des arguments, que l'on appelle passage par valeur. En effet, ces derniers sont une copie de ceux passés en arguments et la fonction ne fait que travailler sur des copies.

```
#include <stdio.h>

void echange(int a, int b) {
  int tmp;
  tmp=a;
  a=b;
  b=tmp;
}

int main() {
  int x=2,y=3;
  echange(x,y);
  printf("%d, %d\n",x,y);
  return 0;
}
```

Si on veut travailler directement sur les contenus, il est nécessaire de communiquer les pointeurs sur ces variables, on travaille alors par adresse.

```
#include <stdio.h>

void echange(int *pta, int *ptb) {
  int tmp;
  tmp=(*pta);
  (*pta)=(*ptb);
  (*ptb)=tmp;
}

int main() {
  int x=2,y=3;
  echange(&x,&y);
  printf("%d, %d\n",x,y);
  return 0;
}
```

Concernant les tableaux, deux notations sont possibles, par exemple pour un tableau d'entier on pourra utiliser :

- `void fonc(int * tab, int taille)`
- `void fonc(int tab[],int taille)`

Cependant, afin d'éviter toute ambiguïté, on préférera utiliser la deuxième notation on l'on comprend bien que c'est un tableau qui est passé et non juste un pointeur.

Il n'est pas autorisé de retourner un tableau au sens strict du terme, par contre il est possible de retourner un pointeur, qui peut donc être utilisé comme un tableau. De toute façon, il serait inutile de retourner l'adresse d'un tableau déclaré dans la fonction ...

```
int * test() {
  int * tab=(int *)malloc(sizeof(int)*10);
  return tab;
}
```

5 - Prototype de fonction

Il n'est pas nécessaire de décrire complètement une fonction pour pouvoir l'utiliser, à partir du moment ou le compilateur connaît au moins son prototype, c'est à dire les caractéristiques de la fonction.

Un prototype de fonction est définit comme la première ligne de définition de la fonction sans le nom des variables, terminée par un point virgule.

Ainsi, pour la fonction « somme » utilisée précédemment, nous aurions le prototype suivant :

```
int somme(int, int);
```

```
#include <stdio.h>

int somme(int, int);

int main() {
  printf("%d\n",somme(3,2));
  return 0;
}

int somme(int a, int b) {
  return(a+b);
}
```

Ce principe est utilisé également pour la définition d'une fonction qui sera déclarée dans un autre fichier, elle le sera alors bien souvent dans un fichier d'entête « .h » comme nous le verrons plus tard.

6 - Pointeur de fonction

Une fonction possède également une adresse, qu'il est possible d'utiliser comme argument par exemple.

Cela peut être utile pour utiliser une fonction générique, qui prendra en paramètre une fonction particulière, par exemple une fonction de tri qui prend en argument une fonction de comparaison.

Nous aurons besoin de savoir définir l'adresse de la fonction, la définition du type, et l'appel. Si on prend l'exemple de fonction précédente on utilisera :

- « somme » : référence l'adresse de la fonction,
- « int (*fctsomme)() » : représente la définition du type,
- « (*fctsomme)(x,y) » : représente l'appel,

```
#include <stdio.h>

int somme(int a, int b) {
  return(a+b);
}

int ptsomme(int x, int y, int (*fctsomme)()) {
  return((*fctsomme)(x,y));
}

int main() {
  printf("%d\n",(*ptsomme)(3,2,somme));
  return 0;
}
```

7 - La fonction "main"

Comme nous l'avons déjà vu, le programme doit posséder une fonction spécifique nommée « main », qui représente le point d'entrée du programme, et doit retourner un entier.

Il ne peut y avoir qu'un seul « main » par programme, mais tous les fichiers « .c » n'en possèdent pas forcément comme nous le verront dans un prochaine chapitre.

Cette fonction spécifique peut également prendre des paramètres optionnels :

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

ou sous Unix :

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

Avec :

- `argc` : Le nombre d'arguments passés sur la ligne de commande,
- `argv` : Un tableau contenant les arguments,
- `arge` : Un tableau contenant les variables d'environnement, le dernier élément est égal au pointeur NULL.

```
#include <stdio.h>

int main(int argc, char * argv[]) {
  int i;
  printf("Le programme possède %d arguments\n",argc);
  for(i=0;i<argc;i++) printf("Argument %d :%s\n",i,argv[i]);
  return 0;
}
```

7.1 - Sortie du programme

Nous avons déjà vu que le programme se termine lorsque :

- La fonction « main » se termine,
- La fonction « main » exécute un « return »,
- Une instruction « exit » est rencontrée,

Le prototype de la primitive « exit » est le suivant :

```
#include <stdlib.h>
void exit(int status);
```

Cette fonction peut être placée à n'importe que position dans le code du programme, c'est à dire dans une fonction ou dans le « main » et provoque la sortie immédiate de ce dernier.

Dans le « main », cette fonction a donc un comportement identique à l'utilisation de la directive « return ».

Il est possible de demander au programme d'exécuter des actions spécifiques à la fin de ce dernier. Pour faciliter ce mécanisme, un pile spécifique peut être utilisée pour y placer des pointeurs de fonctions, qui seront exécutées lors de la sortie du programme.

Le prototype de la fonction pour placer une fonction sur la pile est le suivant :

```
#include <stdlib.h>
int atexit(void (*function)(void));
```

Plusieurs fonctions peuvent êtres placées dans cette pile et l'ordre d'exécution sera donc du sommet de la pile (le dernier placé) à la base (le premier placé).

Exemple d'utilisation :

```
void fin(void) {
  printf("Fin du programme ... \n") ;
}
...
atexit(fin);
```

Chapitre 7 - Compilation séparée, classe d'allocation

Table des matières

1 - Mémoire d'un programme C

Un programme exécuté, ou processus, possède plusieurs zones de mémoire :

- `text` : Le code en lecture seule,
- `data` : Les données globales initialisées,
- `bss` : Les données globales non initialisées,
- `heap` (ou tas) : Les données allouées dynamiquement,
- `stack` (ou pile) : Les variables locales et d'appels de fonction,

Ainsi, les variables globales seront situées dans les zones « data » ou « bss », par contre celles déclarées dans les fonctions le seront dans la pile.

Ceci a une conséquence sur la durée de vie des variables, en effet dans le premier cas les variables existent durant la vie du processus, alors que dans le second elles n'existent que pendant l'exécution de la fonction.

2 - Fonctionnement de la chaîne de production.

Comme nous l'avons déjà vu, la phase de compilation d'un programme C passe en fait par plusieurs étapes successives :

Code source → **Assembleur** → **Objet** → **Exécutable**

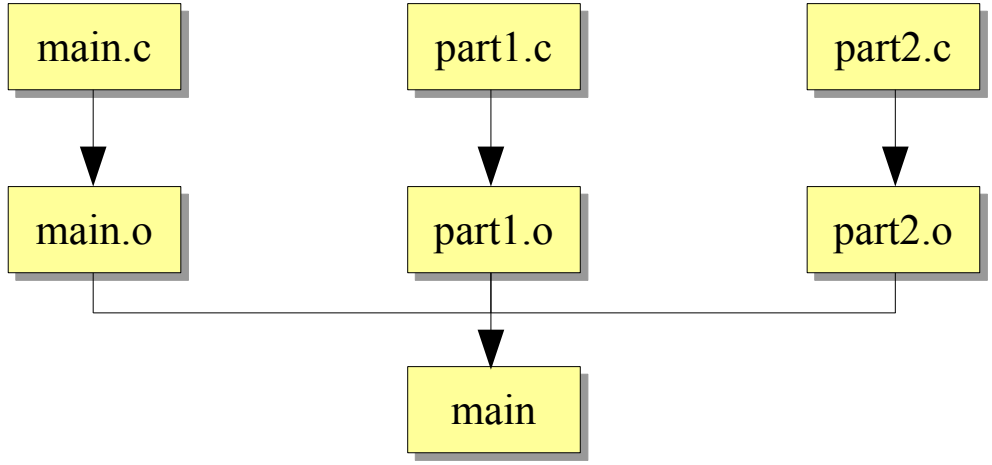
Le fichier objet contient le code binaire associé au programme « .c », mais inutilisable sans la phase d'édition de liens, qui va inclure les bibliothèques utilisés.

Cependant, il peut s'avérer plus simple de décomposer un programme en plusieurs fichiers « .c », et de les compiler séparément.

Ainsi, nous aurons :

- Pour chaque fichier « .c », un fichier objet correspondant,
- Un fichier exécutable basé sur l'ensemble des fichiers objets générés,

Exemple :



Pour compiler avec « gcc » :

```
gcc -c main.c
gcc -c part1.c
gcc -c part2.c
gcc -o main main.o part1.o part2.o
```

Ou plus simple si pas de conservation des fichiers « .o » intermédiaires :

```
gcc -o main main.c part1.c part2.c
```

3 - fichier entête

La séparation seule du code n'est pas suffisante pour pouvoir compiler nos différents programmes.

En effet, lorsque nous aurons besoins d'utiliser une fonction définit dans un autre fichier, le compilateur nous avertira qu'il ne la connaît pas, il est donc nécessaire de déclarer au moins son prototype, ce qui peut être fait dans un fichier d'entête « .h ».

Imaginons que notre fichier « part1.c » contiennent la fonction suivante qui calcule la somme des n premiers entiers :

```
double somme_n(double valeur) {
  return((valeur*(valeur+1))/2) ;
}
```

Il nous faudra donc définir le fichier « part1.h » contenant :

```
double somme_n(double) ;
```

Ainsi, dans le programme principal nous aurons :

```
...
#include "part1.h"
...
int main(void) {
  ...
  resultat=somme_n(8) ;
  ...
}
```

4 - Utilisation de bibliothèque de sources

L'intérêt de découper un programme en plusieurs fichiers sources, est que l'on peut se constituer ses propres bibliothèques de fonctions personnelles sous diverses formes :

- **Fichiers sources** (« .c ») : Qui doivent être recompilés à chaque fois, pour chaque utilisation, l'intérêt principal réside alors dans la portabilité,
- **Fichiers objets** (« .o » ou « .obj ») : On utilise directement les fichiers objets compilés, il ne reste donc plus qu'à réaliser l'édition de liens, méthode plus rapide que la précédente,
- **Librairies statiques** : On constitue une bibliothèque (fichier « .a » ou « .lib ») qui est en fait le regroupement de plusieurs fichiers objets, les librairies statiques sont inclues dans l'exécutable au moment de l'édition de lien comme pour la méthode précédente,
- **Librairies dynamiques** : Toujours une bibliothèque, mais qui possède l'avantage de ne pas être incluse au programme exécutable, cette dernière sera chargée en mémoire lors de son utilisation (fichier « .so » ou « .dll »).

5 - Fichier « Makefile »

5.1 - Introduction

Un fichier de projet « Makefile » est utilisé pour la compilation de programmes complexes (pas nécessairement en C) dont les sources sont réparties en plusieurs fichiers.

Il détermine quelles sont les parties qui doivent être recompilées lorsque des modifications ont étés effectuées et appelle les commandes nécessaires à leur recompilation. Le fichier décrit les relations entre fichiers sources et contient les commandes nécessaires à la compilation.

Une fois écrit le *makefile*, il suffit d'invoquer *make* (pour « gcc ») ou *nmake* (pour Microsoft) pour exécuter toutes les recompilations nécessaires.

5.2 - Le makefile

Le *makefile* est un **ensemble de règles** qui décrivent les relations entre les fichiers sources et les commandes nécessaires à la compilation. Il contient aussi des règles permettant d'exécuter certaines actions utiles comme par exemple nettoyer le répertoire, ou imprimer les sources.

5.3 - Les règle

Une règle *rule* a la forme suivante (les actions doivent être précédées par une tabulation) :

```
cible ...  : dependance ...
      action
      ...
      ...
```

Une cible *target* est en général le nom d'un fichier à générer. Ce peut être aussi le nom d'une action à exécuter. Il y a en général une seule cible par règle.

Une dépendance *dependency* est un fichier utilisé pour générer la cible. Plus généralement les commandes correspondant à une cible sont exécutées lorsque au moins une des dépendances de la cible a été modifiée depuis le dernier appel de *make*. Une cible a en général plusieurs dépendances.

Une action *command* est une ligne de commande Unix qui sera exécutée par *make*. Attention, dans la syntaxe du *makefile*, une action est toujours précédée d'une tabulation.

5.4 - Ecrire un makefile

Voilà, par exemple, un *makefile* pour « gcc » :

```
myprog:   main.o f1.o
        gcc -o myprog main.o f1.o
main.o: main.c
        gcc -c main.c
f1.o: f1.c
        gcc -c f1.c
clean:
        rm myprog main.o f1.o
```

Pour créer l'exécutable myprog, on tapera *make*, pour supprimer les fichiers objets et exécutables du répertoire, on tapera *make clean*.

Le même fichier Makefile pour le compilateur de Microsoft :

```
myprog:   main.obj f1.obj
        link.exe /OUT:myprog.exe main.obj f1.obj
main.obj: main.c
        cl.exe /c main.c
f1.obj: f1.c
        cl.exe /c f1.c
clean:
        del myprog.exe main.obj f1.obj
```

5.5 - Comment le fichier makefile est il interprété ?

Par défaut, c'est la première cible qui est réalisée, « myprog » dans notre exemple. Avant d'exécuter la commande associée, il doit mettre à jour les fichiers qui apparaissent dans ses dépendances (ici main.o et f1.o) puisque ceux-ci apparaissent aussi comme des cibles.

Ainsi il interprète successivement

- La règle associée à main.o, à condition que le fichier « main.c » ait été modifié depuis le dernier appel,
- La règle associée à f1.o, à condition que le fichier « f1.c » ait été modifié depuis le dernier appel,
- La règle associée à « myprog », à condition qu'il ait mis à jour main.o ou f1.o.

5.6 - Exécuter un makefile

La commande *make* interprète par défaut le fichier *makefile* ou *Makefile* si le précédent n'existe pas. On peut donner un nom différent au *makefile* et utiliser l'option -f.

Si aucun argument n'est spécifié, *make* interprète en premier lieu la première cible. N'importe qu'elle autre cible peut être donnée comme argument, auquel cas la règle correspondante est interprétée.

Avec certaines options, *make* peut être utilisé pour exécuter d'autres tâches que celles associées aux règles, par exemple avec l'option -n, il imprime les commandes qu'il devrait exécuter, mais ne les exécute pas, et avec l'option -q, il n'exécute rien et n'imprime rien, mais retourne 0 ou 1 selon que les cibles sont à jour ou non.

5.7 - Des variables dans le makefile

Il peut être intéressant de définir et d'utiliser des variables directement dans le fichier *Makefile* :

```
objects = main.o list.o
cccom = gcc $(ccopts)
ccopts = -Wall -ansi -g

myprog: $(objects)
        $(cccom) -o myprog $(objects)
main.o: main.c table.h
        $(cccom) -c main.c
list.o: list.c table.h
        $(cccom) -c list.c
clean:
        rm myprog $(objects)
```

Substitutions et Noms de variables calculés

Exemple d'utilisation :

```
representation = list
objects = main.o list.o tree.o
$(representation)_obj = main.o $(representation).o
sources = $(objets:.o=.c)
cccom = gcc $(ccopts)
ccopts = -Wall -ansi -g

myprog: $(representation)_obj
        $(cccom) -o myprog $(representation)_obj
main.o: main.c table.h
        $(cccom) -c main.c
list.o: list.c table.h
        $(cccom) -c list.c
tree.o: tree.c table.h
        $(cccom) -c tree.c
```

Les valeurs des variables

On peut définir une variable de différentes manières :

- En spécifiant sa valeur dans le makefile à l'aide de `=', comme on l'a vu précédemment,
- En spécifiant sa valeur sur la ligne de commande, ainsi dans l'exemple précédent de makefile, la ligne de commande :

```
make representation=tree
```

assignera tree à représentation à la place de list et tree.o sera utilisé pour créer l'exécutable.

De plus, make connaît de nombreuses variables qu'il n'est pas nécessaire de définir :

- toutes les variables d'environnement du shell
- certaines variables prédéfinis, comme CC pour la commande de compilation (cc par défaut), ou CFLAGS pour les arguments de cette commande (nulle par défaut).
- certaines variables qui prennent automatiquement une valeur qui dépend de la règle que make est en train d'interpréter. Extraites de la liste de ces variables, voici les plus courantes :

  $@ : le nom du fichier correspondant à la cible courante

  $< : la première dépendance de la cible courante,

  $^ : toutes les dépendances de la cible courante,

  $? : les dépendances de la cible courante qui sont plus récentes que la cible.

Ainsi, si on utilise cc au lieu de gcc, on peut utiliser ce makefile :

```
representation = list
objects = main.o list.o tree.o
$(representation)_obj = main.o $(representation).o
sources = $(objets:.o=.c)
cccom = ${CC} $(ccopts)
ccopts = -Wall -ansi -g

myprog: $(representation)_obj
        $(cccom) -o myprog $^
main.o: main.c table.h
        $(cccom) -c $<
list.o: list.c table.h
        $(cccom) -c $<
tree.o: tree.c table.h
        $(cccom) -c $<
clean:
        rm myprog $(objects)
```

6 - Portée d'une variable

6.1 - Définition

En langage C, le lieu de déclaration d'une variable va influencer sur la **portée** de la variable, c'est à dire l'emplacement dans le programme où la variable est utilisable.

Par défaut :

- Portée locale : Si elle est déclarée dans un bloc, une variable n'existe que dans le bloc où elle est déclarée,
- Portée globale : Si elle est déclarée à l'extérieure de tout bloc, la variable est une variable globale.

On parle également de visibilité d'une variable, surtout dans le cas de définition multiples.

6.2 - Variables locales

Si elle est déclarée dans un bloc, une variable n'existe que dans le bloc où elle est déclarée. Ici la variable « var1 » n'est utilisable que dans la fonction « f1 » :

```
#include <stdio.h>

void f1(void) {
  int var1=1;
  printf("Var1 : %d\n",var1);
}

int main(void) {
  f1();
  return(0);
}
```

Affichage :

```
Var1 : 1
```

6.3 - Variables globales

Si elle est déclarée à l'extérieure de tout bloc, la variable est une variable globale. Ici la variable « var1 » est globale :

```
#include <stdio.h>

int var1=0;

void f1(void) {
  printf("Var1 : %d\n",var1);
}

int main(void) {
  f1();
  return(0);
}
```

Affichage :

```
Var1 : 0
```

6.4 - Visibilité

Si les deux existent, il s'agit de deux variables différentes, pour une meilleure compréhension, ceci est à proscrire :

```
#include <stdio.h>

int var1=0;

void f1(void) {
  int var1=1;
  printf("Var1 : %d\n",var1);
}

int main(void) {
  f1();
  return(0);
}
```

Affichage :

```
Var1 : 1
```

7 - Classes d'allocation des variables

En langage C, il est possible de préciser une classe de stockage lors de la déclaration d'une variable : auto, register, extern et static.

7.1 - Classe « auto »

Cette classe n'est plus utilisée en langage C, car c'est le type implicite quand la classe n'est pas précisée (variables locales et globales précédentes).

7.2 - Classe « register »

Cette classe demande au compilateur de laisser dans la mesure du possible la variable déclarée dans un registre afin d'améliorer les performances. C'est aujourd'hui inutile sauf dans des cas très spécifiques du monde de l'embarqué, car les compilateurs actuels optimisent déjà :

```
register int i=0 ;
```

7.3 - Classe « extern »

Cette classe sert juste à indiquer l'existence d'une variable mais pas à la déclarer, c'est utile dans la compilation séparée où l'on souhaite accéder à une variable définie dans un autre fichier en global (évidemment).

```
/* Fichier1.c */
int global=0 ;
```

```
/* Fichier2.c */
#include <stdio.h>

extern int global;

int main() {
  printf("%d\n",global);
  return(0);
}
```

7.4 - Classe « static »

Cette classe possède plusieurs interprétations en fonction de son utilisation :

- **En global** : Indique que la variable est locale au fichier source dans lequel elle est définit, et ne peut donc pas être utilisé par un autre fichier en utilisant la classe « extern »,
- **Dans un bloc** : Indique que la variable ne sera pas détruite à la fin du bloc, elle est donc déclarée dans la zone de mémoire globale du processus, la portée de la variable reste néanmoins le bloc de déclaration.

Chapitre 8 - Le préprocesseur

Table des matières

1 - Introduction

Comme nous l'avons déjà constaté depuis le début de ce support de cours, un programme en langage C commence par des directives de précompilation.

Ces dernières sont caractérisées par la présence d'un premier caractère de valeur « # », comme « #include », « #define », …

Ces directives ne relèvent pas réellement de la compilation, il sont en fait traités par une première passe du compilateur, appelée précompilation, qui va interpréter ces directives, effectuer les remplacement nécessaires, et les supprimer avant la compilation réelle.

La conclusion directe, est que ces directives ne se traduisent pas directement par du code exécutable, elles ne seront donc pas présentes dans le binaire.

2 - Inclusion de ressources

La première directive de précompilation que nous avons utilisé est l'inclusion de ressource, à l'aide de la directive « #include ».

Cette dernière ne fait **QUE** de prendre le contenu du fichier en question, et de l'inclure tel quel à la place de la directive.

Cette directive possède néanmoins deux syntaxes :

- `#include <chemin/fichier>` : La partie « chemin/fichier » est relatives à l'emplacement des fichiers d'entêtes standards du système, par exemple « #include <stdio.h> »,
- `#include « chemin/fichier »` : La partie « chemin/fichier » est relative au répertoire courant, ou absolu, par exemple « `#include " prog1.h"` » ou « `#include " /tmp/ess.h"` »,

3 - Les constantes

Nous avons vu qu'il était possible de définir des constantes à l'aide de la directive « #define », par exemple :

```
#define TAILLE 100
```

Le pré-compilateur va rechercher chaque occurrence de la chaîne TAILLE dans la programme C et simplement le remplacer par la valeur. On peut donc parfaitement avoir :

```
#define LARGEUR 10
#define HAUTEUR 10
#define TAILLE (LARGEUR*HAUTEUR)
```

Remarque : D'une manière générale, lorsqu'il ne s'agit pas d'une simple valeur, il est préférable de prendre l'habitude de parenthéser la valeur afin d'éviter les problèmes d'ordre des opérations.

Il existe des constantes pré-définies, par exemple :

- `__LINE__` : Qui indique le numéro de la ligne courante dans le programme source,
- `__FILE__` : Qui indique le nom du fichier source,
- `__DATE__` : Qui indique la date de compilation,
- `__TIME__` : Qui indique l'heure de la compilation,

Il est possible à tout moment d'annuler une définition précédente réalisée par « #define » à l'aide de la directive « #undef », c'est utile par exemple pour redéfinir une constante définit précédemment sans obtenir d'avertissement de la part du compilateur, par exemple :

```
#undef TAILLE
#define TAILLE 100
```

4 - Les macros

Il est possible dans une directive « #define » de définir également des macros si la présence de parenthèses dans la partie gauche est détectée, par exemple :

```
#define MAX(a,b) (a>b?a:b)
```

Cette dernière va donc calculer la valeur maximale entre « a » et « b », en fait elle va juste remplacer l'expression de gauche par celle de droite en substituant les variables, par exemple :

```
val = MAX(x,y);
```

Sera remplacé par :

```
val = (x>y?x:y);
```

5 - La compilation conditionnelle

Il est possible de réaliser des opérations conditionnelles, notamment sur le contenu d'une variable, à l'aide de « #if », par exemple :

```
#define VERSION 2

#if VERSION < 1
...
#elif VERSION < 2
...
#else
...
#endif
```

Les opérateurs de comparaison sont ceux classiquement utilisés en langage C, à savoir : <, >, ==, >=, <= et !=.

Pour tester simplement l'existence d'une constante, on utilisera « #ifdef » ou « #ifndef », par exemple en début d'un fichier d'entête « .h », pour éviter de redéfinir plusieurs fois la même chose en cas d'inclusion multiple :

```
#ifndef H_PART1
#define H_PART1
/* Suite du code */
...
#endif
```

Certaines constantes peuvent être définies suivant le compilateur :

- `_WIN32` ou `__WIN32__` : Environnement Windows,
- `linux` ou `__linux__` : Environnement Linux,
- `__APPLE__` ou `__MACH__` : Environnement Apple
- `unix` ou `__unix__` : Environnement Unix en général (Linux, Apple, BSD, ...),

A noter que si l'on souhaite tester l'existence de plusieurs variables en même temps, on utilisera la directive « defined(...) » :

```
#if defined(_WIN32) || defined(__WIN32__)
/* Nous sommes sous Windows */
...
#elif defined(unix) || defined(__unix__)
/* Nous sommes sous Unix */
...
#else
/* On ne sait pas ? */
#error "Systeme inconnu"
#endif
```

Remarque : On pourra remarquer l'utilisation de la directive « #error » qui permet justement d'indiquer une erreur de compilation et de stopper immédiatement cette dernière avec une erreur.

6 - Autres directives

Il est possible de concaténer des éléments dans une macro à l'aide de l'opérateur « ## », par exemple :

```
#define variable(x) x ## _cd
```

Qu'on utilisera dans le programme avec par exemple :

```
int variable(tmp);
```

Et qui sera équivalent à la déclaration suivante :

```
int tmp_cd ;
```

Chapitre 9 - Les bibliothèques standard

Table des matières

1 - Les fonctions de gestion de la mémoire

Nous avons déjà vu au chapitre 4 les fonctions d'allocation dynamique dans le cadre des déclarations de tableaux avec les fonctions suivantes :

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
void free(void *ptr);
```

Qui s'utilisent de la manière suivante :

- `malloc(size)` : Alloue un espace de taille « size » octets,
- `calloc(nmemb, size)` : Utilise « Malloc » pour allouer « nmemb » de taille « size »,
- `realloc(ptr, size)` : Ré-alloue l'espace mémoire pointé par « ptr » avec une nouvelle taille « size ».
- `free(ptr)` : Libère l'espace mémoire pointé par « ptr ».

2 - Les fonctions de manipulation de chaînes de caractères

Nous avons déjà vu différentes fonctions liées aux chaînes de caractères :

```
#include <string.h>
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
```

Dont l'utilisation est la suivante :

- `strcpy(dest,src)` : Recopie la chaîne « src » dans la chaîne « dst »,
- `strncpy(dest,src,n)` : Idem pour les « n » premiers caractères de « src »,
- `strcat(dest,src)` : Recopie la chaîne « src » à la suite de la chaîne « dst »,
- `strncat(dest,src,n)` : Idem pour les « n » premiers caractères de « src »,
- `strcmp(s1,s2)` : Compare la chaîne « s1 » et la chaîne « s2 »,
- `strncmp(s1,s2,n)` : Idem pour les « n » premiers caractères,

Les fonctions « strcpy », « strncpy », « strcat » et « strncat » retournent un pointeur sur la chaîne de destination. Les fonctions « strcmp » et « strncmp » retournent la valeur « 0 » si les chaînes sont identiques.

Nous avons également utilisé la fonction suivante qui permet de connaître la taille d'une chaîne de caractères :

```
#include <string.h>
size_t strlen(const char *s);
```

Il existe un multitude d'autres fonctions liées aux chaînes de caractères que nous ne pourrons détailler ici : `strcasecmp, strchr, strcoll, strcspn, strdup, strfry, strncasecmp, strpbrk, strrchr, strsep, strspn, strstr, strtok, strxfrm, index, rindex, …`

3 - La bibliothèque d'entrée-sortie standard

Cet ensemble de fonction d'E/S, très portable, constitue une couche au dessus des primitives systèmes POSIX qui ne seront pas étudiées ici (open/close/read/write). Il permet de diminuer le nombre d'appels système et simplifie la manipulation des fichiers.

3.1 - Le fichier <stdio.h>

- Définition du type « FILE ». A chaque fichier ouvert au niveau de la bibliothèque standard est associé un objet de ce type. Parmi les champs de la structure, on trouvera notamment un descripteur,
- Macro définition de constante. En particulier NULL (pointeur nul), EOF (indication de fin de fichier) et surtout « stdin », « stdout » et « stderr » de type « FILE * ».
- Prototypes de fonctions (fopen, fclose, fread, ...),

3.2 - Ouverture d'un fichier

```
#include<stdio.h>
FILE *fopen(const char *ref, const char *mode);
```

Cause l'ouverture du fichier dont la référence est pointée par « *ref* » et le renvoie d'un pointeur sur l'objet de type FILE associé à ce fichier. En cas d'échec, la fonction renvoie NULL.

Le paramètre « *mode* » précise le mode d'ouverture dont les trois principaux sont:

- r lecture
- w écriture avec troncature ou création
- a écriture en fin de fichier ou création

Il est possible d'ajouter la lettre « b » au mode, par exemple « wb », pour indiquer qu'il s'agit d'un fichier binaire.

3.3 - Fermeture d'un fichier

```
#include<stdio.h>
int fclose(FILE *ptr)
```

3.4 - Ouverture bibliothèque et système

Il est possible de récupérer le descripteur (niveau système) associé à un fichier ouvert au niveau de la bibliothèque

```
#include<stdio.h>
int fileno(FILE *ptr)
```

3.5 - Lecture dans un fichier

```
#include<stdio.h>
int   fread(void *buf, size_t taille, size_t nombre, FILE
*ptr);
int  fgetc(FILE *ptr);
char *fgets(char *buf, int taille, FILE *ptr);
```

La fonction « *fread( )* » lit « *nombre* » objets de « *taille* » caractères qu'elle écrit à partir de l'adresse « *buf* » et renvoie le nombre d'objets lus.

La fonction « *fgetc( )* » lit un caractère. Une fin de fichier ou une erreur est signalée par la valeur EOF.

La fonction « *fgets( )* » lit une chaîne d'au plus « *taille-1* » caractères, un caractère ASCII nul étant ajouté en dernière position, et renvoie « *buf* », l'adresse de cette chaîne. (NULL si échec).

Cette fonction est préconisé comme remplacement de la fonction « gets ».

```
#include<stdio.h>
int fscanf(FILE *ptr, const char *format,…);
```

Cette fonction réalise une lecture formatée et renvoie soit le nombre de conversion réalisées, soit EOF en cas de fin de fichier ou d'erreur.

Il faut se rappeler qu'à partir du troisième paramètre, on passe des **adresses** d'objets !

La fonction *scanf* est un cas particulier de cette fonction avec un premier paramètre égal à *stdin*.

3.6 - Ecriture dans un fichier

```
#include<stdio.h>
int fputc(int c, FILE *ptr);
int fputs(char *buf, FILE *ptr);
int fwrite(void *buf, size_t taille, size_t nombre, FILE
*ptr);
```

La fonction *fputc* écrit le caractère (unsigned char)*c*.

La fonction *fputs* écrit la chaîne pointée par *buf*, sans recopier le caractère ASCII nul de fin de chaîne.

La fonction *fwrite* écrit *nombre* objets de *taille* caractères qu'elle a trouvés à partir de l'adresse *buf* et renvoie le nombre d'objets écrits.

Toutes ces fonction renvoient une valeur négative en cas de problème.

```
#include<stdio.h>
int fprintf(FILE *ptr, const char *format,…);
```

Cette fonction réalise une écriture formatée et renvoie le nombre de conversion réalisées, ou une valeur négative en cas d'erreur.

Contrairement à *fscanf*, à partir du troisième paramètre, on passe des **noms** d'objets !

La fonction *printf* est un cas particulier de cette fonction avec un premier paramètre égal à *stdout*.

3.7 - Déplacement dans un fichier

Il est possible de se déplacer dans un fichier à l'aide de la fonction « fseek » :

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence);
```

Avec :

- `stream` : Le fichier ouvert,
- `offset` : Le déplacement par rapport à « whence » en octets,
- `whence` : Position de référence du déplacement ou SEEK_SET : pour le début de fichier, SEEK_CUR : pour la position courante, SEEK_END : pour la fin du fichier.

3.8 - Fin de fichier

Il est possible de tester la fin de fichier à l'aide de la fonction « feof » qui retourne un nombre positif si c'est le cas :

```
#include <stdio.h>
int feof(FILE *stream);
```

Exemple d'utilisation :

```
while(!feof(monfichier)) {
  fread(buffer,10,1,monfichier) ;
  ...
}
```

Exemple :

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

int main(void) {
  FILE * monfichier;
  char chaine[256];

  monfichier=fopen("sortie.txt","w");
  if(monfichier==NULL) {
    printf("Erreur de fichier ...\n");
    exit(1);
  }
  do {
    printf("Entrez une chaine : ");
    scanf("%255s",chaine);
    fprintf(monfichier,"%s\n",chaine);
  } while(strcmp(chaine,"fin")!=0);
  fclose(monfichier);
  return(0);
}
```

Dont l'exécution donne :

```
Entrez une chaine : coucou
Entrez une chaine : baba
Entrez une chaine : fin
```

Le contenu du fichier « sortie.txt » sera alors :

```
coucou
baba
fin
```

3.9 - Remarques

Toutes les fonctions précédentes peuvent fonctionner avec l'entrée/sortie standard, il suffit d'indiquer « stdin » pour un flux d'entrée et « stdout » pour un flux de sortie.

4 - Fonctions intéressantes

4.1 - Génération aléatoire

Il est possible de générer des nombres aléatoires à partir des fonctions suivantes :

```
#include <stdlib.h>
int rand(void);
int rand_r(unsigned int *seedp);
void srand(unsigned int seed);
```

La génération de chiffre aléatoire est impossible par un algorithme, au mieux on peut se rapprocher de ce qu'on appelle du « pseudo-aléatoire ». Ce dernier demande une graine de perturbation qui servira d'initialisation, ce qui est fait par la fonction « srand() », puis un chiffre sera généré à chaque appelle à la fonction « rand() ».

```
int a ;
srand(2039) ;
a=rand()%10;
```

4.2 - valeur absolue

La fonction de valeur absolue n'est pas définie dans la librairie mathématique mais directement dans la librairie standard :

```
#include <stdlib.h>
int abs(int j);
long int labs(long int j);
long long int llabs(long long int j);
```

Exemple :

```
if(a==abs(a)) then printf("a est positif") ;
else printf("a est négatif") ;
```

5 - La librairie mathématique

La librairie mathématique s'utilise en compilant le programme avec l'option d'utilisation de la librairie « libm », donc par l'option suivante :

```
gcc -lm -o exec prog.c
```

Et en incluant le fichier d'entête « math.h » par la directive :

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

Cette librairie contient la définition des fonctions suivantes :

- racine carré : sqrt, sqrtd et sqrtl,
- Puissance : pow, powf et powl
- Reste de la division : fmod, fmodf et fmodl,
- Entier inférieur : floor, floorf et floorl
- Valeur absolue : fabs, fabsf et fabsl,
- Entier supérieur : ceil, ceilf et ceill
- Exponentielle : exp, expf et expl,

- Logarithme : log, logf et logl
- Logarithme base 10 : log10, log10f et log10l,
- Cosinus : cos, cosf et cosl,
- Sinus : sin, sinf et sinl,
- Tangente : tan, tanf et tanl,
- Arc cosinus : acos, acosf et acosl,
- Arc sinus : asin, asinf et asinl,
- Arc tangente : atan, atanf, atanl, atan2, atan2f et atan2l,
- Cosinus hyperbolique : cosh, coshf et coshl,
- Sinus hyperbolique : sinh, sinhf et sinhl,
- Tangente hyperbolique : tanh, tanhf et tanhl,

La librairie contient également la définition des constantes suivantes :

M_E	e	2.71828182845904523536
M_LOG2E	log2(e)	1.44269504088896340736
M_LOG10E	log10(e)	0.434294481903251827651
M_LN2	ln(2)	0.693147180559945309417
M_LN10	ln(10)	2.30258509299404568402
M_PI	PI	3.14159265358979323846
M_PI_2	pi/2	1.57079632679489661923
M_PI_4	pi/4	0.785398163397448309616
M_1_PI	1/pi	0.318309886183790671538
M_2_PI	2/pi	0.636619772367581343076
M_2_SQRTPI	2/sqrt (pi)	1.12837916709551257390
M_SQRT2	sqrt(2)	1.41421356237309504880
M_SQRT1_2	1/sqrt(2)	0.707106781186547524401

Les prototypes de fonctions sont les suivants :

```
double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);
double pow(double x, double y);
float powf(float x, float y);
long double powl(long double x, long double y);
double fmod(double x, double y);
float fmodf(float x, float y);
long double fmodl(long double x, long double y);
double floor(double x);
float floorf(float x);
long double floorl(long double x);
double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);
double ceil(double x);
float ceilf(float x);
long double ceill(long double x);
double exp(double x);
float expf(float x);
```

```
long double expl(long double x);
double log(double x);
float logf(float x);
long double logl(long double x);
double log10(double x);
float log10f(float x);
long double log10l(long double x);
double cos(double x);
float cosf(float x);
long double cosl(long double x);
double sin(double x);
float sinf(float x);
long double sinl(long double x);
double tan(double x);
float tanf(float x);
long double tanl(long double x);
double acos(double x);
float acosf(float x);
long double acosl(long double x);
double asin(double x);
float asinf(float x);
long double asinl(long double x);
```

```
double atan(double x);
float atanf(float x);
long double atanl( long double x);
double atan2(double y, double x);
float atan2f(float y, float x);
long double atan2l(long double y, long double x);
double cosh(double x);
float coshf(float x);
long double coshl(long double x);
double sinh(double x);
float sinhf(float x);
long double sinhl(long double x);
double tanh(double x);
float tanhf(float x);
long double tanhl(long double x);
```