Programmation en C

Introduction

Le C est un langage de programmation impératif, c'est à dire dont le code ressemble au travail effectué par le processeur dans une architecture de Von Newman (1947). C'est un langage typé, c'est-à-dire dont le compilateur effectue une vérification du type à la compilation.

Compilation avec GCC

GCC est le compilateur C du projet GNU sous UNIX. Un compilateur traduit un code source, indépendant de la machine, en un fichier exécutable (en langage machine).

Compilation séparée : Pour compiler un programme composé de plusieurs fichiers C, il faut premièrement construire tous les fichiers objets. Par l'exécution de la commande : *gcc –c <fichier.c>*, on crée le fichier objet *file.o.*. Puis, on procède à *l'édition de lien*, par la commande : *gcc -o <fichier.exe> {<fichier.o>} –l <librairies>*, qui construit le fichier exécutable *fichier.exe.* On utilise l'option *-l m* pour charger la librairie mathématique.
Dans le cas, où le programme se composerait d'un unique fichier C, on peut le compiler directement en exécutant la commande : *gcc –o <fichier.c>*.

Makefile

Le fichier makefile permet d'automatiser la compilation, et encore d'autres tâches, grâce à la commande

make -k [<nom cible>]

On donne la syntaxe du fichier *makefile* :

```
<nom cible>: {<nom dépendance>}
        <nom commande>
```

Il faut respecter une tabulation avant la commande. La création du fichier cible dépend de plusieurs fichiers. L'automate commence par tester l'existence du fichier cible, puis effectue une comparaison des dates entre cible et dépendance.

On définit généralement les cibles standards suivantes :
- *clean* : supprimer les fichiers objets créés par le *makefile* hormis l'exécutable.
- *cleanall* ou *realclean* : supprime tous les fichiers crées par le *makefile*.
- *install* : compile et copie les fichiers à exporter (exécutable, .h, bibliothèques).
- *uninstall*
- *tags :* mise à jours des étiquettes.
- *all :* compile le programme

On donne ci-dessous un exemple de fichier *makefile*.

```
CC= gcc
CFLAGS= -g -Wall -W

main.exe: main.o module1.o module2.o
        $(CC) $(CFLAGS)  -o main.exe main.o module1.o module2.o -lm

main.o: main.c
        $(CC) $(CFLAGS) -c  main.c

module1.o: module1.c module1.h
```

```
	$(CC) $(CFLAGS) -c   generation_graphe_pondere.c

module2.o: module2.c module2.h
	$(CC) $(CFLAGS) -c   generation_graphe.c

all: main.exe

clean:
	rm *.o

cleanall: clean
	rm main.exe
```

On notera l'utilisation de macro-définitions *$(MACRO)* que l'on définit simplement en écrivant *MACRO=* … Par ailleurs, on peut employer utilement les abréviations : *$@* qui inique le nom de la cible courante, et *$** qui donne le nom de la cible courante privée de son suffixe. Ainsi on peut simplifier l'écriture du *makefile* :

```
OBJ= main.o module1.o module2.o

main.exe: main.o module1.o module2.o
	$(CC) $(CFLAGS)  $(OBJ) -lm -o $@

main.o: main.c
	$(CC) $(CFLAGS) -c  $*.c
```

Options de compilation

- affichage des warnings, et informations pour debbugage avec *ddd* : *-Wall -w -g*
- pour ne pas prendre en compte les assertions : -dndebug

Creation d'une librairie

Pour créer une librairie ou bibliothèque en C, on utilise les commandes :

```
ar  ruv  library.a  module1.o module2.o  …
```

On ajoute un index des fichiers objets dans la librairie pour faciliter l'édition de lien avec la commande :

```
ranlib  library.a
```

Pour l'édition de lien, il faut par la suite utiliser les options '-l*library'* pour utiliser cette bibliothèque et '-L.' pour préciser sa localisation (par exemple, le répertoire courant).

Les structures de contrôle

On appelle *instruction*, une instruction simple se terminant par un point-virgule, ou un bloc d'instruction {…} regroupant plusieurs instructions.

Les tests

- ***if** (expression) instruction;*

  Si *expression* est vraie, alors *instruction* est exécuté, sinon *instruction* est ignoré.

- ***if** (expression) instruction1*
  ***else** instruction2*

  Si *expression* est vraie, alors *instruction1* est exécuté, sinon *instruction2* est exécuté.

- ***if** (expression) instruction1*
  ***else if** (expression2) instruction2*
  ***else if** (expression3) instruction3*
  …
  ***else if** (expressionN) instructionN*
  ***else** instructionN+1*

  Une et une seule des N instructions sera exécutée. L'instruction qui est exécutée correspond à la première expression vraie. Dans le cas, où il n'y a aucune expression vraie et si l'instruction *else* est présente, c'est *instructionN+1* qui est exécutée. Cette dernière instruction est facultative.

Les boucles

- ***while** (expression)*
  instruction

  instruction est exécuté tant que *expression* est vraie ($\neq 0$).

- ***do***
  instruction
  ***while** (expression);*

  On exécute pour commencer *instruction*, puis si *expression* est vraie on exécute *instruction* à nouveau, sinon on quitte de la structure, et on passe à l'instruction suivante.

- ***for** (expression1; expression2; expression3)*
  instruction

  On exécute *expression1* (initialisation de la boucle). Si *expression2* (test) est vraie, alors on exécute *instruction*, puis *expression3* (incrémentation, décrémentation), jusqu'à ce que *expression2* deviennent faux.

L'instruction switch[1]

```
switch (expression)
        {
        case expression-constante1 :
                instructions
                break;
        case expression-constante2 :
                instructions
                break;
        …
```

[1] interrupteur en anglais

```
case expression-constanteN :
        instructions
        break;
default :
        instructions
        break ;
}
```

expression est évalué et permet de faire le branchement sur l'une des constantes (entiers ou caractères, énumérations) de *case* correspondant à la valeur d'*expression*. Notons que *expression-constante* ne peut en aucun cas être une variable. Les constantes à l'intérieur d'un *switch* doivent être toutes de valeurs distinctes. Les *break* sont à priori facultatifs, mais il est préférable de toujours les garder, pour forcer la sortie du *switch*, et prévenir un fonctionnement hasardeux. L'étiquette *default* est facultative et l'instruction correspondante à celle-ci est exécutée lorsqu'aucun branchement n'a été effectué. On peut rattacher plusieurs cas à une même instruction de la façon suivante :

```
case expression-constante1 : case expression-constante2 : case expression-constante3 :
        instructions
        break ;
```

Rupture de séquence

- ***break****;*

  break exécute une sortie d'un bloc d'instructions dépendant de l'une des instructions suivantes *do, for, while, ou switch.*

- ***continue***;

  Instruction forçant le passage à l'itération suivante de la boucle la plus proche.

- ***goto*** *label;*
  label : instruction

  L'instruction *goto* permet de se brancher inconditionnellement à *instruction*. A éviter absolument.

- ***return*** *[expression]*;

  Exécute une sortie d'une fonction en rendant le contrôle à la fonction appelante, tout en retournant la valeur d'un expression si la fonction appelée l'autorise (≠ *void*).

Les données mémoires

Les types élémentaires

A toutes données doit correspondre un type, qui autorise la réservation d'une zone mémoire de taille prédéfinie, la vérification par le compilateur de la validités des opérations, et permet éventuellement la création de nouveaux types.

Parmi les types élémentaires, on distingues les types entiers et réels. Les constantes relatifs à ces types sont référencés dans le bibliothèque *limits.h et float.h*.

- *entiers*

short int[2]	2 octets	-32 768 … 32 767
unsigned short int	2 octets	0 … 65 535
long int	4 octets	-2 147 483 647 … 2 147 483 648
unsigned long int	4 octets	0 … 4 294 967 295

On utilise les abréviations *short* et *long* pour désigner respectivement un *short int* et un *long int. L*e type *int* désigne selon les compilateur soit un *short* soit un *long*. On peut également spécifier explicitement qu'un type est signé en ajoutant *signed*.

- *réels*

float	*4 octets*	*simple précision, 1.18 E +/- 38*
double	*8 octets*	*double précision, 9.46 E -308 … 1.8 E 308*
long double	*10 octets*	

- *caractères*

Un caractère est représenté en machine par un entier, ainsi *'c'* désigne la valeur entière du caractère dans la table *ASCII*.

char	1 octet	-128 … 127
unsigned char	1 octet	0 … 255

- *booléen*

Souvent on définit les macro-constantes *FALSE* et *TRUE* comme suit :

enum bool {FALSE, TRUE } ;

En C, une expression conditionnelle est fausse, quand elle prend la valeur (entière) 0, et vraie dans tous les autres cas.

Les tableaux ou vecteurs

Un tableau est un type structuré homogène[3]. On déclare le tableau *nom* comportant *max* éléments de type *type_élement* en écrivant :

*type_élement nom **[**max**]**.*

max doit être une constante littérale, comme *10* ou défini par *#define max 10*, ce qui revient au même. A priori, la définition de *max* comme *const max = 10* génère une erreur de compilation.

La déclaration du tableau réserve une zone mémoire séquentielle de taille *max * sizeof(type_élément)* débutant à l'adresse *nom*. Pour accéder aux éléments contenus dans le tableau, on écrit *nom[i]* ou *i* est un entier compris

[2] *int* est facultatif dans ces déclarations.
[3] toutes les données sont d'un même type.

entre 0 et *max - 1*. *i* représente un décalage en mémoire par rapport à l'adresse *nom* ; ainsi l'adresse de *nom[i]* est *nom + i * sizeof(type_élément)*. On donne un exemple d'initialisation :

int vecteur [ ] = {0, 1, 2}.

A l'initialisation, la taille du tableau est facultative, le compilateur évalue la taille mémoire de la donnée avant de procéder à la réservation mémoire. Une fois déclaré, l'adresse mémoire du tableau n'est pas modifiable. Ainsi toute opération du type *tableau = adresse* est illicite, on dit que le tableau est une *r-value*[4].

On peut augmenter la dimension du tableau en déclarant des tableaux de tableau, *type_élement nom[max1][max2]*. Par exemple, les deux initialisions suivantes sont équivalentes :

float vecteur [3] [2] = {{0.0, 0.1}, {1.0, 1.1}, {2.0, 2.1}} ;
float vecteur [3] [2] = {0.0, 0.1, 1.0, 1.1, 2.0, 2.1} ;

Les chaînes de caractères sont des tableaux particuliers, des tableaux de caractères (*char)*, dont l'élément *'\0'* code la fin de la chaîne, mais pas nécessairement du tableau ! Considérons par exemple, la déclaration :

char chaine [20] = {'c', 'o', 'u', 'c', 'o', 'u', '\0'} ;

ou de façon raccourci :

char chaine [20] = "coucou" ;

On peut accéder aux caractères de la chaîne directement en écrivant par exemple *chaine[2]* qui vaut *'u'*. La bibliothèque standard *string.h* fournit un certain nombre d'opérations élémentaires sur les chaînes : copie, concaténation, longueur, comparaison…

Les pointeurs et les tableaux

Un pointeur est une variable qui contient l'adresse d'une autre variable. Le langage C fait un grand usage des pointeurs. Les pointeurs génériques sont déclarés de type *void **.

Déclarons un pointeur *p* sur *char* et un caractère *c* :

*char c , * p ;*

*char * p* peut se lire de deux façon : * *p* est un caractère (*char*), ou bien *p* est un pointeur sur caractère (*char **).

L'opérateur unaire & donne l'adresse d'un objet ; l'instruction

p = &c ;

affecte l'adresse de *c* à la variable *p*, on dit que *p* pointe sur *c*. L'opérateur & s'applique uniquement aux objets en mémoire : les variables, les éléments de tableaux, etc. Il ne peut pas s'appliquer à des expressions, des constantes, ou des variables de type *register*.

L'opérateur unaire * représente l'opérateur *d'indirection* ou de *déréférence* ; quand on l'applique à un pointeur, il donne accès à l'objet pointé par ce pointeur. Par exemple, on peut modifier le contenu de *c* en écrivant

** p = 'a' ;*

Les tableaux et les pointeurs sont fortement liés ; toute opération que l'on peut effectuer par indexation dans un tableau peut être réalisé à l'aide de pointeurs. La version utilisant les pointeurs sera en général plus rapide, mais un peu plus difficile à comprendre, du moins pour les non-initiés.
Considérons un tableau *a* de 10 entiers et un pointeur *pa* sur entier, déclarés ainsi :

int a[10] , * pa ;

[4] *right value, à droite du signe égal dans les affectations*

alors l'affectation

pa = & a [0] ;

fait pointer *pa* sur l'élément zéro de *a*, c'est-à-dire que *pa* contient l'adresse de a et ainsi * *pa* va désigner le contenu de *a [0]*. De ce fait, si *pa* pointe sur un certain élément du tableau, *pa + 1* pointe sur l'élément suivant, *pa + i* pointe sur le $i^{ième}$ élément après *pa* et *pa - i* sur le $i^{ième}$ élément avant *pa*. Par conséquent, si *pa* pointe sur *a [0]*, alors

** (pa + i)*

représente le contenu de *a [i]*. Aussi on peut utiliser l'expression équivalente *pa[i]* pour désigner **(pa + i)*. Ces remarques sont vraies quelque soit le type et la taille des variables du tableau *a*. Le sens « ajouter 1 à un pointeur » est que *pa + 1* pointe sur l'objet suivant (*arithmétique des pointeurs*).

La correspondance entre l'indexation d'un tableau et les calculs sur les pointeurs est très étroites. **Par définition, la valeur d'une variable de type tableau est l'adresse de l'élément zéro du tableau.** Par conséquent, après l'affectation *pa = & a [0]*, *pa* et *a* ont même valeur ; on peut également écrire :

pa = a ;

D'un autre côté, une référence à *a [i]* peut aussi bien s'écrire ** (a + i)*. En évaluant la première expression, le C la convertit immédiatement sous la seconde forme. De même, les expressions *& a [i]* et *a + i* sont équivalentes. En résumé, un expression comportant un tableau et un indice est équivalente à une autre écrite avec un pointeur et un déplacement.

Cependant il faut garder à l'esprit qu'il existe des différences entre un nom de tableau et un pointeur. Un pointeur est une variable et donc les instructions *pa = a* et *pa++* sont licites. En revanche, **un nom de tableau n'est pas une variable**, ce qui implique des instructions telle que *a = pa* ou *a++* sont incorrectes.

Quand on passe un nom de tableau en argument à une fonction, c'est l'adresse de son élément initiale qui est transmise. A l'intérieur de la fonction appelée, cet argument est une variable locale, et par conséquent, un nom de tableau en paramètre est un pointeur.

En tant que paramètre formels dans une déclaration / définition de fonction

void f (char s [ ])

et

*void f (char * s)*

sont équivalents ; cependant la dernière est plus explicite, parce qu'elle indique que le paramètre est un pointeur. Puisque *s* est un pointeur, on a parfaitement le droit de l'incrémenter ; l'instruction *s++* n'a aucun effet sur l'adresse de la chaîne de caractère de la fonction qui appelle *f*, car elle ne travaille que sur une copie du pointeur locale à *f*. On peut donc réaliser les trois appels suivants en déclarant *char tab [100]* et *char * ptr :*

```
f ( "bonjour, maître" ) ;
f ( tab ) ;
f (ptr ) ;
```

La gestion dynamique de la mémoire, s'effectue au moyen des fonction de la bibliothèque standard *stdlib.h*. On allouera de la mémoire par le biais des fonctions *malloc* et *realloc*, et on la libérera avec *free ;* on dispose de la macro-constante *NULL* indiquant qu'un pointeur ne pointe sur rien.

Pointeurs de fonctions : …

Les structures ou enregistrements

La structure est un type structuré hétérogène[5]. On définit la structure *nom_structure* possédant N champs de type *type_i* :

```
struct nom_structure
        {
        type_1 champ_1 ;
        …
        type_N champ_N ;
        } ;
```

Par la suite, on déclare une variable *article* de type *struct nom_ structure* en écrivant

```
struct nom_structure article.
```

On accède aux données du *champs_i* de la variable *article* en écrivant

```
article.champs_i.
```

Voici un exemple de définition d'une structure *point*,

```
struct point
        {
         float x  ;
        float y ;
        } ;
```

Par ailleurs, les structures peuvent être *autoréférentielle*, comme par exemple :

```
 struct nœud
        {
        char *  mot ;
        struct nœud * gauche ;
        struct nœud * droit ;
        } ;
```

On peut initialiser une variable de type structure, en écrivant :

```
struct point  origine = {0.0, 0.0} ;
```

Pour initialiser correctement une structure, il faut tenir compte de l'ordre des champs lors de la définition.
Les seules opérations autorisées sur une structure sont la copie ou l'affectation en la considérant comme un tout, la récupération de son adresse au moyen de l'opérateur &, et l'accès à ces membres. La copie et l'affectation des structures permettent de les passer en arguments, et également de les utiliser comme valeur de retour de fonctions. Si l'on doit passer une structure de taille importante en argument à une fonction, il est généralement plus efficace de lui passer un pointeur plutôt que de copier la structure entière. La déclaration

```
struct point origine, * pp ;
```

signifie que *pp* est une pointeur sur une structure de type *struct point*, tandis que *origine* est une structure simple. Par la suite, on peut écrire :

```
pp = &origine ;
```

On accède au champs de la structure, en écrivant

```
(*pp).x ;
```

[5] les champs possèdent des types identiques ou différents.

les parenthèses sont nécessaires parce que la priorité de l'opérateur . est supérieur à celle de *. Aussi emploie-t-on une notation abrégée. Si p est un pointeur sur une structure, alors on accède directement au champ par

p→membre-de-structure

Le type union

Le type de données *union* sert un peu de fourre-tout. Une union est une zone mémoire définit par le programmateur de façon suffisamment large pour pouvoir contenir tout type de donnée qui viendrait à s'y présenter. Tous les membres ont un décalage nul par rapport à la base La syntaxe des *union* se fonde sur celle des structures :

union *nom-union*
{
type_1 membre_1 ;
…
type_N membre_N ;
} ;

On déclare la variable *var-union* par

union nom-union var-union.

La zone mémoire réservée pour var-*union* correspond à la taille du plus grand des types contenu dans l'union. Syntaxiquement, on accède aux membres d'une union par var-*union.membre_i* L'union ne peut être initialisé que par une constante dont le type est celui de son premier membre.

Les champs de bits

Un champ de bits est un ensemble de bits adjacents définis à l'intérieur d'une structure.

struct *nom {*
unsigned int mot_1 : taille_1 ;
…
unsigned int mot_N : taille_N ;
} ;

Comme un *mot* est toujours moins long qu'un octet (0 < *taille* < 8), il ne possède pas d'adresse mémoire, on ne peut donc pas lui faire référence avec &. Si on utilise la *taille* spéciale 0, on force l'alignement sur le début du mot suivant. Les champs ne peuvent être déclaré que comme des *signed* ou *unsigned int.* Les champs sont affectés de gauche à droite sur certaines machines et de droite à gauche sur d'autre. Les programmes qui dépendent de telles propriétés ne sont pas *portables*.

Les énumérations

Les énumérations constituent un moyen pratique d'associer des noms à des valeurs *constantes entières ou caractère,* comme avec *#define,* mais avec l'avantage de la génération automatique des valeurs.

enum *nom {constantet_1, …, constante_N} ;*

Le premier nom *constantet_1* vaut 0, le suivant 1, et ainsi de suite, sauf si l'on précise des valeurs explicites.

enum *nom {constantet_1 = C1, …, constante_N = CN}*

Si l'on ne donne que certaines valeurs, les autres se déduisent par incrément successifs, comme par exemple :

enum mois { JAN = 1, FEV, MAR, AVR, MAI, JUN, JUL, AOU, SEP, OCT, NOV, DEC } ;

Bien que l'on puisse déclarer des variables de types *enum,* les compilateurs ne sont pas obligés de vérifier que la valeur de la variable appartient bien à l'énumération. Cependant, il vaut mieux employer des énumérations que des directives *#define,* car elles ont une chance d'être vérifiées.

L'initialisation des variables

L'initialisation consiste à spécifier la valeur d'une variable au moment de la création de sa zone mémoire. En l'absence d'initialisation explicite, les variables externes et statiques sont toujours initialisées à zéro ; les variables automatiques et les *variables-regitres* ont des valeurs initiales indéfinies. On peut initialiser les variables scalaires simplement en écrivant :

```
int x = 1 ;
char c = 'a' ;
long  j = 1000L * 100L ;
```

Pour les variables externes et statiques, l'initialisateur doit être une expression constante ; l'initialisation s'effectue une seule fois comme si elle avait lieu avant le début de l'exécution du programme. Dans le cas des variables automatiques et des variables-registres, elle s'effectue chaque fois que l'on entre dans la fonction ou dans le bloc correspondant. Pour les variables automatiques et les variables-registres, l'initialisateur n'est pas forcément une constante : il peut être constitué d'une expression quelconque se servant de valeurs définies précédemment, ou même d'appels de fonctions. En effet l'initialisation des variables automatiques sont simplement une manière d'abréger les instruction d'affectation. **De façon générale, l'initialisation des variables globales se fait « à la compilation », celle des variables locales se fait à l'exécution.**

Pour l'initialisation des tableaux et des structures, voire les rubriques concernées.

Les fonctions

La définition d'un fonction est de la forme :

```
type-de-retour nom-de-fonction ( déclarations d'arguments )
{
        déclarations et instructions
}
```

Généralement, la définition de la fonction joue aussi le rôle de la déclaration ; mais on peut très bien séparer la déclaration. Par exemple, si l'on souhaite réaliser un module exportable, ou encore si l'on souhaite utiliser une fonction défini plus bas :

```
type-de-retour nom-de-fonction ( déclarations d'arguments ) ;
```

Certaines parties de ces définitions sont facultatives ; la fonction la plus succincte possible est

```
dummy ( ) { }
```

qui ne fait rien et ne retourne rien. Une telle fonction sert parfois à remplacer une fonction qui n'est pas écrite pendant le développement d'un programme. Si l'on ne précise pas le type de retour, c'est le type *int* qui est pris par défaut. Les fonctions communiquent par le mécanisme des arguments et des valeurs de retour, et encore via le mécanisme des variables externes.
L'instruction *return* est le mécanisme par lequel une fonction appelée retourne une valeur à la fonction appelante.

```
return expression ;
```

On met souvent l'expression entre parenthèse, mais c'est facultatif. On donne le type de retour *void* pour spécifier que la fonction ne retourne rien.

Le C ne permet pas de définir une fonction à l'intérieur d'une autre fonction.

Comme **le langage C passe tous les arguments des fonctions par valeurs** (*passage par valeur*), la fonction appelée n'a aucun moyen de modifier les arguments du programme appelant ; elle ne traite que des copies, aussi la fonction *echanger (a ,b)*, ci-dessous, ne réalise pas l'échange des entiers *a* et *b*.

```
void echanger ( int x, int y)
```

```
/* erreur */
{
        int temp ;
        temp = x ;
        x = y ;
        y = temp ;
}
```

Le moyen d'obtenir le résultat voulu est de faire en sorte que le programme appelant passe en arguments des pointeurs sur les valeurs à modifier (*passage par adresse*) : *echanger (&a , &b).* On propose la solution

```
void echanger ( int * px, int * py)
{
        int temp ;
        temp = * px ;
        * px = * py ;
        * py = temp ;
}
```

Les arguments de type pointeur permettent à une fonction d'accéder aux objets de la fonction appelante, et de les modifier.

Les classes de stockage

Un *identificateur*[6] est caractérisé par :

- *la durée de vie*, *temporaire* ou *permanente* (durant toute la durée du programme).
- *la portée*, *locale* ou *globale*, qui correspond à la zone de validité d'un identificateur dans un source, a-t-on accès à la zone mémoire de l'identificateur ?
- *la visibilité*, qui précise si un identificateur peut être utilisé à un endroit du source et à quelle zone mémoire on se réfère dans le cas d'ambiguïté.

Considérons l'exemple suivant,

```
{
int i ;
…
        {
        int i ;
        i = 1 ;
        }
…
}
```

Pour connaître à quelle zone mémoire on fait référence, on se rapporte la déclaration la plus proche dans le bloc. Ainsi une déclaration peut en cacher une autre (restriction de visibilité) !

Lorsqu'on déclare un identificateur *dans un bloc (identificateur interne)*, la zone mémoire est réservée dynamiquement sur *la pile* et sa durée de vie est temporaire. La portée s'étend de la *déclaration* jusqu'à la fin du bloc. L'initialisation est recommencée à chaque passage dans le bloc.
Pour un identificateur déclaré *hors-bloc*, la zone mémoire est réservée « à la compilation » dans *le segment de données*, sa durée de vie est permanente et sa portée globale, c'est-à-dire étendue tout le programme (les modules y compris) même si elle n'est pas immédiatement visible. L'initialisation de telle variable s'effectue une seule fois « à la compilation ».

On utilise des *modificateur*s de type pour modifier la *durée de vie*, la *portée* et la *visibilité* d'un *identificateur*,

```
modificateur type identificateur ;
```

[6] à priori une variable ou une fonction

Les *modificateurs* de type standards sont *extern, static, auto, register, const* et *volatile*. On ne peut faire figurer qu'un seul spécificateur de classe de stockage par déclaration. Si l'on en donne pas, les règles suivantes s'appliquent : les identificateurs déclarés à l'intérieur d'une fonction sont de classe *auto ;* les fonctions sont par défaut *externe* (défini hors de tous bloc, par opposition à *interne*[7]), car le C ne permet pas de définir des fonction qui serait à l'intérieur d'une autre fonction. Par défaut les variables et les fonctions externes ont la propriété suivante : chaque fois que leur nom apparaît, même dans des fonctions compilés séparément, il désigne la même chose (*external linkage*). Les variables externes sont accessibles partout ; par la suite, on verra comment définir des variables externes (classe *extern)* et des fonctions qui ne soient visibles que à l'intérieur d'un seul fichier source (cf. classe *static*). Les variables externes sont permanentes.

- *extern*

Ce modificateur indique que l'identificateur a déjà une zone mémoire allouée. *extern* s'applique à un identificateur globale et doit être redéclaré dans chaque module pour qu'il soit visible. A la compilation, deux zone mémoire sont réservées puis *à l'édition de lien,* le compilateur en supprime une. Pour éviter tout problème de double initialisation, on observera la règle suivante **: pas d'initialisation sur un identificateur *extern***. Il vaut mieux faire :

```
/* fichier f1.c */
float x = 0.0 ;

/* fichier f2.c */
extern float x ;
```

- *static*

static possède deux sens différents suivant qu'il fait référence à un identificateur dans un bloc ou hors-bloc. Dans le premier cas, *static* change la durée de vie d'un identificateur temporaire, qui devient permanent, mais sa portée reste locale. L'initialisation est réalisée une seule fois à la compilation.

Dans l'exemple suivant *compteur* décompte le nombre d'utilisation de la fonction *feufeu*:

```
void feufeu(void)
{
static compteur = 0 ;
compteur++ ;
…
}
```

Dans le second cas, où la déclaration se situe hors-bloc, *static* modifie la portée qui est restreinte au module, et non pas étendue à tout le programme.

- *auto*

Cette classe de stockage ne peut se déclarer qu'à l'intérieur du bloc qui l'utilise et elle garantit que la variable est détruite en quittant le bloc. La plupart des variables sont *auto* de manière implicite.

- *register*

register est une classe qui donne priorité à une variable pour le stockage dans les registres lors de l'exécution. Les registres du microprocesseur représentent ce qui se fait de mieux pour la rapidité d'accès aux données ; aussi cette classe de variables permet-elle d'en accélérer les transfert. On ne doit pas l'utiliser de manière excessive car le nombre de registres disponibles est limité. De plus, elle ne s'applique qu'à certains types. Le mot *register* n'est pas pris en compte lorsque la déclaration correspondante et interdite

```
register int x ;
register char c ;
```

On ne peut appliquer les déclarations de classe *register* qu'aux variables internes aux blocs (*automatiques*) et aux paramètres formels d'une fonction.

```
f(register unsigned m, register long n)
```

[7] les variables automatiques sont internes

```
{
        register int i ;
        …
}
```

Par ailleurs, on ne peut pas récupérer l'adresse d'une variable de classe *register*, même si celle-ci n'est pas effectivement placé dans un registre du microprocesseur.

- *const*

const est un qualificatif de type, au même titre que *volatile*. On peut initialiser un identificateur *const*, mais on ne peut rien lui affecter par la suite (*r-value*). La variable se comporte dès lors comme une constante typée. La variable est placée en mémoire morte, et peut disposer de certaine optimisation.

La définition de nouveaux types

On peut définir de nouveaux types simplement au moyen de l'instruction *typedef*, en écrivant :

***typedef** ancien_type nouveau_type ;*

Voici un exemple classique :

*typedef char * chaine* ;

Pour les structures, on peut omettre de préciser le nom de la structure, comme par exemple :

```
typedef struct
        {
        float module ;
        float argument ;
        } complexe ;
```

ou choisir abusivement de lui donner le même nom qu'au type défini :

```
typedef struct complexe
        {
        float module ;
        float argument ;
        } complexe ;
```

Activation d'une routine

A une routine correspond une adresse mémoire.

L'unité de contrôle doit permettre :
- de passer à l'instruction suivante, par incrémentation du compteur ordinal
- de sauter à une adresse (jump)
- au besoin de stocker l'adresse de l'instruction courante (compteur ordinal) dans le segment de pile → *adresse de retour*
- de conserver le sommet de la pile

L'activation d'une routine génère la création d'*un cadre d'appel*. On utilise le mécanisme de *la pile d'appel* pour sauvegarder les informations concernant la routine activée, ces informations forment le *cadre d'appel* (*frame*) de la routine.

Tableau 1 : pile d'appel

variables locales
paramètres
variable de retour
adresse de retour
cadre d'appel précédent

Quand on sort de la routine, on supprime le cadre d'appel, et on reprend l'exécution du programme au niveau de l'adresse de retour, en passant cette adresse au compteur ordinal.

Les expressions

Une expression exprime un calcul, et possède une valeur typée. Les éléments de base de construction d'une expression sont d'une part les données (*les constantes littérales, les variables*) et d'autre part les traitements (*les fonctions, les opérateurs*). On va traiter dans cette partie les opérateurs

Les opérateurs

Les opérateurs sont des traitements prédéfinis, propres au langage, à la différence des fonctions qui sont des traitements définis par le programmeur. On distingue, selon la place de l'opérateur par rapport aux opérandes, les opérateurs *préfixes op d1 d2*, *infixes d1 op d2*, et *postfixes d1 d2 op*. *L'arité* est le nombre d'opérande nécessaire à un opérateur.

Pour évaluer les expressions sans ambiguités, on définit un *ordre de priorité* entre les opérateurs. Les parenthèses possède l'ordre de priorité maximum. *L'associativité* définit le sens du traitement gauche-doite, ou droite-gauche, des expressions constituées d'opérateurs de même priorité. *La surcharge des opérateurs* permet de donner le même nom d'opérateur avec des types de paramètres différents, par exemple la multiplication d'entièrs et de réels. Des conversions implicites de types sont générés au besoin. En langage C, il n'y a pas de surchage sur les noms de fonction.

Tableau 2 : Opérateur d'expression rangés par priorité décroissante

Opérateur	Associativité	Arité
( ) [ ] *→ .*	GD	– 2
*! ~ ++ -- - + (type) * & sizeof*	DG	1
** / %*	GD	2
+ -	GD	2
<< >>	GD	2
< <= > >=	GD	2
== !=	GD	2
&	GD	2
^	GD	2
|	GD	2
&&	GD	2
||	GD	2
? :	DG	3
*= += -= *= /= %= >>= <<= &= ^= |=*	DG	2
,	GD	2

En langage C, *l'affectation* = est un opérateur, et on peut l'utiliser dans les expressions. Cependant, il faut prendre garde des effets de bords indésirés. *L'effet de bord* survient lors de l'utilisation simultanée dans une même expression de la *r-value* et de la *l-value*. Le C dispose d'opérateur à effet de bord tels que +=, ++, … **De façon générale, il ne faut pas utiliser deux effets de bord dans une même expression.**

- l'opérateur sizeof : *sizeof (type ou variable)* retourne la taille en octet du type ou de la variable passée en argument. Cette opérateur est très utilisée pour l'allocation dynamique de mémoire.

- *Les opérateurs d'incrémentation et de décrémentation* servent à incrémenter et décrémenter les variables (et non toutes exepressions). L'opérateur d'incrémentation ++ ajoute 1 à son opérande, alors que l'opérateur de décrémentation -- lui retranche 1. L'aspect inhabituel de ces opérateurs est qu'on peut les mettre sous forme préfixée (*++n*) ou postfixée (*n++*). Dans les deux cas, *n* est incrémenté. Mais l'espression *++n* incrémente *n* avant de prendre sa valeur, alors que *n++* l'incrémente après avoir pris sa valeur. Les opérateurs d'incrémentation et de décrémentation ne s'applique qu'à des variables, c'est-à-dire *(i + j)++* est illégale.

- L'expression conditionnelle, qui utilise *l'opérateur conditionnel* ? : s'écrit :

  expression1 ? expression2 : expression3

Cet opérateur est ternaire. On commence par évaluer l'expression *expression1*. Si elle est vrai (c'est-à-dire $\neq 0$), alors on évalue *expression2* et c'est elle qui donne sa valeur à l'expression conditionnelle ;sinon on évalue *expression3* et c'est elle qui donne sa valeur. Ainsi pour affecter le maximum de a et b à *max*, il suffit d'écrire:

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

Il n'est pas obligatoire de mettre des parenthèses autour de la première expression, car la priorité de *? :* est très faible, immédiatement supérieur à celle de l'affectation.

- *Les opérateurs d'affectation*
  On remplace souvent en raccourci, et pour plus de clarté

  i = i + 2 ;

  par l'expression équivalente utilisant l'opérateur d'affectation +=

  i += 2;

  Le C associe à la plupart des opérateurs binaires *op* un opérateur d'affectation *op=*, où *op* fait partie de

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

  Si *expression1* et *expression2* sont des expressions alors

  expression1 op= expression2

  équivaut à

  expression1 = (expression1) op (expression2)

  mis à part que *expression1* n'est calculé qu'une fois. Les parenthèses qui entourent *expression2* ont leur importance. Les opérateur d'affectation, = y compris, on tous même priorité, la plus faible après l'opérateur de séquencement *(,)*.

Les opérateurs de traitement des bits

Le C fournit six opérateurs (classés ci-dessous par ordre de priorité) qui réalisent des manipulations au niveau des bits ; on ne peut les appliquer qu'aux opérandes entiers, c'est-à-dire *char*, *short*, *int* et *long* signés ou non.

~	*complément à un (opérateur unaire)*
>>	*décalage à droite*
<<	*décalage à gauche*
&	*ET bit à bit*
^	*OU exclusif bit à bit*
|	*OU inclusif bit à bit*

On se sert souvent du ET bit à bit & pour masquer certains bits ; par exemple

n = n & 0177 ;

met à zéro les bits de n qui ne font pas partie de ses 7 bits de poids faibles. En effet, *0177* correspond au masque binaire suivant 0…0 001 111 111 ; à chaque chiffre de l'opérande correspond la valeur octale d'une séquence de trois bits du masque.

Le OU bit à bit | sert à mettre des bits à 1 :

x = x | masque ;

met à 1 tous les bits de x qui sont à 1 dans *masque*.

Le OU exclusif ^ met à 0 tous les bits qui sont identiques dans ces deux opérandes, et à 1 ceux qui diffèrent.

Les opérateurs << et >> décale leur opérande de gauche du nombres de bits indiqué par leur opérande de droite, qui doit être positif. Ainsi

x << 2

décale la valeur de x de deux bits vers la gauche, en remplissant à droite par des 0, ce qui revient à une multiplication par 4. Si l'on décale une quantité non signée vers la droite, les bits de gauche sont toujours mis à 0. Si cette quantité est signée, les bits à gauche se rempliront par des bits de signes sur certaines machines et par des 0 sur d'autres.

L'opérateur unaire ~ donne *le complément à un* d'un entier ; c'est à dire qu'il met à 1 les bit qui sont à 0 et vice-versa. Par exemple,

x = x & ~077 ;

met à 0 les six bits de poids faibles de x. Cette instruction est portative, car elle ne dépend pas de la taille de x ; si l'on suppose que x est codé sur 16 bits, elle est équivalente à cette autre :

x = x & 0177700 ;

Ce dernier masque s'écrit ~~0…0 00~~1 111 111 111 000 000, avec les bits rayés pour indiquer qu'ils ne sont pas significatifs. Cependant, ils sont nécessaires à la cohérence du codage octal.

Les conversions de type

Lorsqu'un opérateur a des opérandes de types différents, ils sont convertis en un type commun d'après quelques règles. En général, les seules *conversions automatiques* sont celles qui convertissent un opérande « étroit » en un opérande plus « large », sans qu'il y ait perte d'information. Les expressions qui peuvent conduire à une perte d'informations comme l'affectation d'un *long* à un *int* génère un *warning*[8] mais ne sont pas illégale !

Si aucun des opérandes n'est *unsigned*, les quelques règles suivantes suffisent :

- si l'un des opérandes est *long double*, convertir l'autre en *long double*
- sinon, si l'un des opérandes est *double*, convertir l'autre en *double*
- sinon, si l'un des opérandes est *float*, convertir l'autre en *float*
- sinon, convertir les opérandes de type *char* et *short* en *int*
- Puis, si l'un des opérandes est *long*, convertir l'autre en *long*

Les règles de conversions se compliquent dans le cas d'opérandes *unsigned.* Le problème est que les comparaisons entre des valeurs signées et non signées dépendent de la machine, parce qu'elle dépendent des tailles des divers types entiers.

Des conversions s'effectuent également lors des affectations ; la valeur de la partie droite de l'affectation est convertie dans le type de la partie gauche, qui est le type du résultat. Les caractères sont convertis en entier selon les règles énoncées ci-dessus, avec ou sans extension de signe. Les entiers trop longs sont transformés en des entiers plus courts ou en *char*s en tronquant les bits de poids fort.

Enfin, dans toute expression, on peut forcer explicitement des conversions grâce à un opérateur unaire appelé *cast*. Dans la construction

(nom-de-type) expression

l'expression est convertie dans le type précisé, selon les règles de conversion précédentes. Par exemple, la fonction *sqrt* de la bibliothèque *math.h*, qui calcule la racine carrée de son argument, s'attend à recevoir un *double*. Donc, si *n* est un entier, on peut écrire

[8] avertissement du compilateur

sqrt ((double) n)

pour convertir la valeur *n* en un *double* avant de le passer à *sqrt*. Cependant, si les arguments ont été déclarés dans le prototype de fonction, comme c'est ici le cas,

double sqrt (double) ;

alors cette déclaration force la conversion des arguments dans leurs types respectifs au moment de l'appel de la fonction.

Le préprocesseur

Un *préprocesseur* retraite le code avant la compilation et effectue des substitutions de macros, une compilation conditionnelle, et l'inclusion de fichiers par leurs noms. On peut utiliser l'option de compilation *-e* pour générer les fichiers d'extension *.e* construits par le préprocesseur.

Macro constante (et macro fonction)

- *# **define** identificateur chaîne*
  *L*e préprocesseur remplace *identificateur* par *chaîne* dans le code source.
  _ _LINE_ _ , _ _ DATE_ _, _ _TIME_ _, _ _FILE_ _ sont des macros constantes prédéfinis indiquant la ligne courante dans le code, la date, l'heure, et le nom du fichier courant.

L'inclusion de fichier

- *# **include** <nom-de-fichier>*
  Le préprocesseur recherche *nom-de-ficher* dans les répertoires standards[9], et l'insère dans le code source.
  Pour étendre la recherche au repertoire *rep*, on peut spécifier l'option de compilation *-I rep*.

- *# **include** "nom-de-fichier"*
  Le préprocesseur recherche *nom-de-ficher* dans le répertoire courant, et l'insère dans le code source.

La compilation conditionnelle

On propose la syntaxe des instructions conditionnelles propres au préprocesseur.

- *# **if** expression-constante* ou *# **ifdef** indentificateur* ou *# **ifndef** identificateur*
  *# **ifdef** identificateur* équivaut à *# **if defined** (identificateur)* ; de même # **ifndef** équivaut à *# **if** !**defined** (identificateur).*
- *texte*
- *# **elif** expression-constante*
- *texte*
- *# **else***
- *texte*
- *# **endif***

Création d'un module

Le module se compose de deux fichiers *module.h*, qui contient la déclaration des variables et des fonctions à exporter et *module.c*, qui contient leurs définitions.

```
/* fichier module.h */

# ifndef MODULE_H

# define MODULE_H

/* déclarations publiques + commentaires */
extern …

# endif
```

```
/* fichier module.c */

# include "module.h"

/* définitions privées */
…
```

[9] répertoire /usr/include/

```
/* définitions publiques */
…
```

La compilation conditionnelle permet d'éviter une redondance du code lors des multiples appels *# include "module.h* . Par ailleurs, l'inclusion de *fichier.h* dans *fichier.c* (qui n'est à priori pas nécessaire) permet au préprocesseur de vérifier la cohérence entre les déclarations et les définitions.

Les fonctions C standards

La bibliothèque standard *stdio.h*

La bibliothèque standard *stdlib.h*

La bibliothèque standard *string.h*

La bibliothèque standard *ctype.h*

La bibliothèque standard *limits.h*

La bibliothèque standard *float.h*

La bibliothèque standard *math.h*