Programmation C++
Mathématiques, Master 2
Université d'Orléans

Thomas Haberkorn

2015

Table des matières

Préambule

Le langage C++ est abondamment utilisé dans le monde de l'industrie et de la recherche. C'est un langage permettant de développer rapidement des applications complexes tout en assurant une certaine robustesse et flexibilité.

Le C++ est avant tout une surcharge du langage C auquel on a rajouté les fonctionnalités de la programmation orienté objet. Ce document commence donc par une introduction au C qui est un langage purement fonctionnel (par opposition aux langages objets). Puis il continue par une introduction au C++ et à ses fonctionnalités objets.

Ce document n'est qu'une introduction et n'a donc pas la prétention de rentrer dans les subtilités de la programmation C++. De plus, comme pour tout langage de programmation, la seule façon de maîtriser le C++ est de l'utiliser.

Chapitre 1

Introduction au Langage C

1.1 Petit historique

Le langage *C* a été écrit par Dennis Ritchie, des laboratoires Bell AT&T, vers 1972, afin de porter le système d'exploitation *UNIX* sur une nouvelle machine : le DEC PDP-11. Ce langage avait au début pour ambition d'être simplement une extension du langage *B* (créée par Ken Thompson vers 1969, encore une fois des laboratoires Bell AT&T). Cette extension est plus portable que son ancêtre car moins liée à l'architecture matérielle du système sur lequel elle est utilisée ; elle permet de traiter efficacement des caractères de 8 mots (contrairement au langage *B*). Très vite, le *C* s'est vu ajouter des fonctionnalités/notions telles que les tableaux, les pointeurs, les nombres à virgules flottantes, les structures... En 1974, des licences *UNIX* ont été accordées aux universités ce qui a grandement popularisé l'utilisation du langage *C*.

Contrairement à ce qu'on peut penser, le *C* est un langage très simple à apprendre (attention, pas à maîtriser), il est très flexible et puissant. Depuis sa création, il a peu évolué, contrairement à son extension, le *C++*. La version communément utilisée du *C* est celle de la norme *ANSI* (1989), bien que la plupart des compilateurs libres respectent une norme plus ancienne. Un bon compilateur libre de *C* est *gcc*, qui lui, respecte la norme *ANSI*. En principe, tout programme écrit en *ANSI C* est valide en *C++*, à quelques exceptions syntaxiques près.

1.2 Créer un Programme en Langage C

Qu'est-ce qu'un Programme ?

Pour emprunter une comparaison glanée sur internet, un programme est comme une recette de cuisine. Il comporte des ingrédients (les variables), une démarche à suivre pour manipuler/combiner ces ingrédients (les instructions) et, pour un plat chaud, une étape finale qui est la cuisson (compilation). Plus encore que pour une recette de cuisine, l'écriture d'un programme doit suivre un vocabulaire/langage fixé dont il ne faut pas dévier sous peine que le programme, qui doit être lu par un ordinateur, ne soit pas compris.

Un programme est donc une suite ordonnée de déclarations de variables (préparation des ingrédients) et de manipulations sur celles-ci.

Il est important de retenir qu'un ordinateur, par l'intermédiaire d'un programme, ne fera que ce que le programmateur lui aura dit de faire, pas forcément ce que le programmateur voudrait que le programme fasse. D'où le classique : **#?!@&* d'ordinateur qui ne fait pas ce que je veux, juste ce que je lui dit !**. Conclusion : si un programme ne marche pas, ce n'est, la plupart du temps, pas la faute du langage ou de l'ordinateur, c'est celle du programmateur.

Le Fichier Source

La recette de cuisine qu'est un programme, s'écrit tout simplement dans un (ou des) fichier(s) texte(s) sous leur plus simple expression. Ce fichier est appelé fichier source et ne doit comporter que des caractères ASCII, ie. pas de caractères spéciaux, pas d'*italiques*, pas de **gras**, pas de soulignés... Il existe cependant des éditeurs de textes aidant l'écriture de programme *C* en mettant en page les instructions, déclarations, fonctions.

Pour être valide, le fichier source doit posséder l'extension *.c* et comporter un programme *main ()*. Un exemple de programme contenu dans un fichier source est le suivant :

Exemple 1.2.1 (Programme Aloha Monde)**.**

```
#include <stdio.h>

int main(){
printf("Aloha Monde!\n");
return 0;}
```

Nous reviendrons plus tard, dans des sections dédiées, à la signification des parties de ce premier petit programme. On peut cependant préciser la structure du fichier source d'un programme de façon générale. Un fichier source comporte :

- Déclaration des interfaces des fonctions qui seront utilisées dans le programme (ici, le *# include <stdio.h>*, car la fonction *printf* utilisée est une fonction usuelle du *C*).
- Déclaration des variables globales (il n'y en a pas ici).
- fonction *main*, qui renvoie obligatoirement un entier, d'où le *int main()* et qui peut éventuellement avoir des arguments, cf. la section 1.9 pour le corps de la fonction. La fonction *main* est le point d'entrée du programme, c'est toujours elle que le programme exécutera quand on l'appellera.
- d'autres fonctions qui la plupart du temps seront appelées dans le corps de *main*.

Remarque 1.2.1. *En fait, en* C ANSI*, il n'est pas obligatoire de déclarer le type de la fonction* main*, mais ce sera le cas en* C++*, donc ne l'oubliez pas. De plus, si vous forcez* main *à renvoyer autre chose qu'un entier, vous aurez au mieux un warning lors de la compilation du fichier source, au pire une erreur de compilation.*

Attention, le *C* est sensible à la casse (distingue majuscules et minuscules) donc *main* n'est pas la même chose que *MAIN* ou *Main* ; et c'est bien une fonction *main* qu'il faut absolument dans le fichier source.

Lorsque vous écrivez un programme d'une certaine taille, il est très fortement conseillé d'y ajouter des commentaires afin de rendre moins hermétique le code et de permettre d'éventuelles retouches/évolutions une fois que vous (ou un autre) auront oublié comment est exactement fait le code. En *C*, un commentaire est signalé par les balises (un peu dans l'idée du *XML*) */** et ***/*. Tout ce qui se situera entre ces 2 balises sera traité comme un commentaire, ie. sera ignoré. Attention, il est interdit d'imbriquer des commentaires (ce qui ne sert à rien) en ouvrant deux fois par */**. De même, le commentaire ne peut contenir la fin de balise ***/*, sinon ce serait compris comme la fin du commentaire. Il est de plus interdit de couper un mot par un commentaire. Un exemple de commentaires est le suivant :

Exemple 1.2.2 (Exemples de commentaire)**.**

```
x = 2; /* ceci est un commentaire */
y = 3*sin(x);  /* Ceci est un autre commentaire ,
                  qui prend plus d'une ligne */
z = x*x + x*y; /* Ce commentaire n'est malheureusement
                  pas */ valide et provoquera une erreur */
var/* c'est mal */pasbonne = z*z + x*y*z;
```

Comme vous avez pu le remarquer sur les exemples précédents, les instructions (l'appel à *printf* ou encore les opérations arithmétiques) sont toutes terminées par '*;*'. Ceci est obligatoire pour signifier au compilateur que le texte situer avant le '*;*' représente un tout.

La Compilation

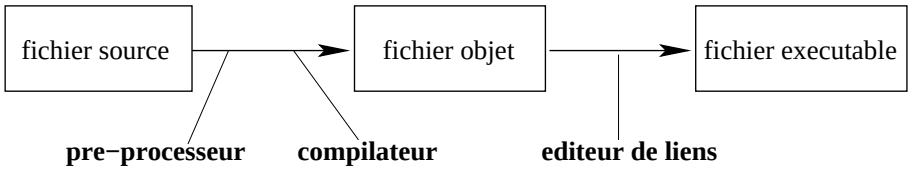
Une fois le fichier source créé, il faut le transcrire du langage *C* dans un langage que comprend l'ordinateur. Ce langage est le *langage machine* et la transcription se fait par l'étape de compilation. Pour donner une idée, ce que fait le compilateur est d'analyser le fichier source, de reconnaître les petits bouts d'instructions qui le compose, éventuellement de les réorganiser de façon plus *optimale*, puis de réécrire ces petits bouts en langage machine (à noter qu'un fichier en langage machine est illisible).

La création du programme exécutable à partir du fichier source nécessite, à proprement parler, plus que la seule étape de compilation. En effet, avant de compiler le fichier source, on lance dessus un pré-processeur qui lui aussi réécrit le fichier source en vrai langage *C*, ie. sans les quelques petits raccourcis de notations permis au programmateur. Ces racourcis sont toutes les lignes de codes commençant par le caractère #. On a déjà vu le # *include* <*nom_header.h*> qui copie le fichier *nom_header.h* dans le fichier source. En général ce fichier contient des déclarations d'interfaces de fonctions (appelées plus communément *prototypes*). On verra par la suite d'autres instructions qui seront préprocessées.

Remarque 1.2.2. *On notera que les instructions à préprocesser ne se terminent pas par un '*;*', étant donné que ce '*;*' est là pour le compilateur, pas le préprocesseur, qui lui mettra les '*;*' où il faut.*

Une fois que le préprocesseur a fait son œuvre, le compilateur fait la sienne et crée un fichier qu'on appelle *objet*. Ce fichier, bien qu'étant en langage compréhensible par l'ordinateur, n'est pas encore exécutable par celui-ci. En effet, dans le fichier objet manque toutes les fonctions *C* prédéfinies que le programmateur a utilisé. Ces fonctions peuvent par exemple se trouver dans des librairies déjà compilées et notre fichier exécutable se doit de connaître toutes ces fonctions. Pour ce faire on passe par l'étape finale d'édition des liens qui consiste tout simplement à piocher dans les primitives *C* ou dans les librairies que l'utilisateur précisera, pour compléter toutes les définitions qu'il nous manque. Par exemple, pour notre programme *aloha monde*, la fonction *printf* ne sera véritablement écrite (recopiée en fait) en langage machine que lors de la dernière étape.

La figure suivante représente les étapes de création d'un fichier exécutable à partir d'un fichier source.



Imaginons que notre programme *aloha monde* soit écrit dans le fichier *aloha.c*. La création du fichier exécutable *aloha* (sous *UNIX*, ou *aloha.exe* sous *Windows*) se fait par les commandes suivantes :

Exemple 1.2.3 (Compilation dans (presque) sa plus simple expression)**.**

```
> gcc -c aloha.c
> gcc aloha.o -o aloha
> ./aloha
Aloha Monde!
```

La première commande lance le préprocesseur et le compilateur sur *aloha.c*, ce qui crée le fichier objet *aloha.o*. La seconde commande fait l'édition des liens qui ici n'a à lier *aloha.o* qu'avec la définition de *printf* qui est une primitive de *C* et donc il n'est nul besoin de préciser qu'il faut chercher cette définition dans *stdio* (*standard input output*).

1.3 Types de Données

Les Types de Données

En *C*, les données manipulées sont typées, ie que quand on utilise une donnée, le langage doit savoir de quelle type il est (entier, caractères...). D'une part, ceci permet de connaître la taille qu'une donnée va prendre en mémoire. D'autre part cela permet de connaître la convention de représentation de cette donnée car toutes les données sont représentées par une succession de 0 et de 1 mais cette succession ne signifie pas la même chose pour un entier et un nombre à virgule.

En *C*, il existe plusieurs types différents et on peut en créer d'autres comme on le verra dans la section 1.7. Le tableau suivant donne une partie des types disponibles en *C*.

Type	Signification	Taille (en octets)	Plage de valeurs
char	Caractère	1	-128 à 127
unsigned char	Caractère non signé	1	0 à 255
short int	Entier court	2	-32768 à 32767
unsigned short int	Entier court non signé	2	0 à $2^{16} - 1 = 65535$
int	Entier	4	-2^{31} à $2^{31} - 1$
unsigned int	Entier non signé	4	0 à $2^{32} - 1$ 4294967295
float	flottant (réel)	4	$3.4 * 10^{-38}$ à $3.4 * 10^{38}$
double	flottant double	8	$1.7 * 10^{-308}$ à $1.7 * 10^{308}$
long double	flottant double long	10	$3.4 * 10^{-4932}$ à $3.4 * 10^{4932}$

Il existe d'autres types, notamment les très importants pointeurs, qui seront vus en section 1.8.

Nombre entier (int)

Un nombre entier est un nombre sans virgule qui peut être négatif ou positif. Il existe un type *short int* non référencé dans le tableau précédent car dans la plupart des cas il correspond au type *int*. Un entier peut être défini dans différentes bases, suivant l'usage que l'on veut en faire :

- Base décimale : c'est la base que vous utilisez tous les jours par défaut, ie une suite de chiffres de 0 à 9. Cependant, en *C*, faites attention de ne jamais commencer l'écriture d'un entier en base décimale par un 0 (sinon le compilateur va croire à une définition en base octale).
- Base hexadécimale : c'est la base 16, dans laquelle un entier est représenté par une suite d'unités : de 0 à 9 ou de A à F, minuscules autorisées. En *C*, pour déclarer qu'on définit un entier en base hexadécimale, on commence l'affectation par 0x ou 0X.
- Base octale : c'est la base 8, où un entier est représenté par une suite de chiffres allant de 0 à 7. On précise la base 8 en débutant l'affectation par 0.

Par exemple :

Exemple 1.3.1 (Entier dans différentes bases)**.**

```
a = 189;  /* en base decimale */
b = 0x35; /* en base hexadecimale, donne b = 3*16+5 = 53 */
c = 047;  /* en base octale, donne c = 4*8+7 = 39 */
```

Par défaut, les entiers sont signés, ie qu'ils peuvent être négatifs. Pour représenter le signe, les ordinateurs utilisent le bit de poids fort et le complément à deux. Celà signifie que pour représenter un entier négatif, on représente sa valeur absolue en binaire (si le bit de poids fort n'est pas à 0, c'est que l'entier est trop grand), on complémente cette représentation binaire puis on lui ajoute 1. Un exemple,

pour un *short int* (donc 2 octets, ie 16 bits) :

$$\begin{array}{rcl} -136 & \overset{|.| \text{ puis binaire}}{\Longrightarrow} & 0000000010001000 \\ & \overset{\text{complément}}{\Longrightarrow} & 1111111101110111 \\ & \overset{+1}{\Longrightarrow} & 1111111101111000 \end{array}$$

Avec ce choix de représentation, si vous additionnez 1 à l'entier (*int*) 32767 (le plus grand entier représentable dans ce type), vous n'obtiendrez pas 32768 (qui n'existe pas) mais -32768, dont la représentation en binaire ne comporte que des 0 sauf sur le bit de poids fort ($10\cdots0$).

L'avantage de cette représentation est que les additions se font bits à bits. Par exemple :

$$\begin{array}{rcl} 137-136 & \overset{\text{binaire}}{\Longrightarrow} & 0000000010001001 \\ & + & 1111111101111000 \\ & = & 0000000000000001 \end{array}$$

Nombre à virgule (float,double)

Un nombre à virgule flottante peut être défini en C de plusieurs manières différentes :
- un entier décimal : 7345
- un nombre comportant une virgule : 314.267
- une fraction : 13./35
- un nombre exponentiel : 3.14e13 (pour $3.4 * 10^{13}$), 2.16792e-2 ou 3.26E10

A part dans le cas d'un entier décimal, l'ordinateur ne sera capable que de stocker une approximation du nombre saisie. Le codage d'un nombre réel x se fait sous la forme $\pm 1, M * 2^n$ où M s'appelle la mantisse et n l'exposant. On peut écrire tout nombre rationnel sous cette forme avec n la part entière de $\log_2 |x|$ ce qui permet de déduire la mantisse qui vaut $|x|/2^n$. Par exemple :

$$-42,4 * \pi \approx -133,203529... \approx -1.04652 * 2^7,\ 0.058 = 1.856 * 2^{-5}$$

Ensuite, suivant que le type du nombre soit *float*, *double* ou *long double*, on a plus ou moins de bits utilisés pour la mantisse et l'exposant (on a toujours le bit de signe en poids fort). Le tableau suivant résume les conventions pour les 3 types :

Type	Nb. total bits	Mantisse	Exposant	Précision
float	32	23	8	6 chiffres après la virgule
double	64	52	11	15 chiffres après la virgule
long double	80	64	15	17 chiffres après la virgule

Représentations des nombres à virgule flottante.

Taille minimum et maximum des entiers et flottants

Pour connaître les tailles des diférents types numériques de votre machine, vous avez à votre disposition les bibliothèques standards *limits* et *float*. Ces dernières définissent des constantes contenant ces tailles. Par exemple, pour afficher la taille de certains des types entiers et flottants, on pourra utiliser le petit programme suivant (qui n'utilise pas toutes les constantes définies par les 2 bibliothèques) :

Exemple 1.3.2 (Affichage de certaines tailles des entiers et flottants)**.**

```
#include <stdio.h>    /* pour pouvoir utiliser printf */
#include <limits.h>   /* taille max & min des types entiers */
```

```
#include <float.h>    /* taille max & min des types flottants */

int main() {
  /* types entiers */
  printf("La valeur maximum d'un 'short' est: %d\n",SHRT_MAX);
  printf("La valeur minimum d'un 'short' est: %d\n",SHRT_MIN);
  printf("La valeur maximum d'un 'int' est: %d\n",INT_MAX);
  printf("La valeur minimum d'un 'int' est: %d\n",INT_MIN);
  printf("La valeur maximum d'un 'unsigned int' est: %u\n",UINT_MAX);
  printf("La valeur maximum d'un 'long' est: %ld\n",LONG_MAX);
  printf("La valeur minimum d'un 'long' est: %ld\n",LONG_MIN);
  /* le type 'long long' n'existe pas forcement sur votre machine */
  /*
    printf("La valeur maximum d'un 'long long' est: %d\n",LLONG_MAX);
    printf("La valeur minimum d'un 'long long' est: %d\n",LLONG_MIN);
  */

  /* types flottants */
  printf("La valeur maximum d'un 'float' est: %e\n",FLT_MAX);
  printf("La plus petite valeur positive d'un 'float' est: %e\n",FLT_MIN);
  printf("Le plus petit epsilon de 'float' est: %e\n",FLT_EPSILON);
  printf("La valeur maximum d'un 'double' est: %e\n",DBL_MAX);
  printf("La plus petite valeur positive d'un 'double' est: %e\n",DBL_MIN);
  printf("Le plus petit epsilon de 'double' est: %e\n",DBL_EPSILON);
  printf("La valeur maximum d'un 'long double' est: %le\n",LDBL_MAX);
  printf("La plus petite valeur positive d'un 'long double' est: %le\n",LDBL_MIN);
  printf("Le plus petit epsilon de 'long double' est: %le\n",LDBL_EPSILON);
  return 0;
}
```

Dans cet exemple, on a abondamment utilisé la fonction *printf* avec des '%d', '%f' et ainsi de suite. On reviendra plus tard sur l'utilisation de printf mais vous aurez compris que ces symboles sont là pour être remplacer par le second argument de la fonction, par exemple, dans le premier *printf*, le '%d' sera remplacé par la valeur de *SHRT_MAX*. Il est important d'utiliser le bon symbole en fonction du type de la variable à afficher, comme on le répètera lorsque l'on parlera des fonction d'entrées/sorties.

Caractère (char)

Le type *char*, pour *character* en anglais, permet de stocker la valeur ASCII (American Standard Code for Information Interchange) d'un caractère, qui est un entier entre 0 et 255 (pour le ASCII étendu, 127 sinon). En réalité, le type *char* stocke par défaut un entier signé, donc entre -128 et 127, ce qui ne veut bien entendu pas dire qu'un caractère à un signe.

Par exemple, le caractère '1' a pour code ASCII 49, 'a' correspond à 97 (puis dans l'ordre), 'A' à 65 (puis dans l'ordre). Plus exhaustivement, la table des caractères ASCII (non étendue) est donnée dans le tableau suivant :

En *C*, on peut donner une valeur à un caractère soit entre ' ', soit par sa représentation ASCII, comme ici :

Exemple 1.3.3 (Affectation d'une variable char)**.**

```
char c;
c = 70; /* c = 'F' */
c = '2'; /* le caractere '2', pas l'entier */
```

Caractère	Code	Caractère	Code	Caractère	Code	Caractère	Code
'/0'	0	Espace	32	@	64	‘	96
	1	!	33	A	65	a	97
	2	”	34	B	66	b	98
	3	#	35	C	67	c	99
	4	$	36	D	68	d	100
	5	%	37	E	69	e	101
	6	&	38	F	70	f	102
	7	’	39	G	71	g	103
	8	(	40	H	72	h	104
	9	)	41	I	73	i	105
	10	*	42	J	74	j	106
	11	+	43	K	75	k	107
	12	,	44	L	76	l	108
	13	-	45	M	77	m	109
	14	.	46	N	78	n	110
	15	/	47	O	79	o	111
	16	0	48	P	80	p	112
	17	1	49	Q	81	q	113
	18	2	50	R	82	r	114
	19	3	51	S	83	s	115
	20	4	52	T	84	t	116
	21	5	53	U	85	u	117
	22	6	54	V	86	v	118
	23	7	55	W	87	w	119
	24	8	56	X	88	x	120
	25	9	57	Y	89	y	121
	26	:	58	Z	90	z	122
	27	;	59	[	91	{	123
	28	¡	60		92	—	124
	29	=	61	]	93	}	125
	30	¿	62	ˆ	94	˜	126
	31	?	63	_	95	□	127

Table des caractères ASCII.

En *C*, il n'existe pas de type pour les chaînes de caractères qui seront représentées soit par un tableau de caractères, soit à l'aide d'un pointeur. Nous verrons ces 2 notions par la suite.

Conversion de type

Bien que *C* soit un langage typé, il l'est très faiblement, ie qu'il est très permissif dans la manipulation des types. On peut par exemple sans problème (de compilation) additionner un *int* avec un *double* avec un *char*. Suivant ce à quoi le résultat d'une telle opération sera affecté, une conversion implicite sera faite.

Exemple 1.3.4 (Exemples de conversion implicite)**.**

```
char c = 'a'; /* 97 en ASCII */
double f = 31.35;
int n = -4;
```

```
n = c + n + f; /* convertira dans le type de n, ie entier
                  on aura donc: n = 97 + 31,35 - 4 = 124 */
c = c + n - 3; /* fera l'addition sur des entiers et convertira
                  le resultat en char, ici c = 'Z' (90 en ASCII) */
n = f; /* converti automatiquement en entier, ici n = 31 */
```

Noter cependant que le compilateur risque de générer un warning. Pour faire une conversion de type propre, il faut qu'elle soit explicite. Ceci se fait à l'aide d'un *cast* qui consiste à spécifier explicitement, entre parenthèses, le type dans lequel une variable doit être converti :

Exemple 1.3.5 (Exemples de conversion explicite)**.**
```
double f = 31.35;
int n ;
n = 3* ((int) f); /* converti f en entier, donc n = 3*31 = 93 */
n = (int) (3*f); /* fois 3 puis converti, donc n = 94 */
f = f + (double) n; /* f = 31.35 + 94.0 = 125.35 */
```

Création d'un type simple

Il est possible de définir un nouveau type de donnée grâce au mot clé *typedef*. La syntaxe est la suivante :

typedef Caracteristiques_du_nouveau_type Nom_du_type

où *Caracteristiques_du_nouveau_type* représente un type déjà existant, comme *int*, *float* ou encore un type plus complexe défini à partir de *struct* comme on le verra à la section 1.7. Et *Nom_du_type* est tout simplement le nom qu'on souhaite donner au nouveau type :

Exemple 1.3.6.
```
typedef long double GrandNombre
typedef float MoinsGrandNombre
```

A noter que cette instruction ne se termine pas par un ';' et doit être placé au début du fichier source.

1.4 Les Variables

1.4.1 Déclaration de variables

Une variable est un objet, définie par un type et un nom et pouvant contenir des données. En général, ces données sont stockées dans une variable en vue d'être manipulées. Le stockage de ces données se fait dans la mémoire de l'ordinateur (disque dur, mémoire vive ou mémoire cache) et occupe un nombre d'octets dépendant du type de la variable.

En *C*, un nom de variable n'a pas de limite de taille, cependant seuls les 32 premiers caractères du nom serviront. Donc en particulier, une variable appelée *chaine_32_caractères*, une autre appelée *chaine_32_caractères_1*, et une autre appelée *chaine_32_caractères_2* (avec *chaine_32_caractères* une chaîne de 32 caractères, la même pour les 3 variables), ne seront pas distinguables. On a quelques autres critères :

- un nom de variable doit commencer par une lettre ou par un '_', donc en particulier pas par un chiffre,
- un nom de variable peut comporter des lettres, des chiffres et le caractère '_', mais pas d'espaces ni de caractères spéciaux,
- une variable ne peut pas avoir pour nom un mot clef du langage, comme par exemple *int*, *float*, *function*, *if* ...

Nous répétons une fois de plus que le *C* est sensible à la casse, donc les variables *toto* et *Toto* ne sont pas les mêmes.

Avant de pouvoir utiliser une variable, il faut la déclarer, ie lui donner un nom et un type afin qu'un espace mémoire puisse lui être réservé. Une variable se déclare tout simplement de la façon suivante :

Exemple 1.4.1.
```
type Nom_de_la_variable;
type Nom_de_la_variable1, Nom_de_la_variable2, ... ;
```

Par exemple :

Exemple 1.4.2.
```
int a;
float x, y, z;
```

1.4.2 Affectation d'une donnée à une variable

Une fois une variable déclarée, on doit lui affecter une valeur, sans quoi son contenu est indéterminé. Ceci se fait tout naturellement avec l'opérateur d'affectation '=', suivant la syntaxe :

$$\text{Nom\_de\_la\_variable} = \text{donnee ou expression}\,;$$

Par exemple, pour stocker la valeur 'a' dans une variable Caractere de type *char*, on écrira :

$$\text{Caractere} = \text{'a'}\,;$$

L'affectation se fait obligatoirement après la déclaration, il est interdit d'affecter une donnée à une variable non déclarée. Par contre, on peut réaliser une affectation en même temps que la déclaration, en suivant la syntaxe suivante :

Exemple 1.4.3.
```
char Caractere = 'a';
int Entier = 1;
int Entier1 = 2, Entier2 = -3, Entier3 = 3, ...;
int Entiern = Entier1 + Entier2;
```

Noter qu'il faut affecter une valeur à une variable avant de l'utiliser, sinon son contenu est aléatoire puisqu'il s'agira simplement de la représentation dans le type de la variable du champs de bit se trouvant à l'endroit de la mémoire réservé lors de la déclaration.

1.4.3 Portée des variables

Selon l'endroit où on déclare une variable, elle sera accessible par toutes les composantes (fonctions) du code ou bien juste par une portion restreinte de ce code. Cette notion est celle de portée ou visibilité d'une variable.

La règle est la suivante :

— une variable déclarée en dehors de toute fonction ou de tout bloc d'instructions sera visible par tous les éléments du code. On parle alors de variable globale.

— une variable déclarée à l'intérieur d'un bloc d'instructions, *ie* entre des accolades, ne sera visible (et donc utilisable) que dans ledit bloc. On parle alors de variable locale.

1.4.4 Les constantes

Une constante est une variable dont la valeur est inchangeable durant l'exécution d'un programme. En langage *C*, une constante se définit grâce à la commande du préprocesseur *#define*, qui remplace toute les occurences d'une chaine de caractères (le nom de la constante) par sa valeur. Par exemple, la commande :

define _Pi 3.14159

remplacera tous les identifiants '_Pi' par la valeur 3.14159, à l'exception de son occurence dans une chaîne de caractères. On notera que comme il s'agit d'une instruction qui sera traitée par le préprocesseur, elle n'est pas terminée par le traditionnel ' *;*'. Voici quelques exemples :

Exemple 1.4.4 (Utilisation de define)**.**

```
resultat = _Pi*cos(theta); /* => remplace */
resultat = _Pi+1; /* => remplace */
resultat = _PiPaPuPo; /* => pas remplace */
printf(''pi = _Pi''); /* => pas remplace */
_Pi = 4; /* => remplace mais genere une erreur */
```

Un problème possible avec la définition de constante par *define* est que cette définition est non typée. Il est donc quelquefois préférable d'utiliser la commande *const*, qui permet de déclarer des constantes typées :

const double e = 2.7182818 ;

Quoiqu'il en soit, la définition de constantes peut être très utile pour rendre le code un peu plus maléable. Par exemple, définir un entier *nblignes* et *nbcols* désignant la taille d'une matrice, permet de changer cette taille et de le faire savoir à tout le code (si ces constantes sont globales) en ne changeant que 2 lignes du code.

1.5 Les Opérateurs en C

En règle générale, on crée des variables dans un programme afin qu'elles soient traitées d'une manière ou d'une autre. Par exemple, si on veut implémenter une méthode de calcul, il va falloir être capable de faire du calcul sur certaines variables. Il va peut-être aussi falloir être capable de prendre plusieurs cas de figures en compte, suivant le signe ou la parité d'une expression. Pour tout cela et plus encore, on possède des opérateurs qui peuvent être classés dans les catégories suivantes :

- les opérateurs de calcul,
- les opérateurs d'assignation,
- les opérateurs de comparaison,
- les opérateurs d'incrémentation,
- les opérateurs de comparaison,
- les opérateurs logiques,
- les opérateurs bit à bit et de rotation de bit.

Noter que les opérateurs ont des priorités différentes, qui permettent d'écrire une expression de façon intelligente sans la surcharger de parenthèses. Cependant, une telle écriture devient très vite illisible pour le commun des mortels et il est donc très fortement conseillé d'utiliser le parenthèsage des expressions où un doute pourrait naître.

1.5.1 Les opérateurs de calcul

Ces opérateurs se retrouvent dans tous les langages de programmation de haut niveau, ce sont les classiques addition, soustraction, multiplication et division. Il est inutile de s'étendre sur ces opérateurs, nous en donnons simplement un tableau récapitulatif :

Opérateurs	Signification	Exemple	Résultat (au début, int x = 9)
+	Additionne 2 nombres	x = x + 1	x reçoit la valeur 10
-	Soustrait 2 nombres	x = x - 18	x reçoit la valeur -9
*	Multiplie 2 nombres	x = 4*x	x reçoit la valeur 36
/	Divise 2 nombres	x = 72/9	x reçoit 8

On peut bien entendu faire plusieurs opérations de calcul à la fois.

Noter que suivant le type des opérandes, l'opération ne renvoie pas le même type. Ainsi, une division entre 2 entiers est par défaut une division entière. Une division où au moins l'une des opérandes est un flottant devient une division flottante. Ainsi, si on souhaite faire la division (non entière) entre 2 entiers a et b, il faut en convertir au moins un en nombre flottant. Pour cela il existe plusieurs possibilités, comme faire un *cast* sur l'une des opérandes, ou encore multiplier au préalable une des opérandes par un flottant (1.0) :

Exemple 1.5.1.

```
int a = 1, b = 3, c;
float x;
c = a/b; /* division entiere => c = 0 */
x = a/b; /* division entiere, le resultat est un float=> x = 0.*/
x = ((float) a)/b; /* division non entiere=> x = 1.3333... */
x = a/((float) b); /* idem */
x = (1.0*a)/b; /* idem */
```

1.5.2 Les opérateurs d'assignation et d'incrémentation

Une spécificité du langage *C* est qu'il permet d'écrire quelques opérations de calcul simple de façon concise. Ceci se fait grâce aux opérateurs d'assignation et d'incrémentation. Encore une fois, rien de compliqué, c'est pourquoi nous nous contentons d'un tableau récapitulatif :

Opérateurs	Exemple	Equivalence
=	x = 3	affectation
+=	x += 3*x	x = x + 3 * x
-=	x -= 2	x = x - 2
*=	x *= x+2	x = x * (x + 2)
/=	x /= 4	x = x / 4
++	x++	x = x+1 ou x += 1
	++x	x = x+1
- -	x- -	x = x-1 ou x -= 1
	- -x	x = x-1

Noter que l'opérateur d'affectation '=' renvoie une valeur correspond à la donnée affectée. On peut donc faire des affections en cascades :

a=b=c=1 ; <=> a=(b=(c=1)) <=> c=1 ; b=c ; a=c ; /* dans cet ordre */

Noter de plus que les opérateurs ++ et - - peuvent être post- et préfixés. S'ils sont mis avant la variable sur laquelle ils s'appliquent, la valeur renvoyée par l'opération est celle de la variable après incrémentation/décrémentation. S'ils sont postfixés, la valeur renvoyée par l'opération est celle de la variable avant incrémentation/décrémentation. Ainsi, le programme suivant :

Exemple 1.5.2 ((Post,Pré)-fixation de ++)**.**

```
int main() {
  int x = 10;
  int y;
  y = x++; /* y = 10 */
  y = ++x; /* y = 12 */
  return 0;
}
```

1.5.3 Les opérateurs de comparaison

Les opérateurs de comparaison permettent (comme leur nom l'indique) de comparer des variables ou des expressions entres elles. Ces opérateurs sont donnés dans le tableau suivant.

Opérateurs	Signification	Exemple	Résultat
==	égalité	x == 3	retourne 1 si x=3, 0 sinon
!=	différence	x != 3	retourne 0 si x=3, 1 sinon
>=	supérieur ou égal	x >=3	retourne 1 si x>=3, 0 sinon
>	supérieur strict	x > 3	retourne 1 si x>3, 0 sinon
<=	inférieur ou égal	x <= 3	retourne 1 si x<=3, 0 sinon
<	inférieur strict	x < 3	retourne 1 si x<3, 0 sinon

On notera que le langage *C* ne possède pas à proprement parler de type booléen et donc que le résultat d'une opération de comparaison est un entier, soit nul soit égal à 1. En réalité, la convention pour représenter un booléen est l'entier 0 pour FAUX et n'importe quoi sauf 0 pour VRAI.

Attention, une erreur fréquente lors d'un test d'égalité est d'utiliser l'opérateur d'affectation '=' au lieu de l'opérateur d'égalité '=='. Une telle étourderie ne mènera pas à une erreur de compilation puisque l'affectation renvoie bien un résultat, par contre le résultat du test ne sera pas celui escompté.

1.5.4 Les opérateurs logiques

En complément des opérateurs de comparaison, on possède également les opérateurs logiques (ou booléens) classiques : ET, OU, NON.

Opérateurs	Signification	Exemple	Résultat
&&	ET	expr1 & expr2	1 si expr1 et expr2 sont $\neq$ de 0, 0 sinon
$\|\|$	OU	expr1$\|\|$expr2	1 si expr1 ou expr2 est $\neq$ de 0, 0 sinon
!	NON	!expr1	retourne 1 si expr1 vaut 0, 0 sinon

1.5.5 Les opérateurs bit à bit et de rotation de bit

Finalement, si on veut travailler à un niveau plus proche de la représentation binaire des données, on en a la possibilité grâce aux opérateurs bit à bit. Puisqu'un bit vaut soit 0 (FAUX) soit 1 (VRAI), les opérateurs bit à bit sont des opérateurs booléens qui traitent les expressions bit à bit mais renvoient tout de même un résultat dans la représentation d'origine des variables/expressions traitées.

On notera au passage, que l'opérateur $^{\prime\prime}$ n'est pas l'opérateur puissance, qui est en fait une fonction de la bibliothèque standard *math.h* et s'appelle *pow*. De plus, les opérateurs bit à bit ne sont valides que pour des arguments de type entier ou caractère (qui sont des entiers).

Opérateurs	Signification	Exemple (binaire)	Résultat
&	ET bit à bit	9 (1001)& 10 (1010)	8 (1000)
|	OU bit à bit	9 (1001)| 10 (1010)	11 (1011)
^	OU exclusif bit à bit	9 (1001)^ 10 (1010)	3 (0011)

En plus de ces opérateurs bit à bit, on a aussi la possibilité de faire des rotations de bits sur les entiers qui correspondent en fait à des division (décalage a droite) ou multiplication (décalage à gauche) par 2. On peut faire plus d'une rotation dans chaque direction.

Opérateurs	Signification	Exemple (binaire)	Résultat
<<	Rotation à gauche	7 (111)<< 1	14 (1110)
>>	Rotation à droite	15 (1111) >> 2	3 (0011)

On notera qu'il ne s'agit pas exactement d'une rotation puisque pour la rotation à droite, le bit de poids faible est perdu plutôt que de devenir le bit de poids fort. On notera de plus que ces opérateurs traitent des champs de 32 bits.

1.6 Les Structures Conditionnelles et de Boucle

Ces structures sont au nombre de 4, il s'agit des instructions *if* (avec éventuellement *else*), *for*, *switch* et *while*. Ces instructions permettent d'exécuter des blocs d'instructions si certaines conditions sont remplies (*if*,*switch*), ou plusieurs fois de suite (*for*,*while*).

Un bloc d'instructions est une suite d'instructions réunies ensembles et encadrées par des accolades ouvrante '{' puis fermante '}'.

Lorsque vous utilisez des structures conditionnelles ou de boucle, il est conseillé d'indenter l'écriture du code, pour des raisons de lisibilité.

1.6.1 L'instruction if...else

L'instruction *if* est l'instruction de test la plus simple, elle correspond à un test *si ... alors*. Elle réalise un bloc d'instructions si une condition est satisfaite. Elle s'écrit sous la forme :

Exemple 1.6.1.
```
if (condition) {
  liste d'instructions;
}
```

Si la liste d'instructions ne comporte qu'une seule instruction, les accolades ne sont pas nécessaires. La condition est une expression booléenne et peut donc comporter des opérateurs logiques et des opérateurs de comparaison.

La plupart du temps, on souhaite exécuter un bloc d'instructions si une condition est réalisée mais aussi exécuter un autre bloc d'instructions dans le cas contraire. D'où l'instruction *if...else*, correspondant à un *si ... alors ... sinon* :

Exemple 1.6.2.
```
if (condition) {
  liste d'instructions;
}
else{
```

```
  liste d'instructions;
}
```

On notera qu'il existe une structure de test beaucoup plus succinte dans le cas ou les 2 listes d'instructions sont réduites à deux listes unitaires. La syntaxe est la suivante :

Exemple 1.6.3.
```
(condition) ? instruction si vrai : instruction si faux
```

Dans ce cas, la condition doit être entre parenthèses, l'instruction à gauche du ' :' est réalisée si la condition est vraie, celle de droite est réalisée si la condition est fausse. De plus, la structure ? renvoie la valeur résultant de l'instruction exécutée. Par exemple :

Exemple 1.6.4.
```
minab = ((a>b) ? b : a);
```

renvoie dans *minab* le minimum entre a et b.

On notera qu'en *C*, l'évaluation des expressions booléennes se fait séquentiellement. C'est à dire que si la première partie de l'expression booléenne suffit à l'évaluer, le reste de l'expression booléenne ne le sera pas. Ceci est particulièrement utile dans le cas où une partie de l'expression booléenne n'est pas toujours définie (l'accès à un élément d'un tableau). Par exemple, la séquence d'instructions suivante est bien définie (en anticipant un peu sur l'introduction des tableaux) :

Exemple 1.6.5.
```
int Tableau[10] = {1,2,3,4,5,6,7,8,9,10};
int indice = 20;
if ((indice<10)&&(indice>=0)&&(Tableau[indice]==5)) {
  printf('Tableau[%d] = %d\n',indice-1,Tableau[indice]);
}
```

Cette suite d'instructions ne fera rien (car on n'entre pas dans la boucle). Par contre, si on avait mis la condition *Tableau[indice]==5* en premier, cela aurait provoqué une erreur d'exécution, puisque Tableau[20] n'est pas définie.

1.6.2 L'instruction switch

Cette instruction permet de tester une variable et d'exécuter différentes instructions suivant la valeur de son contenu. Il s'agit d'un branchement conditionel, sa syntaxe est la suivante :

Exemple 1.6.6.
```
switch (Variable){
case Valeur1:
     liste d'instructions;
     break;

case Valeur2:
     liste d'instructions;
     break;
....

default:
     liste d'instructions;
     break;
}
```

Lorsque la Variable est égale à Valeur1, on exécute la première liste d'instructions, si elle est égale à Valeur2, on exécute la seconde liste d'instructions, et ainsi de suite. Si aucun des cas ne correspond à Variable, on rentre dans la branche *default*. Dans le cas où l'instruction *break* ne finit pas le *case*, le programme entrera dans le *case* suivant. Cela peut s'avérer utile si on souhaite exécuter la même liste d'instructions pour plusieurs valeurs différentes de Variable. Par exemple :

Exemple 1.6.7.
```
switch (Variable){
case Valeur1:
case Valeur2:
     liste d'instructions (pour Valeur1 et Valeur2);
     break;
case Valeur3:
     liste d'instructions;
     break;
...
default:
     liste d'instructions;
     break;
}
```

Il est conseillé de toujours avoir un cas *default*, ne serait-ce que pour afficher un message d'erreur. Cela permet entre autre de rendre le code plus robuste.

1.6.3 L'instruction for

L'instruction *for* est une instruction de boucle qui permet d'exécuter plusieurs fois un même bloc d'instructions. Le nombre d'exécutions successives du bloc dépend d'un compteur, qui sera modifié après chaque exécution du bloc et testé avant chaque exécution de ce bloc. Sa syntaxe est :

Exemple 1.6.8.
```
for (initialisation du compteur; condition pour continuer; modification du compteur) {
  liste d'instructions;
}
```

Par exemple :

Exemple 1.6.9.
```
int i, x = 0;
for (i=1; i<=10; i++) {
x = x + i;
}
```

Cette boucle s'exécutera 10 fois et à sa sortie, x aura pour valeur $\sum_{i=1}^{10} i = 55$.

L'utilisation d'une simple incrémentation $i{+}{+}$ est assez classique dans une boucle, cependant la modification du compteur peut utiliser n'importe quel opérateur de calcul, d'assignation ou d'incrémentation.

Il faut faire très attention à bien vérifier que la boucle ait bien une condition de sortie, ie qu'à un moment ou à autre, la condition n'est plus vraie. Il faut de plus bien compter le nombre de fois que la boucle s'exécutera. Par exemple, dans le cas précédent, si la condition avait été $i < 10$, alors la boucle aurait été exécutée 9 fois, et non 10.

1.6.4 L'instruction while

L'instruction *while* correspondant à un *tant que ....* C'est une instruction de boucle conditionelle dont la syntaxe est la suivante :

Exemple 1.6.10.
```
while (condition) {
  liste d'instructions;
}
```

Dans ce cas, la liste d'instructions est exécutée tant que la condition est réalisée. Donc, pour pouvoir sortir de la boucle, il faut absolument que la liste d'intructions modifie tout ou partie des composantes de la condition afin qu'après un certain nombre d'exécutions, cette condition puisse devenir fausse.

On notera que l'instruction *while* est une généralisation de l'instruction *for* où la modification du compteur peut être très compliquée.

L'instruction *while* est très bien appropriée dans le cas d'un algorithme itératif avec un critère d'arrêt (recherche de zéro par exemple) ou encore dans le cas d'un programme contenant un menu qui en mode texte qui réapparait tant que l'utilisateur n'a pas décidé de stopper le programme.

1.6.5 L'insctruction do .. while

L'instruction *do ... while* correspond à répéter ... tant que. Elle permet d'exécuter une liste d'instructions tant qu'une condition est vrai. La différence avec l'instruction *while* est qu'on exécute d'abord la liste d'instructions avant de vérifier la condition. Donc on exécutera au moins une fois la liste d'instructions, contrairement au *while*. La syntaxe de l'instruction *do ... while* est :

Exemple 1.6.11 (L'instruction do ... while)**.**
```
do {
  liste d'instructions;
} while (condition);
```

Comme pour l'instruction *while*, il est préférable de s'assurer que la condition devienne fausse au bout d'un nombre fini d'itérations.

1.7 Types de Données Complexes

1.7.1 Les Tableaux

Tableaux unidimensionnels : Les variables que nous avons vues à la section 1.4 étaient toutes de types élémetaires. En particulier, elles ne permettent que de stocker une valeur à la fois. Or, il est souvent utile de pouvoir stocker un nombre important de variables de types identiques. Ceci se fait grâce au type de donnée tableau qui dans son expression la plus simple correspond à celle de vecteur, puis à celle de matrice si on augmente la dimension. Un tableau est une collection ordonnée de variables d'un même type stockées en mémoire dans des blocs continus.

La déclaration d'un tableau unidimensionnel se fait de la façon suivante :

type Nom_Du_Tableau [Nombre d'elements] ;

où *type* est le type des éléments du tableau (qui sont donc tous du même type). La taille qu'occupe un tableau en mémoire est la taille qu'occupe le type des données multipliée par le nombre d'éléments. Par exemple, un tableau de 10 *double* se déclare de la manière suivante :

double Tableau [10] ;

Ce tableau occupera un espace de 10*8=80 octets en mémoire, soit 640 bits. Pour accéder à un élément du tableau, on utilise le nom du tableau suivi de l'indice de l'élément entre crochets. Il faut toutefois faire attention, car **en *C*, les indices des tableaux commencent à 0**. Ainsi, pour accèder au 5ème élément de notre tableau de double, on écrira :

Tableau[4] ;

On peut manipuler un élément d'un tableau tout comme une variable classique et s'en servir dans une expression ou lui affecter une valeur. De plus, pour initialiser un tableau, il n'est bien entendu pas nécessaire de le faire élément par élément. On peut le faire comme suit, à la déclaration :

int Tableau[15] = {0,3,4,7,2,7,...} ;

La liste de valeurs entres accolades ne doit pas comporter plus d'éléments que le tableau. Cependant, elle peut en comporter moins, auquel cas les éléments non affectés du tableau prendront la valeur 0. De plus, les éléments entre accolades doivent être des constantes. En particulier, pour initialiser un tableau à 0, il suffit d'écrire :

type Nom_Du_Tableau [Nombre d'elements] = {0} ;

On peut également utiliser une boucle *for* dont le compteur désignera l'indice de l'élément à affecter. Par exemple, pour calculer les 100 premiers termes de la suite de Fibonacchi :

Exemple 1.7.1.
```
int Fibo[100] = {1,1};
int indice;
for (indice=2,indice<100,indice++){
Fibo[indice] = Fibo[indice-1] + Fibo[indice-2];
}
```

Lorsqu'on utilise un tableau, on fait souvent appel à sa taille. C'est pourquoi il est fortement conseillé de définir la taille d'un tableau à l'aide d'une macro *define*. Par exemple :

Exemple 1.7.2.
```
#define NB_ELEMENTS_FIB 50
int Fibo[NB_ELEMENTS_FIB] = {1,1};
int indice;
for (indice=2,indice<NB_ELEMENTS_FIB,indice++){
Fibo[indice] = Fibo[indice-1] + Fibo[indice-2];
}
```

Ceci permet de pouvoir changer la taille du tableau en ne changeant que la valeur de la macro *NB_ELEMENTS_FIB*.

Tableaux Multidimensionnels Un tableau multidimensionnel est en fait un tableau de tableau. Il peut comporter autant de dimension que l'on souhaite, et il se déclare de la manière suivante :

type Nom_Du_Tableau [n1][n2][n3]...[nN] ;

Chaque élément *ni* désigne le nombre d'éléments de la dimension i du tableau. Si N est égale à 2 alors on a affaire à une matrice. On accède aux éléments d'un tableau multidimensionnel de la même manière que pour un tableau unidimensionnel, en spécifiant les indices de l'élément souhaité :

Exemple 1.7.3.
```
int A[5][5];
A[0][0] = 4;
```

Un tableau multidimensionnel peut être initialisé de la même manière qu'un tableau unidimensionnel, ie. avec des boucles imbriquées ou avec une affectation à la déclaration. Typiquement, l'affectation avec des boucles imbriquées se fait de la manière suivante.

Exemple 1.7.4.
```
#define NB_LIGNES 10
#define NB_COLS 10
int A [NB_LIGNES][NB_COLS];
int i,j;
for (i=0,i<NB_LIGNES,i++){
  for (j=0,j<NB_COLS,j++){
    A[i][j] = ...;
  }
}
```

Dans le cas où on souhaite initialiser un tableau multidimensionnel à la déclaration, on peut utiliser une liste d'éléments entre accolades. Cependant, il faut bien faire attention à l'ordre dans lequel les éléments d'un tableau multidimensionnel sont stockés en mémoire. Ainsi, les éléments d'un tableau $tab[n][m][p]$ sont stockés dans l'ordre suivant :

$tab[0][0][0]$	$tab[0][0][1]$	$\cdots$	$tab[0][0][p-1]$	$tab[0][1][0]$	$\cdots$
$tab[0][1][p-1]$	$\cdots$	$tab[0][m-1][p-1]$	$tab[1][0][0]$	$\cdots$	$tab[n-1][m-1][p-1]$

Donc en particulier, cela veut dire que pour une matrice (2 dimensions), ses éléments sont stockés lignes par lignes. L'initialisation suivante :

$$\text{int A[3][4] = \{1,2,3,4,5,6,7,8,9,10\} ;}$$

aura pour résultat la matrice 3 par 4 :

Exemple 1.7.5.
$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 0 & 0 \end{bmatrix}$$

Notons qu'il est cependant préférable d'expliciter la structure ligne par ligne durant l'initialisation en faisant des *blocs* avec les éléments, par exemple, pour la matrice A précédente :

Exemple 1.7.6.
```
int A[3][4] = {{1,2,3,4},
               {5,6,7,8},
               {9,10}};
```

1.7.2 Les Chaînes de Caractères

Comme nous l'avons vu dans notre première introduction aux types de données, il n'existe pas de type propre aux chaînes de caractères. On représentera une chaîne de caractères par un tableau de caractères, terminé par un caractère spécial de fin de chaîne : le caractère nul '\0'. Par exemple, la chaîne 'Salut', est représentée de la manière suivante :

S	a	l	u	t	\0

Pour créer une chaîne de caractères, on la déclare donc comme un tableau de caractères. Avec une telle représentation, on doit fixer à l'avance la taille de la chaîne. On peut par contre prévoir une grande taille et ne l'utiliser que partiellement, puisque la fin de la chaîne n'est pas la fin du tableau mais la première occurence du caractère de fin de chaîne '\0'. De plus, il faut toujours penser à déclarer un tableau du nombres de caractères de la chaîne plus un, pour tenir compte du caractère de fin de chaîne. Par exemple, pour une chaîne de 20 caractères, il faudra déclarer un tableau d'au minimum 21 éléments :

char Chaine_Car[21] ;

Ensuite, pour manipuler une chaîne de caractères, on possède tout un attirail défini dans les bibliothèques standards, surtout *string.h*. Quelques fonctions utiles à la manipulation de chaîne de caractères :

- **strcmp()** : compare 2 chaînes de caractères et renvoie un entier négatif, nul ou positif suivant que la première chaîne de caractère placée en argument est respectivement inférieure, égale ou supérieure à la seconde chaîne de caractères.
- **strncmp()** : de même que *strcmp()* mis à part que *strncmp()* accepte 3 arguments, le dernier étant un entier n spécifiant qu'il ne faut comparer que les n premiers caractères des 2 chaînes.
- **strcpy()** : cette fonction prend en premier argument une chaîne destinataire *dest* et en second une chaîne source *src* et copie la seconde dans la première. Une version ne copiant que les n premiers caractères de *src* dans *dest* est également disponible. La fonction renvoie la nouvelle chaîne *dest*.
- **strcat()** : cette fonction prend en premier argument une chaîne destinataire *dest* et en second argument une chaîne source *src* et concatène *src* au bout de *dest* en prenant soin d'écrire sur le caractère de fin de chaîne de *dest*. Donc en gros il s'agit d'une fonction pour mettre bout à bout 2 chaînes de caractères. La fonction renvoie la nouvelle chaîne *dest*.

On notera qu'un test d'égalité entre 2 chaînes de caractères ne se fait pas avec l'opérateur '=='. Il existe de nombreuses autres fonctions dans la bibliothèque *string.h*, soin est laisser au lecteur de les découvrir (le *man* linux/unix est pour cela très pratique). Un exemple d'utilisation des chaînes de caractères et des fonctions de *string.h* est le suivant :

Exemple 1.7.7.

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

int main(){
  char phrase[100], mot1[20] = "Aloha", mot2[20] = "Monde";
  char phrasecpy[100];

  strcat(phrase,mot1);
  strcat(phrase," ");
  strcat(phrase,mot2);
  strncpy(phrasecpy,phrase,10);
  phrasecpy[10] = '\0';

  printf("Ma phrase est: %s\n",phrase);
  printf("Les 10 premiers caracteres de ma phrase sont: %s\n",phrasecpy);
  return 0;

}
```

Dans cet exemple, on voit qu'il faut rajouter le caractère de fin de chaîne après la copie tronquée de *phrase*, sans quoi *phrasecpy* n'est pas une chaîne de 10 caractères. Il est important de noter que l'affectation d'une valeur à une chaîne de caractères à l'aide de '=' n'est autorisé qu'au moment de la déclaration de la chaîne. En particulier :

Exemple 1.7.8.
```
...
  char mot1[20] = "Aloha"; /* autorise */
  char mot2[20];
  mot2 = "bonjour"; /* interdit => erreur a la compilation*/
...
}
```

1.7.3 Les Structures

Les tableaux de la partie précédente permettent de stocker un grand nombre d'éléments mais tous de même type. Si on veut créer un type de donnée plus complexe, réunissant plusieurs éléments de types différents, on utilise l'instruction *struct*. Une structure est composée de champs (ses éléments) et se déclare de la façon suivante :

Exemple 1.7.9.
```
struct Nom_de_la_Structure {
  type_champ1 Nom_Champ1;
  type_champ2 Nom_Champ2;
  ...
};
```

Dans cette instruction, les noms des champs doivent tous être différents et peuvent être de n'importe quel type excepté le type de la structure. Ainsi, la structure suivante est valide :

Exemple 1.7.10 (Exemple de structure valide)**.**
```
struct Eleve {
  int Age;
  double Moyenne;
  struct Classe Niveau;
};
```

Mais la structure suivante ne l'est pas :

Exemple 1.7.11 (Exemple de structure non valide)**.**
```
struct Eleve {
  int Age;
  double Moyenne;
  float Moyenne;
  struct Classe Niveau;
  struct Eleve voisin;
};
```

Cette structure n'est pas valide car d'une part elle possède 2 champs de même nom (Moyenne) et d'autre part elle possède un champ ayant son propre type.

La déclaration d'une structure ne fait que définir les caractéristiques de la structure. Une fois la structure déclarée, on peut déclarer une variable du type de la structure :

Exemple 1.7.12.
```
struct Nom_de_la_Structure Nom_Variable;
/* ou encore */
struct Nom_de_la_Structure Nom_Variable1, Nom_Variable2, ...;
```

Pour accèder aux champs d'une variable structurée, on fait suivre le nom de la variable par un point '.' et le nom du champ auquel on veut accèder (sauf quand le champ est un pointeur, comme on le verra dans la section dédiée à ces derniers). Par exemple, pour la structure Eleve (la valide) :

Exemple 1.7.13.
```
struct Eleve Benjamin;
Benjamin.Age = 16;
Benjamin.Moyenne = 18.5;
Benjamin.Niveau = 1;
```

La structure étant un type comme un autre, on peut l'utiliser pour former un tableau :

struct Nom_de_la_Structure Nom_Tableau [Nb_Elements] ;

De plus, il peut être intéressant de renommer une structure grâce au mot clé *typedef* qui permet de faire de la structure un type comme les autres. Par 'type comme les autres', il faut comprendre un type dont les variables sont déclarées par *Nom_Type Variable* plutôt que par *struct Nom_Struct Variable*. Par exemple, pour notre structure *Eleve* :

Exemple 1.7.14.
```
typedef struct Eleve {
  int Age;
  float Moyenne;
  struct Classe Niveau;
} Eleve;

...
int main(){
  Eleve Benjamin; /* Et plus struct Eleve Benjamin */
  ...
  return 0;
}
```

1.7.4 Les Types Enumérés

En C, on peut également créer ce qu'on appelle un *type énuméré*. C'est en fait une facilité de codage qui fait correspondre un intervalle d'entier à une collection de nom. Par exemple, si on souhaite utiliser un type *Couleur* prenant pour valeur *rouge*, *vert* ou *bleu*, on pourra utiliser la syntaxe suivante :

Exemple 1.7.15.
```
enum Couleur {rouge, vert, bleu};
...
int main(){
  enum Couleur ma_couleur;
  ...
}
```

Dans notre exemple, la variable *ma_couleur* pourra prendre les valeurs *rouge*, *vert* ou *bleu* (attention, lors d'affectation, ce ne sont pas des chaînes de caractères, donc pas de guillemets). Un type énuméré est en fait un *alias* entre les valeurs possibles du type et les entiers positifs ou nuls. Dans le cas du type Couleur, la correspondance sera : *rouge* = 0, *vert* = 1, *bleu* = 2.

Encore une fois, on peut utiliser un *typedef* lors de la définition du type énuméré, afin de pouvoir utiliser *Couleur* comme un vrai type.

1.8 Les Pointeurs

1.8.1 La Notion de Pointeur

Un pointeur est une variable contenant l'adresse d'une autre variable d'un type donné. Les pointeurs sont probablement ce qui fait du *C* un langage si puissant. Ils permettent de définir des structures de données dynamiques.

On sait déjà qu'une variable d'un type donné doit être stockée quelque part en mémoire. L'endroit où est stocké une variable est appelé son adresse. La valeur d'un pointeur est donc cette adresse associée à la connaissance du type de donnée qui est stocké à cette adresse (sans quoi le champ de bits se trouvant à l'adresse n'est pas interprétable puisqu'on ne connaîtrait ni sa taille ni sa représentation).

Les intérêts principaux des pointeurs sont :

— En possédant l'adresse d'une variable, celle-ci peut être de taille aussi importante qu'on peut l'imaginer, le pointeur associé n'en sera pas de taille plus importante. Donc si on veut communiquer une variable à une fonction, il suffira d'en donner l'adresse (le pointeur) pour que la fonction puisse accéder à la valeur de la variable.

— Toujours pour la communication avec une fonction, lui donner l'adresse d'une variable plutôt que directement sa valeur, autorise la fonction à modifier cette variable. C'est ce qu'on appellera dans la section 1.9, le passage par adresse.

— Il est possible de créer des tableaux dynamiques, ie qu'on n'aura plus forcément besoin de fixer à l'avance la taille d'un tableau. En particulier, ce sera utile pour les chaînes de caractères.

— Les pointeurs permettent de construire des structures chaînées, ie de contourner le fait qu'une structure définie par l'instruction *struct* ne puisse pas avoir un champ de son propre type. On en verra un exemple en la personne des listes chaînées.

1.8.2 Déclaration et Manipulation d'un Pointeur

Tout comme les autres types de variable, un pointeur doit être déclaré avant d'être utilisé. Pour la déclaration et l'utilisation de pointeurs, deux opérateurs unaires non encore introduits sont très importants, ce sont les opérateurs * et & (à ne pas confondre avec l'opérateur binaire de même nom : le ET bit à bit). L'opérateur * permet d'accéder à la valeur associée à une adresse. L'opérateur &, quand il est appliqué à une variable, renvoie son adresse. On notera qu'un pointeur, étant lui-même une variable, possède également une adresse.

La déclaration d'un pointeur se fait de la façon suivante :

type * Nom_Pointeur ;

Le type de la variable pointée par le pointeur peut être n'importe quel type déjà défini (int, char, float, type complexe défini avec struct, voir même int *, float*, int **...). Grâce au type du pointeur, le compilateur saura combien de blocs mémoires sont à réserver après le bloc pointé.

L'initialisation d'un pointeur se fait grâce à l'opérateur unaire & :

Nom_Pointeur = &Nom_Variable_Pointee ;

Un exemple d'utilisation :

Exemple 1.8.1.

```
int * ad;
int n;
n = 10;
ad = &n; /* ad recoit l'adresse de l'entier n */
*ad = 20; /* l'entier stocke a l'adresse ad recoit 20 */
```

A la fin de cet exemple, la valeur de l'entier n est 20. On verra dans la section 1.9, dédiée aux fonctions, l'intérêt du passage par l'adresse pour changer la valeur d'une donnée.

1.8.3 Tableau et Chaîne de Caractères

Un tableau, et incidemment une chaîne de caractères, peut être vu comme un pointeur sur le premier élément du tableau. En effet, la déclaration d'un tableau affecte des cases mémoires contigües, et donc, à partir de la connaissance du premier élément d'un tableau et du type des éléments du tableau, on peut le parcourir et accéder à n'importe quel indice.

Ainsi, la déclaration d'un pointeur sur un type (*int*, *float*, *char* ...) peut être le premier pas vers la définition d'un tableau de ce type. Une fois la déclaration d'un pointeur effectuée, on peut faire appel à la fonction *malloc* qui permet d'affecter un espace de mémoire d'une taille spécifiée en argument. Cette façon de faire permet des allocations dynamique de mémoire pour un tableau, dans le sens où le compilateur n'a pas besoin de connaître a priori la taille d'un tableau qui sera déterminée lors de l'exécution. Un exemple est le suivant :

Exemple 1.8.2.

```
#include <stdio.h>
#include <stdlib.h> /* pour malloc() et sizeof() */

int main(){
  int * TabEntier;
  char * Chaine;
  int i, n;

  printf("Combien d'entiers voulez-vous stocker? ");
  scanf("%d",&n); /* lecture d'un entier, le & pour marquer
                       l'adresse de n */
  TabEntier = (int *) malloc(n*sizeof(int)); /* allocation memoire */
  for(i=0;i<n;i++) {
    printf("Entier numero %d = ",i+1);
    scanf("%d",&TabEntier[i]);
  }
  printf("Taille de la phrase a stocker (nb caracteres)? ");
  scanf("%d",&n);
  Chaine = (char *) malloc(n*sizeof(char)); /* allocation memoire */
  printf("Tapez la phrase: ");
  scanf("%s",Chaine);
  printf("Votre phrase est: %s\n",Chaine);
  free(TabEntier); /* lib\'erer la m\'emoire allou'ee */
  free(Chaine); /* lib\'erer la m\'emoire allou'ee */

  return 0;
}
```

1.8.4 Les Listes Chaînées

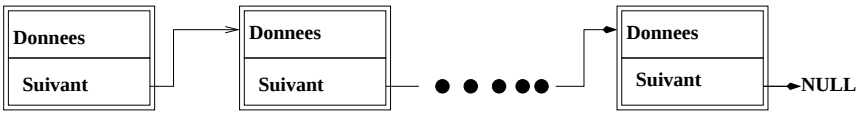
Une liste chaînée est une structure complexe s'autoréférençant. Cette structure comportera des *maillons* qui seront reliés les uns aux autres, ie un ou plusieurs champs pointant vers une structure identique. Suivant le nombre de ces champs et leur signification, on aura affaire à différents types de liste chaînée :

— Lorsque la structure contient des données et un unique pointeur vers la structure suivante on parle de liste chaînée simple.
— Lorsque la structure contient des données et deux pointeurs, un vers la structure suivante, un autre vers la structure précédente, on parle de liste chaînée double.
— Lorsque la structure contient des données et deux pointeurs pointants vers deux structures suivantes (fils), on parle d'arbre binaire.

Liste Chaînée Simple Par exemple, la structure suivante correspond à une liste chaînée contenant un entier et un nombre à virgule flottante :

Exemple 1.8.3.
```
struct Ma_Liste {
int mon_entier;
double mon_nombre;
struct Ma_Liste * pSuivant;
};
```

On notera qu'on a ainsi détourné la restriction voulant qu'il est interdit d'avoir une structure ayant un champ de son propre type. On a ainsi une structure récursive. Cependant, elle ne peut être infinie et il nous faut donc un moyen de la stopper. Cela se fait en assignant au pointeur sur l'élément suivant la valeur NULL, quand le maillon considéré est le dernier. On a également besoin d'un pointeur sur le premier élément de la liste, qui lui ne fera pas parti d'un maillon. On représentera une telle liste de la manière suivante, chaque maillon étant une variable de type Ma_Liste.



Une fois la structure définie, il reste encore à la déclarer. Pour se faire, on utilise deux éléments du type pointeur de la liste, un qui définira la tête de la liste (son commencement), un autre pointant sur un éventuel nouveau maillon.

Exemple 1.8.4.
```
struct Ma_Liste *Nouveau;
struct Ma_Liste *Tete;
Tete = NULL;
```

Pour l'instant notre liste est vide, sa tête ne pointant vers rien. Afin de peupler notre chaîne, il faut pouvoir ajouter des maillons, mais pour cela il faudra à chaque fois réserver un espace mémoire où stocker le nouveau maillon. Comme a priori on ne connaît pas le nombre de maillons de la chaîne, il va falloir être capable de réserver de l'espace mémoire sans passer par l'étape de déclaration. Ceci se fait par l'intermédiaire des fonctions *malloc* et *sizeof* de la bibliothèque *stdlib.h*. Les opérations les plus utiles dans une liste chaînée, sont l'ajout d'un premier élément, l'ajout d'un élément en fin de liste et le parcours de la liste. Pour l'ajout d'un premier élément, on procède de la façon suivante :

Exemple 1.8.5.
```
/* pour ne pas avoir a rappeler struct */
```

```
typedef struct Ma_Liste Ma_Liste;
Nouveau = (Ma_Liste*)malloc(sizeof(struct Ma_Liste));
/* allocation memoire d'un nouveau maillon */
Nouveau->pSuivant = Tete;  /* l'ancien premier element suit le nouveau */
Tete = Nouveau; /* le nouveau maillon est maintenant la tete */
```

Pour l'ajout d'un dernier élément, il faut tout d'abord parcourir la liste, puis allouer de la mémoire pour un nouveau maillon et finalement ajouter l'élément.

Exemple 1.8.6.

```
Ma_Liste * Courant;
if (Tete != NULL) {
  Courant = Tete;
  while (Courant->pSuivant != NULL) Courant = Courant->pSuivant;
}
Nouveau = (Ma_Liste*)malloc(sizeof(struct Ma_Liste));
Courant->pSuivant = Nouveau;
Nouveau->pSuivant = NULL;
```

Liste Chaînée Double La liste chaînée simple ne permet de parcourir la liste que dans une seule direction. Pour pouvoir le faire dans les deux directions, on utilise une liste chaînée double, dont un exemple est le suivant :

Exemple 1.8.7.

```
struct Ma_Liste_Double {
int mon_entier;
double mon_nombre;
struct Ma_Liste_Double * pPrecedent;
struct Ma_Liste_Double * pSuivant;
};
```

On peut également utiliser cette liste chaînée double pour faire une liste circulaire/bouclée ou même un arbre binaire (auquel cas les 2 pointeurs sur *Ma_Liste_Double* seraient les fils).

1.8.5 Pointeurs Génériques

Une fonctionnalitée intéressante des pointeurs est qu'il est permis de ne pas leurs donner de types lors de la déclaration, en les déclarant comme *void**. Ceci permet de créer un pointeur générique auquel on pourra dynamiquement affecter un type particulier. Par exemple :

Exemple 1.8.8.

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

int main(){
  void * Tab = NULL;
  int i,n;

  printf("Type de donnees a stocker?\n");
  printf("1. Entier\n");
  printf("2. Flottant sinple precision\n");
  printf("3. Flottant double precision\n");
```

```
  printf("Votre choix: ");
  scanf("%d",&n);
  switch (n) {
  case 1:
    printf("Nombre d'entiers a stocker? ");
    scanf("%d",&n);
    Tab = (int*) malloc(n*sizeof(int));
    for(i=0;i<n;i++) scanf("%d",&Tab[i]); ...
    break;
  case 2:
    ... Tab = (float*) malloc(n*sizeof(float)); ...
  case 3:
    ... Tab = (double*) malloc(n*sizeof(double)); ...
  default:
    printf("!!! Entree incorrecte !!!\n");
    break;
  }
  if (Tab != NULL)
    free(Tab); /* liberer la memoire allouee */
  return 0;
}
```

La possibilité d'avoir un pointeur générique permet entre autre de construire des structures de données génériques, comme par exemple une pile contenant des données spécifiées lors de l'exécution. Ainsi, on peut définir des fonctions manipulant des pointeurs génériques, et qui accepteront donc tout type d'argument pourvu que les instructions de la fonction aient un sens pour le type qui sera spécifié à l'exécution.

Ceci étant une fonctionnalité relativement évoluée du *C*, nous n'iront pas plus avant dans sa présentation.

1.9 Les Fonctions

1.9.1 Définition

Dans le langage *C*, une fonction est plus qu'une suite d'instructions renvoyant une valeur. La notion de fonction, en *C*, inclut également celle de procédure, ie une suite d'instructions réalisant une action.

Une fonction se définit par son argument de sortie, ses arguments d'entrées, et sa liste d'instructions. La déclaration d'une fonction se fait par la syntaxe suivante :

Exemple 1.9.1.

```
type_donnee Nom_Fonction (type1 arg1, type2 arg2, ...) {
  liste d'instructions;
  return ...;
}
```

Quelques remarques :

- Le type *type_donnee* est celui de la valeur renvoyée par la fonction.
- Tout comme pour le *main* du code, la fonction doit se terminer par un *return arg_sortie* qui sera le résultat visible de l'exécution de la fonction.
- Si la fonction ne renvoie aucune valeur, son type est *void*.
- Si aucun type de sortie n'est précisé, il sera par défaut considéré comme *int*.

— Suivant le type des arguments d'entrées, on aura un passage par valeur ou par adresse, comme on l'expliquera un peu plus tard.
— La liste d'instructions est encadrée par des accolades.

Une fois la fonction définie, elle ne sera exécutée que si elle est appelée. Et pour l'appeler, il faut également la déclarer, comme pour une variable, à l'aide de son prototype. Ainsi, si on a défini une fonction *polycube*, évaluant un polynôme du troisième degré, on pourrait procéder de la façon suivante :

Exemple 1.9.2.

```
#include <stdio.h>
#include <math.h>
int main() {
  double coeff[4] = {4,3,2,1};
  double x;
  double polycube(double a[4], double x);

  scanf(''Entrer un nombre %f'',&x);
  printf(''Apres moulinette: %f\n'',polycube(coeff,x));

  return 0;
}

double polycube(double a[4], double x) {
  double aux;
  aux = a[0]*pow(x,3) + a[1]*pow(x,2) + a[2]*pow(x,1) + a[3];
  return aux;
}
```

Dans notre exemple, on a placé les arguments par valeur, ie que lors de l'appel à la fonction, celle-ci ne voit que le contenu des variables *coeff* et de x, qui est recopié lors de l'appel. Avec cette manière de procéder, la fonction ne peut modifier ses arguments d'entrée. Pour pouvoir modifier la valeur des arguments d'entrée, il faut faire un passage par adresse qui consiste à ne pas donner la valeur de la variable d'entrée qu'on souhaite modifier, mais son adresse. Ainsi, une fonction qui prendrait un nombre en paramètre d'entrée et transformerait ce nombre en son carré serait la suivante :

Exemple 1.9.3.

```
void carre(double * x) {
  *x = (*x)*(*x);
  return;
}
....
double x = 3.1;
carre(&x); // appel
printf(''le carre de 3.1 est %f\n'',x);
....
```

On comprend maintenant un peu mieux l'intérêt des pointeurs.

Le passage par adresse sert également à placer un tableau en paramètre d'une fonction sans pour autant avoir à en copier tous les éléments. Ainsi l'exemple suivant :

Exemple 1.9.4.

```
void carre_tab(int n, double * Tab) {
  int i;
```

```
  for(i=0;i<n;i++)
    Tab[i] *= Tab[i]; /* ou encore: *(Tab+i) = *(Tab+i)* *(Tab+i); */
  return;
}
....
double A[10] = 3.1;
carre_tab(10,A); // appel
....
```

On notera que si on passe par adresse un tableau à une fonction, il faut également passer en paramètre la taille du tableau.

1.9.2 Fonction Récursive

Une fonction a le droit de s'appeler elle-même, à condition qu'elle se *voit*, ie. que son protoptype soit défini en dehors de tout bloc d'instructions. Ainsi, une fonction *factorielle* serait définie de la façon suivante :

Exemple 1.9.5.

```
...
int factorielle(int); /* prototype */

int main(){...}

int factorielle(int n) {
  if (n <= 1)
    return 1;
  else
    return n*factorielle(n-1);
}
```

1.9.3 Variables statiques

Une fonction peut possèder une variable dite statique, qui ne sera pas effacée après chaque appel de la fonction. Ceci permet de réaliser des fonctions qui se servent de variables calculées lors des précédents appels de cette même fonction.

Un exemple intéressant est celui d'un générateur (pseudo)-alératoire de nombres entiers. La génération aléatoire se fait en générale à l'aide d'une suite lineaire basée sur un modulo, de la forme :

$$x_{n+1} = (a * x_n + b) mod\ c$$

avec a, b, c et x_0 des constantes. Ici, on comprend bien que ce qui nous intéressera seront les valeurs de x_n pour des n croissants avec un incrément unitaire. Si on veut en faire une fonction, il faudra donc, à chaque fois qu'on l'appel pour connaître la valeur du prochain nombre généré, se souvenir de la valeur précédente. Pour se faire, on pourra écrire la fonction comme suit (glc pour Générateur Linéaire Congruant) :

Exemple 1.9.6 (Variable statique : exemple de nombres pseudo-aléatoire)**.**

```
#define X0  0
#define A  15
#define B  7
#define C  101
```

```
int glc(){
  static int etat = X0;
  etat = (A*etat + B)%C;
  return etat;
}
```

Cette implémentation fera que la première fois que *glc* est appelée, le programme intialisera la variable *etat* à X0 (0 ici), puis renverra x_1, à savoir 7. La seconde fois que *glc* sera appelée, la variable *etat* aura conservée sa valeur 7 et ne sera plus initialisée a X0. Une suite de 10 appels à cette fonction sortira les valeurs suivantes :

$$7, 11, 71, 62, 28, 23, 49, 35, 27, 8$$

1.9.4 Les arguments du main

La fonction principale d'un programme (le *main*), possède deux arguments. Le premier argument est un entier, le second un tableau de chaîne de caractères. L'interface du *main* est donc :

$$int\ main(int\ n, char * params[])$$

ou encore, plus communément :

$$int\ main(int\ argv, char * *argc)$$

Le premier des arguments représente le nombre d'arguments avec lequel a été appelé le programme, augmenté de un. Le second argument, dont la dimension est *argv* possède comme premier élément le nom sous lequel a été appelé le programme, ensuite, les éléments suivants correspondent aux paramètres qui ont suivi l'appel du programme, mais sous forme de chaînes de caractères. Plus précisément :

Exemple 1.9.7 (Arguments du main)**.**

```
int main(int n, char * params[]){
  ...
  printf("Nombre d'arguments = %d\n",n-1);
  printf("Nom du programme: %s\n",params[0]);
  for(i=1;i<n;i++)
    printf("Argument numero %d = %s\n",i,params[i]);
  ...
}
/* Appel apres compilation: gcc -o mon_prog mon_fichier.c
> ./mon_prog toto 13 a bidule
Nombre d'arguments = 4
Nom du programme: ./mon_prog
Argument numero 1 = toto
Argument numero 2 = 13
Argument numero 3 = a
Argument numero 4 = bidule
```

Sur cet exemple, il faut bien faire attention au fait que durant l'exécution du programme, tous les arguments seront considérés comme des chaînes de caractères. Ainsi, si l'argument "13" doit être utilisé comme l'entier "13", il faudra penser à d'abord convertir *params[2]* en entier (ici avec la fonction *atoi()* par exemple).

De plus, si une fonction doit utiliser des paramètres fournis lors de l'appel du programme, il est toujours plus prudent de s'assurer que ces paramètres sont présents avant de vouloir les utiliser. En pratique, cela se fait en testant le premier argument du *main* pour s'assurer qu'il a la bonne valeur. Par exemple :

Exemple 1.9.8 (Arguments du main, test du nombre de paramètres)**.**

```
int main(int n, char *params[]){
  ...
  if (n < 2) {
    printf("Le programme doit etre appele avec 1 parametre!\n'');
    return 0;}
  ...
}
```

1.10 Bibliothèques Standards du C

1.10.1 Entrées/Sorties en C : stdio.h

Les fonctions d'entrées/sorties en *C* peuvent être relativement *piègeuses*. Heureusement, le traitement des entrées/sorties a été grandement simplifié en *C++*, même si la façon de faire du *C* est toujours valide en *C++*. Nous donnerons ici un rapide aperçu de ces fonctions. Les fonctions d'entrées/sorties se trouvent dans la bibliothèque *stdio.h*, pour *STanDard Input Ouput*. Il s'agit des fonctions (les explications ne seront pas forcément très utiles, mais elles sont illuminées par l'exemple) :

— **int printf(char *format, liste arguments)** : Ecrit dans la sortie standard (l'écran en général) la chaîne de caractères *format* où tous les *%.* (avec .= d,f,lf,s,c...) sont remplacés par la valeur des arguments de la liste (voir l'exemple). Parmi les valeurs possibles de *%.*, on trouve (entre autre) :

Format	Type correspondant
%d	int
%u	unsigned int
%f	float en notation flottante
%e	float en notation exponentielle
%lf	double (long float) en notation flottante
%le	double en notation exponentielle
%c	char
%s	char *
...	...

— **int scanf(char * format, liste arguments*)** : Lit sur l'entrée standard (le clavier en général) la chaîne de caractères *format* et affecte aux variables pointées par la liste d'arguments, ce qui se trouve à la place des *%.* dans *format*.

— **FILE *fopen(char *nom, char *mode)** : ouverture d'un fichier, le type *FILE ** représente un *flux* qui sera ce qu'on manipulera en lieu et place du fichier. La chaîne de caractère *nom* désigne le nom du fichier à ouvrir et *mode* précise si on veut l'ouvrir en lecture (*"r"*), écriture (*"w"*) ou concaténation (*"a"*).

— **int fclose(FILE *fid)** : fermeture du fichier dont le flux est *fid*.

— **int fprintf(FILE *fid, char *format, liste arguments)** : Ecriture, dans le fichier dont le flux est *fid*, de la chaîne de caractères définie par *format* (même utilisation que *printf()*).

— **int fscanf(FILE *fid, char *format, liste arguments*)** : Lecture, dans le fichier dont le flux est *fid*, de la chaîne de caractères définie par *format* et stockage de la lecture dans la liste d'arguments.

Il existe beaucoup d'autres fonctions dans *stdio.h*, que vous apprendrez à utiliser quand vous en aurez besoin (le *man* est votre ami). Un exemple d'utilisation des fonctions présentées est le suivant :

Exemple 1.10.1 (Exemple d'Entrées/Sorties)**.**

```
#include <stdio.h>

int main(){
  FILE * fid;
  int a,b;
  float x,y;
  char truc[20] = "bonjour";

  printf("Entrez %d entier, puis %f flottant: ",1,1.0);
  scanf("%d %f",&a,&x); /* ou scanf("%d",&a);scanf("%f",&x); */

  /* Creation fichier */
  fid = fopen("toto.txt","w");
  fprintf(fid,"Un entier: %d\n",a);
  fprintf(fid,"%f",x);
  fprintf(fid,"%s",truc);
  fclose(fid);
  /* Lecture */
  fid = fopen("toto.txt","r");
  fscanf(fid,"Un entier: %d",&b);
  fscanf(fid,"%f",&y);
  fscanf(fid,"bon%s",truc);
  fclose(fid);

  printf("%d, %f, %s\n",b,y,truc);

  return 0;
}
```

Le résultat de l'exécution de ce programme donnera :

Exemple 1.10.2 (Exécution du programme précédent)**.**

```
> gcc -ansi -Wall -o programme fichier.c
> ./programme
Entrez 1 entier, puis 1.000000 flottant: 2 3 -> ces 2 derniers taper
                                                par l'utilisateur
2, 3.000000, jour -> resultat du dernier printf
> cat toto.txt -> afficher le contenu de toto.txt
Un entier: 4
2.000000 bonjour
```

Attention, lorsque l'on utilise les formats (*%.*), il faut absolument que les types soient conformes. Par exemple, la lecture d'un *%f* avec affectation à un double, va donner un résultat tout à fait inattendu (c'est un *%lf* qu'il faut utiliser).

Notons que pour les lectures (*scanf*, *fscanf*), on utilise l'opérateur unaire & pour donner à la fonction l'adresse de la variable à modifier, car si on ne lui donnait que la valeur actuelle de la variable, aucune modification de cette dernière ne serait possible.

1.10.2 La bibliothèque d'utilitaires : stdlib.h

Cette bibliothèque standart regroupe plusieurs fonctions, dites utilitaires, ayant des buts différents mais toutes très utiles. Vous l'avez déjà rencontré lors de l'introduction de l'allocation dynamique de mémoire (avec *malloc* et *free*). Elle sert également à définir certaines fonctions de conversion de type, de génération aléatoire de nombres, ou encore de gestion de processus. Voici un rapide tour d'horizon de ce que l'on peut y trouver :

- **Conversion de type**
  - *atof* : converti une chaîne de caractère en flottant (utile pour les arguments du main par exemple), interface : double atof (const char * string)
  - *atoi* : converti une chaîne de caractère en entier, interface : int atoi (const char * string)
  - *atol* : converti une chaîne de caractère en entier long (qui est en général un entier classique sur la plupart des machines actuelles), interface : long atol (const char *string)
  - *strtod*, *strtol*, *strtoul* : autres fonctions de conversions d'une chaîne de caractères mais un peu plus évoluées puisqu'elles permettent de spécifier la fin de la chaîne de caractères à convertir. Ceci permet de ne convertir qu'une partie d'une chaîne de caractères. Les interfaces sont les suivantes : double strtod(const char *nptr, char **endptr), long int strtol(const char *nptr, char **endptr, int base), unsigned long int strtoul(const char *nptr, char **endptr, int base). L'entier *base* de *strtol* et *strtoul* permet de convertir des chaînes de caractères en la considérant comme étant dans une *base* donnée (qui doit être entre 2 et 36 ou encore 0 pour la base par défaut, ie 10).
- **Génération aléatoire** La génération de nombre (pseudo)-aléatoire est gèrée par les fonctions *rand* et *srand*.
  - *rand* à pour interface *int rand(void)*, ie qu'elle renvoie un entier et ne prend aucun argument. L'entier renvoyé est compris entre 0 et la constante *RAND_MAX*. Cette fonction ne renvoie qu'une suite de valeur pseudo-aléatoire, ie qu'on aura une répétition de suite de valeurs si on attend assez longtemps. De plus, si on n'initialise pas manuellement la graine du générateur, elle est toujours égale a 0 ce qui a pour conséquence que le programme suivant génère toujours la même suite de valeurs aléatoires d'un appel à l'autre :
    Exemple 1.10.3 (Appel à rand sans initialisation de la graine)**.**
    ```
    #include <stdlib.h>
    #include <stdio.h>
    int main(){
      int i;
      printf("Suite aleatoire? : ");
      for (i=1;i<10;i++){
        printf(" %d,",rand());
      }
      printf(" %d\n",rand());
      return 0;
    }
    ```
    Si on appelle 2 fois de suite l'exécutable provenant de ce code, on obtient 2 fois la même suite de 10 nombres.
  - *srand* à pour interface *void srand(unsigned int seed)* et permet d'initialiser la graine du générateur pseudo-aléatoire *rand* avec *seed*. Ceci permet de palier au problème de prédictabilité du *rand*. En général, on combine l'appel à cette procédure avec la fonction *time* qui renvoie l'heure de la machine sur laquelle est exécuté le programme. Dans l'exemple précédent, pour ne pas avoir 2 fois de suite la même suite de nombre, on introduira simplement un appel à *srand(time(NULL))* avant le premier appel a *rand*.

La génération de nombres pseudo aléatoires est un sujet compliqué. On notera que certaines implémentation de *rand* peuvent avoir le défaut que les bits de poids faible des nombres générés ne sont pas aussi aléatoire que les bits de poids forts (en général *RAND_MAX* est assez important). Ceci n'est en principe plus le cas sur les implémentations récentes de *rand*. Pour générer des nombres entre disons, a et b, on fait des divisions du résultat de *rand* par *RAND_MAX* ou encore des modulos.

- **Allocation de mémoire** Vous connaissez déjà en partie ces fonctions, il s'agit de :
  - *void * malloc(size_t size)* : réalise la réservation en mémoire de *size* octets contigus et renvoie un pointeur sur le premier octet (renvoie le pointeur vide NULL en cas d'échec).
  - *void * calloc(size_t nobj, size_t size)* : réalise la réservation de *nobj*size* octets contigus et renvoie un pointeur sur le premier octet réservé.
  - *void * realloc(void *p,size_t size)* : réalise la réservation de *size* octets contigus en mémoire et y copie les données pointées par le pointeur p. Cette fonction est surtout utile pour augmenter la taille d'un tableau sans avoir a recopier à la main les éléments déjà existant. Attention, ceci n'est pas très efficace comme méthode du point de vue du temps d'exécution.
  - *void free(void *p)* : libère les octets pointées par le pointeur p (pour éviter les fuites de mémoire).
- **Contrôles de processus** Dans ces fonctions, on notera :
  - *void abort(void)* : cette procédure cause l'arrêt du programme de façon anormale, à moins que le signal *SIGABRT* soit attrapé par un *catch* (voir la gestion d'exception dans la partie *C++*). Si un programme s'arrête sur un *abort*, alors tous les flux ouverts sont fermés.
  - *void exit(int signal)* : provoque l'arrêt du programme de façon normale (?).
  - *int system(const char *command)* : exécute la chaîne de caractère comme une commande shell (ou ligne de commande windows). La valeur de retour est -1 si l'exécution de la commande à levée une erreur, sinon cette valeur est celle retournée par la commande.
  - *char *getenv(const char *name)* : retourne la variable d'environnement correspond à la chaîne de caractère *name* (si celle-ci existe dans l'environnement bien entendu). Ceci est, tout comme *system*, une commande pour intéragir avec le système d'exploitation de la machine sur laquelle est exécutée le programme.
- **Maths, recherche et tri**
  - *void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *))* : recherche dans un tableau de *nmenb* objets, le premier étant pointé par *base*, le premier qui correspond à l'objet pointé par *key*. La taille de chacun des éléments du tableau est de *size*. La fonction *compar* est une fonction de comparaison (le tableau pointé par *base* doit être rangé dans l'ordre croissant suivant cette relation de comparaison). C'est cette relation qui est utilisée pour dire si l'objet est présent ou non.
  - *void qsort(void *base, size_t nmemb, size_t size, int(*compar)(const void *, const void *))* : trie un tableau de *nmemb* éléments dont le premier est pointé par *base* et dont chacun à une taille de *size*. La fonction *compar* sert de relation de comparaison pour le tri.
  - *div_t div(int numerator, int denominator)* effectue la division entière entre *numerator* et *denominator*, et retourne une structure de type *div_t* contenant le quotient et le reste de la division.
  - *int abs(int j)* retourne la valeur absolue de l'entier j. Il existe également une fonction *labs* et *llabs* pour les *long int* et *long long*. On notera que pour les fonctions valeurs absolues sur des flottants, la bibliothèque à utiliser est la bibliothèque mathématiques *math.h*.

1.10.3 Chaînes de Caratères : string.h

Cette bibliothèque contient non seulement les fonctions de manipulation de chaînes de caractères *C* (ie *char **) mais également certaines définitions de macros, de constantes et de fonctions de mani-

pulation de la mémoire.

Voici un pot-pourri de ce qu'on peut y trouver :

— **NULL** : la macro représentant un pointeur qui pointe sur une adresse mémoire non valide.
— **size_t** : un type d'entier non signé qui est le type de retour de la fonction *sizeof*.
— **char *strcpy(char *dest, const char *src)** : copie la chaîne de caractères pointée par *src* dans la chaîne de caractères pointée par *dest*.
— **char *strncpy(char *dest, const char *src, size_t n)** : copie les n premiers caractères de la chaîne de caractères pointée par *src* vers la chaîne de carcatères pointée par *dest*. On notera qu'il faudra donc rajouter le caractère de fin de chaîne pour rendre la chaîne pointée par *dest* valide après l'exécution de *strncpy*.
— **char *strcat(char *dest, const char *src)** : concatène à la suite de la chaîne de caractères pointée par *dest*, celle pointée par *src*, le caractère de fin de chaîne de *dest* est écrasée par le premier caractère de *src*.
— **char *strncat(char *dest, const char *src, size_t n)** : comme *strcat* mais ne concatène que les n premiers caractères de *src* à *dest*. La chaîne résultante sera valide et se terminera bien par le caractère de fin de chaîne (qui sera rajouté).
— **int strcmp(const char *s1, const char *s2)** : compare les chaînes de caractères pointée par *s1* et *s2*. Renvoie un entier négatif si $s1 < s2$, nul si $s1 = s2$, positif si $s1 > s2$.
— **int strncmp(const char *s1, const char *s2, size_t n)** : idem que *strcmp* mais ne compare que les n premiers caractères des 2 chaînes.
— **char *strchr(const char *s, int c)** : cherche le caractère c (un int devient facilement un char) dans la chaîne de caractère pointée par *s*. Renvoie un pointeur sur le premier caractère trouvé, et *NULL* si le caractère n'est pas trouvé. La fonction *strrchr* à la même interface mais renvoie la dernière occurence du caractère. La fonction *strchrnul* effectue la même chose que *strchr* à l'exception qu'en cas d'échec le pointeur retourné pointe vers le caractère de fin de chaîne de *s*.
— **size_t strlen(const char *s)** : renvoie la longueur de la chaîne de caractères pointée par *s*, le caractère de fin de chaîne ne compte pas.
— **char *strpbrk(const char *s, const char *accept)** : renvoie la première occurence de n'importe lequel des caractères de la chaîne pointée par *accept*, dans la chaîne *s*.
— **void *memcpy(void *dest, const void *src, size_t n)** : copie n octets à partir de la zone mémoire pointée par *src* vers la zone mémoire pointée par *dest*. Les 2 zones mémoires ne doivent pas avoir d'intersection commune.
— **void *memmove(void *dest, const void *src, size_t n)** : idem que *memcpy* mais autorise les zones mémoires à s'intersecter. Cependant cette fonction est moins performante que *memcpy*.
— **int memcmp(const void *s1, const void *s2, size_t n)** : compare les n premiers octets des zones mémoires pointées par *s1* et *s2*.

Bien d'autres fonctions sont disponibles dans cette bibliothèque, mais le *C++* ayant une meilleure gestion des chaînes de caractères que le *C*, il n'est pas nécessaire de trop approfondir.

1.10.4 Mathématiques : math.h

La bibliothèque mathématique, dont vous aurez très certainement besoin, comporte un grand nombre de fonctions mathématiques de base. On notera que lors de la compilation d'un code utilisant la bibliothèque mathématique, il faut en général faire le lien explicitement grâce à l'option de compilation *-lm* (par exemple : gcc truc.o bidule.o -lm -o machin). Voici quelques fonctions de cette bibliothèque :

(1) Fonctions trigonométriques
— *double cos(double x)* : cosinus
— *double sin(double x)* : sinus
— *double tan(double x)* : tangente

- *double acos(double x)* : arcosinus
- *double asin(double x)* : arcsinus
- *double atan(double x)* : arctangente
- *double atan2(double y, double x)* : arctangente de y/x mais utilise les signes de y et x pour déterminer le bon quadrant

(2) Fonctions hyperboliques
- *double cosh(double x)* : cosinus hyperbolique
- *double sinh(double x)* : sinus hyperbolique
- *double tanh(double x)* : tangente hyperbolique

(3) Fonctions exponentielle, logarithmique et puissance
- *double exp(double x)* : exponentielle
- *double log(double x)* : logarithme népèrien
- *double log10(double x)* : logarithme en base 10
- *double pow(double x, double y)* : calcule x à la puissance y
- *double sqrt(double x)* : racine carrée

(4) Fonctions d'approximations, valeur absolue
- *double ceil(double x)* : entier le plus proche
- *double floor(double x)* : entier inférieur le plus proche (en gros : part entière)
- *double fabs(double x)* : valeur absolue d'un flottant
- *double fmod(double x, double y)* : reste de la division 'entière' entre x et y (fmod = x - n*y, n entier)

Chapitre 2

Conception Objet et langage C++

2.1 Le langage C++, une extension du C ?

2.1.1 Historiette

Le langage C++ a deux principaux ancêtres. Le premier, *Simula*, a été conçu en 1967 et est le premier langage introduisant les concepts de la programmation objet. Il a été développé par une équipe de chercheurs Norvégiens et permettait de traiter des problèmes de simulation. On y trouve déjà les notions de classe, d'héritage et de masquage de l'information. Il n'a cependant jamais vraiment percé car il venait probablement un peu tôt. Le second ancêtre est bien entendu le langage C auquel vous avez déjà été brièvement introduit dans la première partie de ce document. Le C++ reprend tout ou presque de la syntaxe du C. Il en garde les points forts, en corrige certains points faibles (mais si, il y en a quelques uns) et ajoute toute une panoplie de techniques pour la programmation objet. Le C++ a connu une première forme stable vers 1983, soit environ 10 ans après la naissance du C, et a très rapidement connu un grand succès dans le monde industriel. D'autres langages, comme le Java, se sont inspirés de la syntaxe (objet) du C++. De part son ancêtre fonctionnel (le C), le C++ n'est pas un langage objet pur mais un langage hybride. Nous allons donc dans un premier temps nous intéresser à la partie fonctionnelle du C++, celle directement dérivée du C. Le reste de ce chapitre est donc dédié aux différences entre le C et le C++.

2.1.2 Structure d'un programme

Encore plus qu'en C, il est fortement conseillé de séparer les définitions de types, constantes, déclarations de fonctions de l'implémentation du programme. Tout comme en C, on sépare alors le code en un (ou des) fichier interface, d'extension .hh, .H ou .h ; noter qu'à terme il était prévu que ces fichiers perdent leur extension, d'où des bibliothèques standards n'en ayant plus. par exemple, la nouvelle bibliothèque standard d'entrée/sortie, *iostream*, s'inclut désormais par la syntaxe *#include <iostream>* (< ... > et non "..." puisqu'en général il ne s'agit pas de bibliothèques physiquement présentes dans le répertoire de compilation). Les bibliothèques déjà disponibles en C peuvent toujours être utilisées mais elles ont elles aussi perdu leur extension et sont préfixées par le caractère 'c' (ex : *cstdio*). Pour inclure, dans un fichier source, un fichier interface ayant une extension, on utilise la syntaxe *#include "NomAvecExtension.Extension"*. Par exemple :

Exemple 2.1.1.

```
#include <iostream> /* sans extension */
#include "point.h" /* un fichier interface avec extension, present
                      dans le repertoire courant */
```

Un code C étant compatible C++, il devrait (dépend du compilateur utilisé) encore être possible

d'utiliser les bibliothèques standards du C sous leur forme originelle, ie avec l'extension '.h' et sans le préfixe.

Les fichiers sources, contenant l'implémentation, portent traditionnellement les extensions .cc, .cpp ou .C. Les codes C étant compatibles, on gardera l'extension .c pour les codes écrits purement en C (en particulier les codes uniquement fonctionnels). On notera que certains compilateurs peuvent avoir tendance à se servir des extensions pour 'deviner' le contenu du fichier (c'est d'ailleurs le cas des systèmes d'exploitation). De plus, **le type de retour de main doit obligatoirement être *int*, et rien d'autre**.

Pour compiler un code C++, il faut bien entendu un compilateur C++ et non pas un compilateur C. Dans tout le document on choisira par défaut le compilateur C++ de GNU, appelé *g++* et ayant à peu près les mêmes options que *gcc*.

2.1.3 Un nouveau type

Tous les types élémentaires du C sont disponibles en C++ (int, float, double, char,...), mais il en introduit un nouveau pour représenter les booléens. Le nom de ce type est *bool* et une variable booléenne peut prendre la valeur *true* ou la valeur *false*. Bien entendu, il est toujours possible d'utiliser un entier pour représenter un booléen. Voici un exemple :

Exemple 2.1.2 (Exemple d'utilisation du type *bool*)**.**
```
bool b;
double x = 1.5;
double y = 2.3;
b = (x > y); // => b = false
```

2.1.4 Commentaires

En plus des commentaires C encadrés par */** et ***/**, le C++ offre la possibilité de faire des commentaires n'excédant pas une ligne, et ce, en faisant précédé le commentaire par le symbole *'//'* :

Exemple 2.1.3.
```
i++; // commentaire court (C++)
j++; /* commentaire long qui
        prend plus d'une ligne */
```

On notera qu'en général les compilateurs C acceptent déjà les commentaires courts.

2.1.5 Typage

Le contrôle de type en C++ est beaucoup moins permissif qu'en C. En particulier, il est obligatoire de déclarer une fonction (avec son prototype) avant de l'utiliser. En C on pouvait éventuellement s'en passer (mais ça restait mal !) et le compilateur assignait par défaut le type de retour *int*. En C++, l'utilisation d'une fonction non déclarée provoquera une erreur de compilation. Par exemple :

Exemple 2.1.4.
```
int main() {
  int a=3, b=1, c;
  // int g(int, int); ok si on decommentait
  c = g(a,b); // erreur a la compilation
  ...}

int g (int x, int y)
{ return (x*x + y*y * x*y);}
```

2.1.6 Déclarations

Dorénavant, on peut déclarer les variables du corps d'une fonction un peu partout. On n'est en particulier plus obligé de faire toutes les déclarations au début. On doit cependant toujours déclarer une variable avant de l'utiliser, sous peine de générer une erreur à la compilation :

Exemple 2.1.5.
```
#include <cstdio>
int main() {
  int n;
  printf("taille?: ");
  scanf("%d",&n);
  int tab[n];
  for (int i=0;i<n;i++) {...}
  return 0;}
```

On déconseille pourtant d'utiliser cette fonctionnalité quand on peut s'en passer, car cela est loin d'améliorer la lisibilité du code. Cette liberté est surtout appréciable pour éviter les allocations dynamiques, comme dans notre exemple.

2.1.7 Constantes

En C, la notion de constantes, déclarées grâce au mot-clef *const* n'était pas très utile. La raison principale était que le compilateur C traite ce genre de variable comme une variable normale, dans le sens où il alloue de la mémoire à la variable et ne connaît donc pas la valeur de la variable lors de la compilation (car l'allocation de mémoire ne se déroule qu'à l'exécution). Du coup, on ne peut pas, en C, utiliser ce genre de variable pour définir une autre structure, comme un tableau et on en est réduit à utiliser le *#define* du préprocesseur. En *C++*, le mot-clef *const* devient beaucoup plus pratique et remplace avantageusement le *#define* tout en permettant de sécuriser le code. Il permet de plus de tracer une frontière nette entre ce qui change et ce qui ne change pas (en tout cas ce qui ne doit pas changer). Ainsi, une fois une variable déclarée comme constante, le compilateur interdira toute opération sur cette constante qui pourrait résulter en la modification de cette dernière.

En C++, une variable globale déclarée avec la commande *const* a une portée limitée au fichier source dans lequel elle est déclarée. En C, on pouvait l'exporter grâce à la commande *extern*, ce qui n'est plus le cas en C++. Par contre, en C++ une telle constante peut être utilisée dans une expression constante alors qu'en C on devait faire appel à une macro *#define*. Ceci permet d'utiliser une constante déclarée par *const* dans la définition d'un tableau en ayant un contrôle de type (on rappelle qu'avec le *#define*, il ne pouvait y avoir de contrôle de type).

Exemple 2.1.6.
```
const int taille = 10;
int main() {
  int tableau [taille]; // bon en C++, erreur en C
  ...}
```

On notera qu'une constante doit être initialisée à la déclaration sans quoi on se retrouve avec une erreur à la compilation.

Le compilateur s'assurant qu'une constante le reste, il devient interdit d'appeler une fonction sur une constante à moins d'être certain que la fonction ne modifie pas son paramètre d'entrée. Ceci est bien entendu le cas lors de passage par valeur puisque dans ce cas, il est tout à fait impossible à la fonction de modifier son argument. Cependant, le passage par adresse étant parfois utile pour d'autres chose que pour modifier un paramètre d'entrée d'une fonction (pour les gros tableaux par exemple), il devient nécessaire de pouvoir signaler au compilateur, lors de la déclaration d'une fonction, que

cette dernière, bien qu'ayant la possibilité de modifier ses paramètres passés par adresse (ou référence comme on le verra sous peu), ne le fait pas. Ceci se fait encore une fois grâce au mot-clef *const* mais qu'on rajoute maintenant à l'interface de la fonction. Voici un exemple succint :

Exemple 2.1.7 (Exemple de paramètres constants d'une fonction)**.**

```
int fct1(int *tab, int pos); // retourne tab[pos]
int fct2(const int* tab,int pos); // idem que fct1

int main(){
  const int T[5] = {1,2,3,4,5};
  int a, b;
  a = fct1(T,0); // cet appel provoque une erreur de compilation
  b = fct2(T,0); // celui-ci n'en provoque pas
  ...
```

Bien entendu, si vous déclarer dans l'interface d'une fonction que vous ne modifierez pas un des arguments, il ne faut pas le modifier, sinon le compilateur protestera.

2.1.8 Fonctions et Surcharge

Les fonctions doivent avoir un type de retour explicité, ce type peut être *void*. Une fonction sans argument se déclare et se définit en fournissant une liste d'arguments vide.

Exemple 2.1.8.

```
double fct1(); // et non pas double fct1(void);
void fct2();   // et non pas fct2();
```

Et rappelons encore une fois que le type de retour du point d'entrée du programme (*main*) est obligatoirement *int*.

Une caractéristique intéressante du C++ (et qui le deviendra encore plus dans sa couche objet) est la surcharge ou surdéfinition de fonctions. En effet, il est permis de définir plusieurs fonctions portant le même nom, à condition que leurs interfaces diffèrent. Lors de la compilation, le compilateur choisira quelle fonction utiliser en se basant sur le contexte dans lequel la fonction est utilisée. Ainsi, une fonction division :

Exemple 2.1.9.

```
int main(){
  int a=3, b=2, c;
  float x, y, z;
  int division (int, int);
  float division (float, float);
  c = division(a,b); // premiere fonction appelee: donc division entiere
  x = a; y = b;
  z = division(x,y); // seconde fonction appelee
...}


int division (int a, int b) {...}
float division (float x, float y) {...}
```

Une telle surcharge était bien entendu déjà disponible sur des opérations telles que +, -, /, * mais maintenant l'utilisateur peut en rajouter lui-même. Un gros avantage de la surcharge est qu'elle permet de définir des fonctions ayant des paramètres par défaut. Imaginons qu'on décide de créer une

fonction de tri d'un tableau avec des options permettant de trier les valeurs par ordre croissant ou décroissant et aussi d'appliquer une fonction affine aux éléments du tableau. La plupart du temps on voudra utiliser cette fonction pour trier par ordre croissant un tableau sans appliquer de fonction affine (enfin juste l'identité). On définira et utilisera la fonction comme dans l'exemple.

Exemple 2.1.10 (Définition de paramètres par défaut)**.**

```
const int taille = 10;
/* protoptype de la fonction */
void trier(float *, int, int = 1, float = 1, float = 0);

int main(){
  float tableau[taille];
  int sens = -1;
  float c1 = 3.4, c2 = -4.3;
  ... initialisation du tableau ...
  trier(tableau,taille); // appel avec sens=1, a=1, b=0
  trier(tableau,taille,sens); // appel avec a=1, b=0
  ...}

/* definition de la fonction */
void trier(float *tab, int taille, int sens, float a, float b);
{... corps de la fonction ...}
```

Lors de l'utilisation de cette fonctionnalité, on définit en réalité plusieurs fonctions (4 dans notre cas) ayant chacunes un nombre d'arguments différent. En particulier, on sera obligé de fournir la valeur de 'sens' si on veut fournir celle de 'a' et de 'b'.

On notera que les paramètres par défaut ne sont spécifiés qu'une seule et unique fois lors de la définition de la fonction ou de son prototype. Il est en général préférable de déclarer les paramètres par défaut dans le prototype de la fonction car ce sera lui qui sera en général connu des instructions qui utiliseront la fonction.

2.1.9 Fonction template

La possibilité de surdéfinir une fonction amène, en plus des paramètres par défaut, la notion de généricité. En effet, il peut arriver assez souvent que la surdéfinition d'une fonction ne soit là que pour prendre en compte des arguments de types différents mais que le corps des différentes fonctions ainsi définies soient, eux, exactement les mêmes. C'est en général le cas des fonctions faisant intervenir des opérateurs arithmétiques ou de comparaison. Dans ce cas, on utilisera le mot clé *template* pour définir des types abstraits que l'on utilisera dans la déclaration et définition de la fonction. Par exemple, pour la fonction qu'on définira dans le fichier interface comme suit :

Exemple 2.1.11 (Fonction template, fichier interface)**.**

```
template <class T>
T mini (T a, T b)
{ return ((a<b)? a:b);}
```

Il est important de noter que :

— chaque type de données utilisées dans la déclaration doit impérativement être utilisé dans la définition,

— les fonctions génériques doivent absolument être définies dans le fichier interface (en fait les types sont décidés à la compilation). Donc en particulier il n'y a pas de séparation entre prototype et définition.

Pour utiliser la fonction template, comme elle aura déjà été déclarée dans le fichier interface (qui sera inclus dans le fichier source), on n'a plus qu'à l'appeler et le compilateur déterminera le type effectif T suivant le contexte. Cette première notion de généricité utilise le mot clé *class* qui désigne une classe, ie un type de données comme on le verra en détail dans la troisième partie de ce chapitre.

2.1.10 Fonction inline

Lorsque l'on a affaire à une fonction dont le corps est particulièrement petit, comme par exemple la fonction *mini* dont la définition ne comporte qu'une seule instruction, il peut être avantageux de remplacer chaque appel de la fonction par son corps. Ceci permet de gagner un peu en efficacité d'exécution. A cette fin, le langage permet de définir des fonctions dites *inline*, un peu dans le même esprit que le *#define* pour les variables. Pour ce faire, on définit, dans le fichier interface, la fonction en employant le mot clé *inline*. Par exemple :

Exemple 2.1.12 (Fonction inline, fichier interface)**.**
```
inline int max (int x, int y)
 { return ((x<y)? y:x);}
```

Lors de la précompilation, le compilateur tentera de faire le remplacement des appels à la fonction par son corps ; cependant c'est le compilateur qui décidera si oui ou non ce remplacement est profitable. Donc la définition d'une fonction comme *inline* n'est qu'une indication et pas un ordre, contrairement à la définition utilisant le *#define* du préprocesseur.

2.1.11 Pointeur & Référence

La manipulation de pointeur est toujours disponible en C++, mais on y a adjoint un nouveau genre de variable : les références. Une référence est une sorte d'alias sur une variable. Une opération sur une référence est en fait une opération sur la variable référencée. On a donc toujours l'idée de l'adresse d'une variable, comme pour le pointeur, mais la manipulation d'une référence se fait plus naturellement que celle d'un pointeur. On déclare une référence de la façon suivante :

$$\text{Type \&NomRef} = \text{VariableAReferencer} ;$$

Cette déclaration est la seule possible, en particulier, une référence ne pouvant être vide, il faut obligatoirement l'initialiser à la déclaration. C'est dans ce type de cas que la possibilité de faire des déclarations où bon nous semble devient intéressante. On donne ici un exemple de fonctions échangeant 2 variables entières en utilisant des pointeurs ou des références.

Exemple 2.1.13 (Pointeurs et Références : Swap)**.**
```
/* Declaration des interfaces */
void swapC(int *,int *); // avec pointeurs
void swapCpp(int &, int &); // avec references

/* Point d'entree */
int main()
{ int m=1, n=2;
  swapC(&m,&n);
  swapCpp(m,n);
  return 0;
}

/* Avec Pointeurs */
```

```
void swapC(int *i,int *j)
{ int tmp = *i;
  *i = *j;
  *j = tmp;}

/* Avec References */
void swapCpp(int &i, int &j)
{ int tmp = i;
  i = j;
  j = tmp;}
```

On notera que les 2 versions de 'swap' sont valides en C++ mais que celle utilisant les références est plus simple à écrire et à appeler. De plus, notons qu'une référence est fixée, ie. qu'elle ne peut référencer qu'une variable (donnée à la déclaration) et ne peut plus changer après (contrairement à un pointeur).

2.1.12 Allocation dynamique de mémoire

En plus des références, le C++ allège la façon de faire des allocations dynamiques de mémoire. Plus besoin d'utiliser les fonctions *malloc*, *sizeof* et *free*. Maintenant, une allocation se fait grâce à la commande *new* et une libération de mémoire se fait grâce à *delete*. Il n'est en outre plus nécessaire de spécifier le nombre d'octets à réserver, le compilateur le calculera suivant le contexte d'appel de *new* et son argument (optionnel). Par contre, il est interdit de mélanger les commandes, ie on ne peut libérer avec *delete* une case mémoire allouée par un *malloc* et on ne peut pas non plus utiliser *free* pour libérer une variable provenant de *new*. Un exemple d'utilisation de *new* et *delete* :

Exemple 2.1.14 (Allocation dynamique de mémoire)**.**

```
/* En C */
int * pEntier;
float * tab;
...
pEntier = (int *) malloc(sizeof(int));
tab = (float *) malloc(50*sizeof(float));
...
free(pEntier);
free(tab);
...

/* En C++ */
int * pEntier;
float * tab;
...
pEntier = new int;
tab = new float[50];
...
delete pEntier;
delete[] tab;
...
```

On voit que la syntaxe de *new* est *Pointeur = new type[taille]*. On notera qu'encore une fois, c'est dans ce genre de cas que la déclaration au petit bonheur peut être avantageuse.

Par contre, malgré sa simplicité l'allocation dynamique par *new* comporte un inconvénient. En effet, elle ne renvoie rien en cas d'échec (pas de valeur *NULL*) mais provoque une erreur (renvoie une *exception*). Alors qu'avec *malloc*, on pouvait tester le retour et agir en conséquence. On peut cependant récupérer les *exceptions* avec *try* et *catch*, ce qui est surtout utile en cas de "grosse" allocation de mémoire.

2.1.13 Entrée/Sortie standard

En C, l'utilisation des fonctions d'entrées/sorties demandait une bonne connaissance du format des données manipulées. Ceci pouvant être quelque peu compliqué et même induire quelques erreurs (comme lire un *double* avec *%f* au lieu de *%lf*). En C++, la bibliothèque d'entrée/sortie *iostream* rend l'utilisation des entrées/sorties clavier et écran beaucoup plus facile. Les opérations standards d'entrées/sorties sont fournies par trois flots (streams), désignées par les variables :

cin pour le flot d'entrée standard (le clavier en général)
cout pour le flot de sortie standard (l'écran en général)
cerr pour le flot standard des messages d'erreurs.

Pour être exact, ce sont *std : :cin*, *std : :cout* et *std : :cerr* qui désignent ces flots standards. Pour les manipuler, les opérateurs << et >> sont utilisés, comme le montre cet exemple.

Exemple 2.1.15 (Exemple d'utilisation de iostream)**.**

```
#include <string> // pour definir le type 'string'
#include <iostream>
using namespace std; // prefixe cin, cout, cerr par 'std::'
                     // prefixe egalement string
int main()
{ string nom;
  cout << "Aloha, quel est votre nom? ";
  cin >> nom;
  cout << "Enchante " << nom << endl; //ou "\n"
  return 1;
}
```

On notera que le passage à la ligne peut se faire indépendamment avec le "\n" du C ou avec *endl*.

On notera également l'emploi de la bibliothèque *string* qui permet de manipuler des chaînes de caractères plus facilement qu'avec un type *char ** ou *char[taille]*.

Nous n'irons pas plus avant dans la description des fonctionnalités si ce n'est pour mentionner que les bibliothèques *ofstream*, *ifstream* et *fstream* permettent respectivement d'écrire, lire et lire/écrire dans un fichier. La raison pour laquelle nous ne détaillons pas l'emploi de ces bibliothèques est qu'il s'agit en fait de classes dont nous ne présentons l'utilisation qu'à partir de la section 2.5.

2.2 Conception Orientée Objet

2.2.1 Généralités

Dans les années 80, les logiciels sont devenus de plus en plus complexes et donc difficiles à développer, maintenir et faire évoluer. D'où le besoin de logiciel de qualité et donc de méthode de développement propre. De plus, pour limiter la phase de développement, il est intéressant de pouvoir réutiliser des "bouts" de codes existants. La conception objet tente de répondre à ces préoccupations en insistant sur les notions suivantes :

— robustesse, en cachant le plus possible l'implémentation du logiciel et en limitant strictement ce que l'utilisateur a le droit de faire et de voir,

— modularité, qui rend le code plus flexible et donc plus facilement évolutif,
— réutilisabilité, afin de ne pas repartir de zéro lors de futurs développements.

2.2.2 La Modularité

Le principe d'un code modulaire est là pour éviter de produire du code monolithique qui rendrait tout changement difficilement réalisable. En effet, une telle architecture donne souvent lieu à des relations compliquées entre les composants du code et donc changer un composant implique de changer tous les autres. La modularité tente de créer une architecture logicielle flexible en rendant les composants de l'architecture les plus indépendants possibles pour isoler les changements éventuels à apporter. Pour atteindre cette indépendance, on tente de limiter et cadrer les relations que les composants entretiennent entre eux. De plus, dans chacun de ces composants, que l'on appellera modules, on concentrera la connaissance des caractéristiques et des méthodes lui étant relatives.

La conception des modules peut se faire suivant 2 familles d'approches. La première est l'approche **descendante**, consistant à partir du problème que l'on veut résoudre et à le découper en sous-problèmes jusqu'à ce que ces derniers soient triviaux. La seconde approche est l'**ascendante** qui consiste à partir des bouts de codes (des solutions à des problèmes existants) déjà disponibles et de les utiliser pour construire une solution au problème courant. Bien entendu, ces deux approches sont en général utilisées en conjonction.

Quelque soit l'approche utilisée, les modules se doivent de répondre à certains critères de qualité. Il doivent être, entre autre :

— compréhensibles, ie. clairs, logiques et ne communiquer qu'avec peu d'autres modules,
— continus, ie. qu'une petite modification des spécifications ne doit entraîner le changement que d'un petit nombre de modules,
— protégés, ie. que l'action d'un module doit être le plus possible restreinte à lui seul. Ceci permet d'isoler les erreurs.

Ces critères de qualité nous amènent à considérer des modules ayant des interfaces limitées et explicites. De plus, les communications entre modules doivent également être limitées et les informations contenues dans les modules doivent être masquées. En particulier, le masquage de l'information interdit (déconseille fortement) l'utilisation de variables globales.

2.2.3 La Réutilisabilité

Le principe de réutilisabilité est connu de longue date et est derrière l'idée de bibliothèque, ie de ne pas repartir de zéro mais de s'appuyer sur la résolution de problèmes passés. L'idée de bibliothèque est bonne mais en général ne donne lieu qu'à des solutions peu flexibles et difficilement adaptables. La conception objet cherche à formaliser cette notion de réutilisabilité en définissant un bon module comme un module réutilisable. Un tel module doit pouvoir manipuler plusieurs types différents, mais aussi offrir des fonctions à l'utilisateur sans que celui-ci n'ait à connaître son implémentation. Les opérations communes à un groupe de modules doivent pouvoir porter le même nom pour agir sur des types différents. C'est l'idée du polymorphisme.

Les techniques permettant de répondre aux critères de la réutilisabilité sont la surcharge d'opérateurs et la généricité (module paramétré par le type qu'il manipule).

2.2.4 Principes de la Conception Objet

Pour faciliter la conception suivant les critères énoncés précédemment, il est nécessaire de repenser la manière dont on conçoit un programme. Un programme est un ensemble d'algorithmes et de structures de données. La manière intuitive de réaliser un programme est de l'orienter vers les traitements/algorithmes qu'il doit effectuer. Avec cette approche, on considère les tâches que le programme doit accomplir, puis on les décompose en tâches élémentaires ou déjà écrites. Le problème est qu'alors,

les modules découlent des tâches à accomplir, ce qui les rend peu flexibles et réutilisables pour d'autres tâches. Une architecture née d'une conception orientée traitements devient très rapidement lourde et entortillée. De plus, les traitements sont par essence beaucoup moins stables que les données et une évolution du programme impliquera donc une complète refonte de l'architecture. Cependant, cette approche a tout de même l'avantage d'être intuitive, en général facilement et rapidement développable et donc bien adaptée à des applications de taille réduite.

A l'opposé de la conception orientée vers les traitements, se trouve la conception orientée vers les données (les objets). Elle consiste à se dire la chose suivante : *"Ne commencez pas par demander ce que fait le système, demandez à* **qui** *il le fait"*[1]. On va alors organiser le programme autour des objets qu'il manipule, car ils sont plus stables que les traitements. Une fois que ces objets seront définis, le concepteur n'aura plus qu'à écrire les traitements devant s'effectuer sur ces objets. Au départ, cette approche est très contre-intuitive mais permet de satisfaire naturellement aux exigences de qualité de conception de logiciel.

La grande question est bien entendu de comment choisir les objets. Il n'y a pas de réponses universelles mais en général on peut tenter de se raccrocher aux objets physiques ou abstraits qui nous entourent. Par exemple, pour simuler la logistique d'une école, on aura besoin d'objets tels que des personnes (avec des catégories : notion d'héritage), des moyens physiques (locaux, fournitures,...) et je ne sais quoi d'autre. Les objets vont représenter toutes les entités, des plus simples aux plus complexes, et sont un peu comparables aux structures (*struct*) du langage C. Ils en sont en fait une extension car ils ont des propriétés supplémentaires.

Les objets suivront un modèle que l'on appellera **classe**. Une classe sera décrite par ses attributs (ses champs ; par exemple, pour une personne, ses attributs pourront être : sexe, âge, taille, ...) mais aussi par les fonctions (méthodes) qu'elle fournit. Par exemple, la classe Date, déclarée comme le montre l'exemple :

Exemple 2.2.1 (Exemple de la classe Date)**.**

```
class Date {
  int jour, mois, annee;
  ...
  void ajoute_jour(int n);
  bool bissextile();
  void affiche();
};
```

Les attributs et les méthodes d'une classe seront appelés **membres** de la classe. Une classe sera un nouveau type dont les occurrences/instances seront les objets.

Voyons maintenant plus en détail cette notion de classe et la façon dont on la déclare en C++.

2.3 Les Classes

2.3.1 Définition

Une classe peut être vue comme un type de données complexe défini par l'utilisateur : une définition. C'est un enrichissement très important de la notion de structure (*struct*) du C et c'est la base de la programmation orientée objet.

Si une classe est avant tout un nouveau type, c'est également une collection de méthodes s'appliquant sur les instances (les objets) de la classe. Dans toute la suite, nous utiliserons comme exemple la définition de la classe **complexe** (qui en réalité existe déjà et se nomme **complex**). Une classe **complexe** pourra avoir la déclaration suivante, que nous raffinerons et corrigerons par la suite :

1. B.Meyer

Exemple 2.3.1 (Exemple de déclaration de la classe **complexe** (fichier complexe.H))**.**

```
#ifndef COMPLEXE_H
#define COMPLEXE_H
class complexe {
  double re, im; // partie reelle et imaginaire
  double module();
};
#endif
```

Cette déclaration de la classe **complexe** doit se trouver dans un fichier interface, par exemple *complexe.H*. Comme on peut le voir, on déclare **complexe** comme un type comportant les attributs (les champs) *re* et *im*. De plus, cette classe possède une méthode qui se nomme *module* et qui bien entendu renverra le module du nombre complexe considéré. Dans ce fichier interface, on notera l'utilisation des instructions du préprocesseur *#ifndef*, *#define* et *#endif* qui assure que la classe **complexe** n'a pas déjà été déclarée (en particulier si on inclut plusieurs fois de suite complexe.H), ce qui provoquerait une erreur à la compilation. Une fois cette classe déclarée, il faut la définir dans un fichier source, par exemple *complexe.cpp*, dont le contenu sera :

Exemple 2.3.2 (Exemple de définition de la classe **complexe** (fichier complexe.cpp))**.**

```
#include <cmath> // pour le sqrt
#include "complexe.H"
double complexe::module(){
  return sqrt(re*re+im*im);
}
// plus les 2 autres methodes bissextile et affiche
```

On voit ici la syntaxe à utiliser pour définir une méthode de classe. Elle ressemble beaucoup à la syntaxe pour la définition des fonctions à la différence qu'on doit rappeler le nom de la classe avant de rappeler celui de la fonction (donc *Type NomClasse*::*NomMethode (arguments)*). Une autre différence vient du fait que dans le corps de la méthode, on a accès aux attributs de la classe, ie qu'ici on n'a pas a redéclarer re et im.

2.3.2 Utilisation

Une fois une classe définie, on peut l'utiliser. Pour illustrer l'utilisation de notre classe **complexe**, prenons un exemple :

Exemple 2.3.3 (Exemple d'utilisation de la classe **complexe**)**.**

```
#include <iostream>
#include "complexe.H"
int main(){
  complexe p;
  p.re = 1.2;
  p.im = 3.5;
  std::cout << "Le module du complexe " << p.re << "+(" << p.im
            << ")i est: " << p.module();
  return 0;
}
```

Ce petit programme crée un objet de la classe **complexe**, cette déclaration illustre le fait que **complexe** est bien un type. Les 2 lignes suivantes correspondent à l'initialisation du **complexe**, étape qui ne se fera en réalité jamais comme indiquée, ainsi que nous le verrons dans les paragraphes

suivants. La dernière instruction correspond à l'affichage du **complexe** ainsi que de son module, qui est calculé par l'appel de la méthode correspondante. Cet appel montre bien que la méthode est appliquée sur un objet de classe et reflète très bien l'approche orientée objet plutôt que traitement. Dans un langage fonctionnel comme le C, les fonctions ne sont pas autant liées à un type (une classe), et l'appel à une fonction calculant le module d'une nombre complexe prendrait ce nombre complexe comme argument.

On notera qu'il est possible d'utiliser une méthode sur un pointeur pointant vers un objet en utilisant la syntaxe *pobjet → méthode(arguments)*. Cependant ce cas de figure ne se produira que rarement d'autant plus qu'il est en général plus facile d'utiliser une référence (auquel cas on la manipule comme l'objet lui-même). Juste au cas où, voici un exemple d'utilisation d'un pointeur sur un objet ainsi que d'une référence :

Exemple 2.3.4 (Application de méthode sur un pointeur et une référence)**.**
```
#include "complexe.H"
int main(){
  complexe p;
  p.re = 1;
  p.im = 2;
  complexe *pp = &p; // un pointeur qui pointe sur p
  double m = pp->module();
  complexe &rp = p; // une reference, attention elle doit obligatoirement
                    // etre initialisee a la declaration
  m = rp.module();
  return 0;
}
```

2.3.3 Masquage

Dans notre petit laïus sur la conception objet, nous avons souligné le fait que pour être robuste, une classe se doit de masquer certaines informations. En règle générale, on n'autorise jamais les attributs d'une classe à être directement consultables et modifiables, car cela poserait de gros problèmes de sécurité (les utilisateurs des classes peuvent faire de très vilaines choses, vous n'avez pas idée !). Pour ce faire, on dispose de trois niveaux pour la visibilité des membres d'une classe (ie. ses attributs mais aussi ses méthodes). Un membre d'une classe peut être public, protégé ou encore privé. La visibilité des attributs se déclare lors de la déclaration de la classe en plaçant les mots-clés *public*, *protected* ou *private* suivi d'un ' :' juste avant la déclaration des dits membres. Ainsi, notre classe **complexe** avec masquage peut se déclarer de la façon suivante :

Exemple 2.3.5 (Masquage des membres de la classe **complexe** (fichier complexe.H))**.**
```
class complexe{
  private:
    double re, im;
  public:
    double module();
    double Re();
    double Im();
    double set_Re(double);
    double set_Im(double);
};
```

La signification des 3 niveaux est la suivante :
— public (*public*) : membre visible par tout client de la classe (niveau par défaut),

— protégé (*protected*) et privé (*private*) : membre inaccessible aux clients de la classe mais pas à la classe elle-même (ie qu'il est visible lors de la définition de la classe). La différence entre *protected* et *private* ne sera discernable qu'au moment de l'introduction de la notion d'héritage.

Le fait de masquer les attributs, appelé **encapsulation**, permet de montrer à l'utilisateur ce que l'on veut et sous la forme que l'on veut. Ainsi l'utilisateur ne connaît pas le détail de la structure des données, et celle-ci peut être modifiée/enrichie à loisir sans pour autant chambouler toute l'architecture.

Dans la déclaration de notre classe **complexe**, on remarque 4 nouvelles méthodes. Leurs définitions pourraient être les suivantes :

Exemple 2.3.6 (Masquage : définition de la classe **complexe** (fichier complexe.cpp))**.**

```
#include "complexe.H"
double complexe::Re() {return re;}
double complexe::Im() {return im;}
double complexe::set_Re(double x) {re = x; return re;}
double complexe::set_Im(double y) {im = y; return im;}
...
```

Les méthodes *Re()* et *Im()* sont appelées **accesseurs** et permettent au client de connaître les parties réelle et imaginaire d'un objet de la classe **complexe**. En effet, les commandes *objet.re* et *objet.im* ne sont plus réalisables par le client et provoqueraient une erreur de compilation si utilisées dans un programme (car *re* et *im* sont privées). Les méthodes *set_Re(double)* et *set_Im(double)* sont là pour permettre au client de donner une valeur aux parties réelle et imaginaire du **complexe**. On verra par la suite qu'on préfèrera en général affecter une valeur aux attributs de la classe lors de la déclaration de l'objet (ie avec une déclaration/initialisation).

2.3.4 Auto-référence : le this

Lors de la définition d'une classe (ie de ses méthodes), il peut être nécessaire d'appliquer des méthodes de la classe sur l'objet sur lequel est appliqué la méthode ou encore de renvoyer cet objet (cas le plus probable quand on parle du *this*). Or, on ne connaît pas le nom de l'objet puisqu'il n'est pas donné en argument de la méthode. Pour remédier à ce problème, l'objet manipulé lors de la définition de la classe est toujours pointé par le pointeur *this*. Un exemple pour notre classe **complexe** serait :

Exemple 2.3.7 (Auto-référence : interface)**.**

```
class complexe{
  private:
    double re, im;
    double Re2(); // renvoie carre de re
  public:
    double module();
    complexe conjugue();
    double set_Re(double);
    ...
};
```

Avec la continuation (...) indiquant d'autres méthodes telles que celles déjà employées précédemment. Une définition serait alors :

Exemple 2.3.8 (Auto-référence : définition)**.**

```
#include <cmath>
#include "complexe.H"
```

```
...
double complexe::Re2() {return re*re;}
double complexe::module() {return sqrt(Re2()+im*im);}
// ou de maniere equivalente: return sqrt(this->Re2() + im*im);
complexe complexe::conjugue() {
  im = -im;
  return *this;}
double complexe::set_Re(double re) {
  this->re = re;
  return this->re;}
...
```

Ici, la définition de la méthode conjuguant un complexe et renvoyant le résultat de cette conjugaison, utilise le pointeur *this* afin de pouvoir renvoyer l'objet sur lequel à été effectué la conjugaison. Une autre utilisation du pointeur *this* est celle faite dans la méthode *set_Re* dont la définition à la mauvaise idée d'appeler son argument *re*, ie exactement le nom d'un des attributs de la classe **complexe**. Afin de ne pas confondre les 2 variables *re* (l'attribut de la classe et l'argument de la méthode), on utilise *this* pour lever l'ambiguité. Bien entendu ceci est un exemple académique, et en principe on évite ce genre d'ambiguité.

Finalement, on notera aussi que dans la définition de la méthode *module*, on a utilisé la méthode privée *Re2* qui renvoie le carré de la partie réelle du **complexe**. Pour appeler cette méthode, on n'a pas eu besoin de spécifier sur quel objet on l'applique, puisque ce sera par défaut l'objet sur lequel a été appelé la méthode *module*.

2.3.5 Constructeur(s) & Destructeur

Pour l'instant, l'exemple de classe que nous avons vu comporte une énorme faille de robustesse. En effet, rien n'interdit de créer un objet de cette classe puis d'appliquer une méthode manipulant l'objet sans pour autant avoir initialisé de manière cohérente ses attributs. Pour remédier à ce problème, il est conseillé (en fait obligatoire) d'ajouter une ou plusieurs méthodes, appelées *constructeurs* et portant le même nom que la classe. Ces méthodes seront automatiquement appelées lors de la création d'un objet et s'occuperont de l'initialiser de façon propre. Un exemple pour notre classe **complexe** :

Exemple 2.3.9 (Classe **complexe** avec constructeurs : interface (complexe.H) et définition (complexe.cpp))**.**

```
// Interface (.H)
class complexe{
  private:
    double re, im;
  public:
    complexe();
    complexe(const complexe &)
    complexe(double,double);
};

// implementation (.cpp)
#include <iostream>

complexe::complexe() {re = 0; im = 0;}
complexe::complexe(const complexe& p) {re = p.re; im = p.im;}
complexe::complexe(double x, double y) {re = x; im = y;}
...
```

On a ici 3 constructeurs. Le premier existe toujours, même si par défaut il ne fera pas ce que l'on souhaite. Ce constructeur par défaut est celui appelé lors de la déclaration classique d'un objet. Le second constructeur est le constructeur de copie et est également toujours présent par défaut dans la classe mais ne fait que copier les attributs de la classe d'un objet à l'autre. Il peut arriver qu'il faille absolument redéfinir ce constructeur dans le cas où un des attributs de la classe est un pointeur, par exemple un tableau dynamique (pour un statique c'est inutile), car dans ce cas le constructeur par copie ne fera que copier les adresses au lieu de copier le tableau en réallouant de la mémoire. L'autre constructeur permet d'initialiser un objet au moment de sa déclaration. Grâce à la surcharge de définition, le compilateur saura quel constructeur appeler suivant le nombre et le type des arguments. Il est important de noter qu'un constructeur n'a pas de type de retour, pas même *void*. De plus, comme on peut créer plusieurs constructeurs, il est très important qu'ils aient tous un prototype différent, ie qu'il n'y ait pas d'ambiguité possible entre eux.

Le constructeur par copie est appelé à chaque fois qu'un objet de la classe est passé par valeur en argument d'une fonction. De plus il sert également de base à l'opération d'affection = quand il est utilisé dans une déclaration affectation. Par contre l'opérateur d'affectation utilisé dans son cadre classique, c'est-à-dire *Objet1 = Objet2*, n'est pas basé sur le constructeur par copie. **Mal redéfinir le constructeur par copie peut donc avoir de nombreux effets de bord difficilement traçable**. La redéfiniton de l'opérateur d'affectation est aussi à faire dans certains cas, voir la section sur la surcharge d'opérateurs.

Avec notre exemple, on a 3 façons de déclarer un objet de classe **complexe** :

Exemple 2.3.10 (Exemple d'utilisation des constructeurs)**.**

```
...
complexe p1; // constructeur par defaut appele => p1.re=0; p1.im=0;
complexe p2(1.5,-3.1); // constructeur numero 3
complexe p3(p2); // copie de p2
...
```

De plus, on note que notre premier constructeur est tout simplement le troisième avec comme arguments (0,0). Ce cas de figure se reproduit assez souvent et on peut aisément regrouper ces 2 constructeurs en un seul en prenant avantage de la possibilité de définir des arguments par défauts. Ainsi, les premier et troisième conctructeurs pourront se réécrire en un seul :

```
/* Dans complexe.H */
...
complexe(double=0,double=0);

/* Dans complexe.cpp */
...
complexe::complexe(double x, double y) {re = x; im = y;}
```

Le pendant des constructeurs est le destructeur. Il permet de libérer la mémoire allouée à un objet et est appelée automatiquement par le compilateur. Comme pour un constructeur, le destructeur ne renvoie rien, même pas *void*. Il se nomme obligatoirement *~NomClasse* et n'admet aucun argument. Dans notre exemple, la définition du destructeur serait vide puisqu'aucun type complexe n'est utilisé dans la classe **complexe** (pas de tableaux, de chaînes, de listes ..., ie. un objet nécessitant une allocation explicite de mémoire). Finalement, il est conseillé de préfixer la déclaration du destructeur par le mot-clé *virtual*, le pourquoi sera expliqué plus tard. Dans notre cas, on aurait :

Exemple 2.3.11 (Déclaration et définition du destructeur)**.**

```
// Interface
...
```

```
 virtual ~complexe();
...
// Definition
...
complexe::~complexe()
{ }
```

Typiquement, le destructeur contiendra des instructions *delete* s'il y a vraiment de la mémoire à libérer. Un destructeur n'est jamais appelé explicitement dans un programme, c'est le compilateur qui décidera tout seul quand détruire un objet. Notons cependant qu'on peut faire un appel explicite au constructeur et détruire soi-même un objet dans le cas d'une allocation dynamique :

Exemple 2.3.12 (Allocation dynamique d'un objet et destruction explicite)**.**
```
complexe *p4;
...
p4 = new complexe(p3); // appel du constructeur par copie
...
delete p4; //destruction explicite de p4
```

2.3.6 Membre statique

Un membre statique est un attribut ou une méthode persistant pour tous les objets d'une même classe, ie commun à tous les objets d'une même classe. Ceci permet de lier un membre non pas à un objet mais à une classe. Pour déclarer un membre de classe comme statique on utilise tout simplement le mot-clé *static* suivi de la déclaration usuelle du membre. Par exemple, pour connaître le nombre de **complexes** créés :

Exemple 2.3.13 (Membre statique (fichier interface complexe.H))**.**
```
class complexe{
  private:
    double re, im;
    static int cpt;
  public:
    point(double x = 0, double y = 0) {re = x; im = y; cpt++;}
    ...
    static int compteur();
    ...};
```

Ensuite, dans le cas d'un attribut statique, il ne faut pas oublier de l'initialiser dans le fichier d'implémentation. Ceci se fait simplement par *Type NomClasse*::*NomVarStatique* = *Valeur*.

Exemple 2.3.14 (Membre statique (fichier implémentation))**.**
```
#include "complexe.H"
int complexe::cpt = 0;
int complexe::compteur(){return cpt;}
...
```

A noter qu'on ne peut avoir qu'une seule initialisation d'un attribut statique. De plus, une méthode statique étant liée à la classe et pas à un de ses objets, elle n'a accès qu'aux variables statiques de la classe, pas aux attributs de l'objet (ie pas de *this* dans la définition d'une méthode statique ni d'accès aux attributs non statiques). Dans le cas d'un compteur, il faut en général l'incrémenter à chaque création d'un objet, donc il faut s'assurer que tous les moyens de création d'objet font bien l'incrémentation (ici il manque la redéfinition du constructeur par copie). De plus, il faudrait également

s'assurer qu'à chaque fois qu'un **complexe** est détruit, le compteur soit bien décrémenté, ce qui se fera dans le destructeur.

Voici un exemple d'utilisation de notre classe avec compteur.

Exemple 2.3.15 (Membre statique : exemple d'utilisation)**.**

```
#include <iostream>
#include "complexe.H"

int main(){
  complexe p1(1,2);
  complexe p2(3,4);
  complexe * p3;
  std::cout << "Il y a " << complexe::compteur() // renverra 2
            << " complexes en circulation\n";
  p3 = new complexe(5,6);
  int n = p1.compteur(); // renverra 3
  delete p3;
  return 0;
}
```

On notera que la méthode statique *compteur* peut soit être appelée directement par *complexe*::*compteur()* soit être appelée sur un objet de type **complexe**, ces 2 appels étant strictement équivalents puisque la méthode statique n'est pas attachée à un **complexe** particulier mais à toute la classe.

Un membre statique peut être vu comme une variable globale mais privée ou protégée.

2.3.7 Méthodes constantes

Une méthode constante d'une classe est une méthode dans laquelle on assure que l'objet sur lequel on travaille n'est pas modifié. La constance d'une méthode se déclare encore une fois par le mot-clé *const* qu'on place juste après la liste des arguments de la méthode (dans la déclaration et la définition).

Quand on manipule un objet déclaré comme constant, on ne peut qu'appliquer des méthodes constantes dessus, sans quoi le compilateur n'a pas de garantie que l'objet ne sera pas modifié. Voici un exemple pour les accesseurs de la classe **complexe** :

Exemple 2.3.16 (Méthode constante)**.**

```
/*  Interface 'complexe.H' */
class complexe {
  private:
    double re, im;
  public:
    double Re() const; // methode constante
    double Im(); // methode non constante mais qui devrait l'etre
    ...
};

/* Definition 'complexe.cpp' */
...
double complexe::Re() const {return re;}
couble complexe::Im() {return im;}

/* Utilisation 'main.cpp' */
#include ...
```

```
int main(){
  const complexe A(1,2);
  complexe B(3,4);
  double reA = A.Re(); // autorisee car methode constante
  double imA = A.Im(); // interdit
  double reB = B.Re(); // autorisee
  double imB = B.Im(); // autorisee
  ...
}
```

Ainsi, une méthode constante peut s'appliquer sur n'importe quel objet, qu'il soit constant ou pas. Par contre, un objet constant ne peut se voir appliquer qu'une méthode constante. **Il est donc important de déclarer les méthodes comme constantes si elles le sont**, c'est notamment le cas des accesseurs.

2.3.8 Les Amis

Une fonction, une classe ou une méthode pouvant utiliser une autre classe, il peut être intéressant d'accorder des droits privilégiés à certains clients. Pour ce faire, le langage introduit la notion d'amitié qui peut concerner une classe, une fonction ou une méthode. La déclaration d'amitié se fait dans le fichier interface de la classe voulant partager l'accès à ses membres privés/protégés et se déclare à l'aide du mot-clé *friend* suivi du nom de l'ami.

Exemple 2.3.17 (Les Amis)**.**

```
class point{
  private:
    double x, y;
  public
    ...
    // Une fonction amie: double*point
    friend point operator*(double,point);
    // Une classe amie: la classe complexe
    friend class complexe;
    // Une methode amie
    friend void uneClasse::methode();
}
```

Après ces déclarations d'amitiés, la définition de l'opérateur '*' pourra contenir des accès directs aux attributs privés x et y du **point** en argument. De même, la classe **complexe** pourra utiliser tous les membres privés de la classe **point** si jamais elle a affaire à un objet de ce type (par exemple pour construire un **complexe** à partir d'un **point**).

2.3.9 Surcharge d'opérateurs internes et externes

Comme on l'a déjà vu, la technique de surcharge peut bien entendu être utilisée dans le cadre des classes. Une surcharge particulièrement commode est la surcharge de fonctions spéciales : les opérateurs. Effectivement, il peut être utile de définir certains opérateurs classiques (+,-,<,...) dans le cadre de la nouvelle classe. Pour préciser qu'on surdéfinit un opérateur, on utilise le mot-clé *operator* directement suivi du symbole de l'opérateur. Par exemple, voici l'implémentation de la surcharge du * pour notre classe **complexe** :

Exemple 2.3.18 (Surcharge externe de l'opérateur *)**.**

```
complexe operator*(double c, const complexe &A)
{ return complexe(c*A.Re(),c*A.Im());}
```

On pourra ensuite utiliser l'opérateur de multiplication entre un *double* et un **complexe** (uniquement dans ce sens) de la même manière que sur les autres types numériques, c'est-à-dire qu'on pourra écrire $p2 = c * p1$, avec $p1$ et $p2$ des **complexes** et c un *double*. De plus, il sera possible d'utiliser la surcharge classique en définissant par exemple une autre opération de multiplication cette fois entre 2 **complexes**.

Ce type de surcharge d'opérateur est appelé surchage des opérateurs externes, car la définition est celle d'une fonction et non d'une méthode (ie. en dehors de la classe). Il est possible de faire de la surcharge d'opérateurs internes, auquel cas l'écriture :

$$A\ Op\ B$$

sera traduite par :

$$A.operatorOp(B), \text{ plutôt que par } operatorOp(A, B)$$

Avec cette syntaxe, l'ordre des opérandes est tout aussi important que pour la surcharge externe puisque c'est sur la première que s'applique la méthode. La surcharge interne d'opérateurs est donc particulièrement bien adaptée aux opérateurs qui modifient l'objet sur lequel ils travaillent, comme par exemple les opérateurs =, +=, ++, etc, mais aussi pour les opérateurs dont la première opérande est de la classe dans laquelle on surcharge l'opérateur.

Cerains opérateurs définis en internes renverront l'objet sur lequel ils travaillent, ce qui est possible grâce au pointeur *this*. Toujours pour notre classe **complexe**, on peut déclarer et définir les opérateurs + et += comme suit :

Exemple 2.3.19 (Surcharge interne d'opérateur)**.**

```
/* Interface: complexe.H */
class complexe {
...
  complexe operator+(const complexe&) const;
  complexe operator+=(const complexe&);
};

/* Implementation: complexe.cpp */
#include <complexe.H>
...
complexe complexe::operator+(const complexe &p) const {
  return complexe(re+p.re,im+p.im);
}
complexe complexe::operator+=(const complexe &p) {
  re += p.re;
  im += p.im;
  return *this;
}
/* Utilisation */
  ...
  complexe A(1,2), B(4,5), C;
  A += B;
  C = A + B;
  ...
```

On notera qu'on retourne l'objet sur lequel on travail, qui est **this* (vu que *this* est un pointeur et non l'objet lui-même).

Certains opérateurs ne peuvent cependant pas être surchargés, il s'agit de ' : :', '.', '.*', ' ? :', 'sizeof', 'typeid', 'static_cast', 'dynamic_cast', 'const_cast' et 'reinterpret_cast'.

Les opérateurs de post/pré incrémentation/décrémentation $++$ et $--$ se surcharge d'une manière un peu particulière. En effet, quand ils sont préfixés ils peuvent être déclaré comme interne et dans ce cas ils ne prennent par d'arguments (autre que l'objet sur lequel ils sont appliqués). Du coup, il faut un moyen de différencier les $--/++$ postfixés des préfixés. Ce moyen est l'utilisation d'un paramètre de type entier qui ne servira qu'à indiquer que l'opérateur $--$ ou $++$ est postfixé plutôt que préfixé. Cet argument entier ne servira pas dans la définition de la surcharge. Bien que pour notre classe **complexe** ces opérateurs n'ont pas vraiment de raison d'être, les voici tout de même.

Exemple 2.3.20 (Surcharges de $++$ et $--$)**.**

```
/* Interface: complexe.H */
class complexe {
  ...
  complexe operator++(); // prefixe
  complexe operator++(int); // postfixe
  ...
};

/* Implementation: complexe.cpp */
#include "complexe.H"
...
complexe complexe::operator++(int dummy){ // on choisit d'incrementer im
  im++;
  return *this; // post incrementation, on retourne le complexe apres modification
}
complexe complexe::operator++(){ // on choisit d'incrementer re
  complexe x(*this); // copie avant modification
  re++;
  return z; // pre incrementation, on retourne le complexe avant modification
}

/* Utilisation */
...
complexe A(1,2), B;
B = A++; // => B = complexe(1,2) et A = complexe(1,3)
B = ++A; // => B = complexe(0,3) et A = complexe(0,3)
...
```

Pour finir, un opérateur qu'il est quelquefois utile de surcharger (en tout cas pour les classes numériques) est le '<<' qu'on a déjà pu voir dans l'utilisation du *cout*. Si par exemple on veut pouvoir écrire $std :: cout << complexe(1,2)$ pour afficher à l'ecran $1 + (2)i$, on pourra surcharger $<<$ de la façon suivante :

Exemple 2.3.21 (Surcharge de $<<$)**.**

```
std::ostream & operator<<(std::ostream& f, const complexe& p){
  f << p.Re() << " + (" << p.Im() << ")i";
  return f;
}
```

Ceci représente une surcharge externe d'un flot de type *ostream*. La surcharge est forcément externe car le flot sur lequel écrire le **complexe** sera toujours l'opérande à gauche du <<.

2.3.10 Les Classes Templates

Il s'agit de classes paramétrées par un type de données abstrait, un peu comme les fonctions templates. Il faut encore une fois créer le type abstrait avec la commande *template <class T>*, puis utiliser le type T dans la définition de la classe. Pour une classe template, toutes les définitions doivent impérativement se trouver dans le fichier interface. Par exemple, pour notre classe **complexe**, on la déclarera et définira dans le même fichier et on l'utilisera de la manière suivante.

Exemple 2.3.22 (Exemple de classe template)**.**

```
/* Interface */
template <class T>
class complexeT{
  private:
    T re, im;
  public:
    complexeT(T x=0, T y=0) {re = x; im = y;}
    T Re() const {return re;}
    ...};

/* Utilisation */
#include "complexeT.H"
int main(){
  complexeT <double> A(1.3,2.1); // complexe double
  complexeT <int> B(1,2); // complexe entier
  ...
}
```

On notera que lors de la définition d'un objet de la classe **complexeT**, on spécifie le type utilisé en l'encadrant par <...>. Les classes templates sont surtout utilisées pour des listes, des tableaux, des piles, des files, ...

2.4 Héritage

2.4.1 Introduction

L'héritage est une technique clef de la programmation objet et sert principalement à répondre au souci de réutilisation. Cette technique permet de copier virtuellement les caractéristiques d'une (héritage simple) ou plusieurs (héritage multiple) classes déjà existantes dans la définition d'une nouvelle classe.

L'héritage peut servir plusieurs intérêts dont les principaux sont :

— l'extension d'une classe,
— la spécialisation d'une classe,
— l'implantation d'une classe abstraite,
— l'adaptation d'une classe.

Ces quatre points sont très proches les uns des autres et peuvent facilement être confondus. Cependant, quelque soit le but d'un héritage, la méthode reste la même : *la dérivation*.

Pour savoir si une classe hérite d'une autre, on peut utiliser un simple test sémantique. On dit qu'une classe B hérite d'une classe A si on peut dire la chose suivante :

Un objet de la classe B *est un* **objet de la classe A**

Dans ce cas, on dit que la classe A est la **classe mère** ou encore la **classe de base**. La classe B se nomme alors **classe fille** ou encore **classe dérivée**.

2.4.2 Dérivation

On dit qu'une classe B dérive d'une classe A si la classe B hérite de la classe A. Créer une classe fille se fait grâce à la syntaxe suivante :

class B : <mode_dérivation> A { ... } ;

Lors d'un héritage, la classe fille possède tous les attributs et méthodes de la classe mère. L'option <mode_dérivation> permet de déterminer quelles seront les portées des membres de B, hérités de A. Les 3 modes de dérivation sont les 3 portées définies préalablement, ie. *public*, *protected* et *private*. L'effet de ces modes de dérivation est :

- **public** : les membres publiques et protégés de la classe mère conservent leur portée, les membres privés deviennent inaccessibles (il faudra passer par les accesseurs).
- **protected** : les membres publiques et protégés de la classe mère deviennent protégés, les membres privés inaccessibles.
- **private** : les membres publiques et protégés de la classe mère deviennent privés, les membres privés deviennent inaccessibles.

Le mode de dérivation par défaut est *private*, mais le mode de dérivation le plus courant est *public* puisqu'il donne aux membres dérivés les même statuts que ceux de la classe de base.

Cependant, quelque soit le mode de dérivation, les membres privés de la classe mère deviennent inaccessibles dans la classe fille. Il est donc conseillé d'utiliser le statut *protected* plutôt que *private* si l'on souhaite pouvoir manipuler directement tous les membres d'une classe fille sans avoir à passer par les méthodes.

Voici un exemple académique de plusieurs héritages.

Exemple 2.4.1 (Exemple d'héritages simple et multiple)**.**

```
class base1 { ... };
class base2 { ... };
class derive1: public base1 { ... }; // heritage simple
class derive2: public base1, protected base2 { ... }; // heritage multiple
class derive3: public base1, public derive2,
               private base2 {...}; // heritage multiple
```

Dans notre exemple, la classe *derive1* est fille de la classe *base1* qui est alors appelée classe de base de *derive1*. La classe *derive2* possède 2 classes de base qui sont *base1* et *base2*. La classe *derive3* possède 3 classes de base qui sont *base1*, *base2* et *derive2*. On notera que *derive3* hérite en fait 2 fois de *base1* et *base2* mais à des niveaux différents.

Dans la suite de cette section, on ne s'intéressera qu'au cas de l'héritage simple.

2.4.3 Redéfinition de Méthodes

Lors d'une dérivation, tous les membres (attributs et méthodes) de la classe de base se retrouvent dans la classe dérivée. Cependant, la dérivation définissant en général des attributs supplémentaires, il peut être nécessaire de redéfinir certaines méthodes de la classe de base dans la classe dérivée. Par exemple, si on veut définir une classe *point3d* héritant d'une classe *point2d* (un point en 3 dimensions **est un** point en 2 dimensions avec une troisième coordonnée) qui possède une méthode *distance_origine* :

Exemple 2.4.2 (Exemple de redéfinition de méthode)**.**

```
// Interfaces (.H)
```

```
class point2d {
public:
  ...
  double distance_origine() const;
  ...
protected:
  double x;
  double y;
};

class point3d: public point2d {
public:
  ...
  double distance_origine() const;
  ...
protected:
  double z;
};

// Implementation de point3d (.cpp)
#include <cmath>
#include "point3d.H"
double point3d::distance_origine() const{
  return sqrt(x*x + y*y + z*z);}

// Utilisation
int main() {
  point2d p1(2,3); // en supposant qu'on a defini le constructeur necessaire
  point3d p2(1,2,3); // idem
  double dp1, dp2, dp3;

  dp1 = p1.distance_origine(); // appel de la methode de point2d
  dp2 = p2.distance_origine(); // appel de la methode de point3d
  dp3 = p2.point2d::distance_origine(); // appel de la methode de point2d
}
```

Dans cet exemple on notera que la méthode de la classe de base est toujours utilisable grâce a l'utilisation de l'opérateur de résolution de portée '::'. De plus, la méthode redéfinie doit absolument avoir la même interface que la méthode de la classe de base (sinon on fait de la surcharge, pas de la redéfinition).

L'opérateur de résolution de portée '::' sert également à récupérer un attribut appartenant à la classe mère, alors qu'il a été redéfini dans la classe fille. Par exemple :

Exemple 2.4.3.

```
// Interfaces (.H)
class mere {
  protected:
    int i;
  ...};

class fille: public mere {
```

```
  protected:
    int i;
  ...};

// Implementation (.cpp)
...
void fille::une_methode() {
  cout << "donnee de la mere = " << mere::i << endl;
  cout << "donnee de la fille = " << i << endl;
}
```

2.4.4 Constructeurs et Destructeurs d'une classe dérivée

Lors de la création d'un objet d'une classe dérivée, le constructeur de la classe de base est appelé avant celui de la classe proprement dite. Cela s'explique par le fait qu'un objet d'une classe dérivée possède tous les attributs de la classe de base en plus de ceux ajoutés dans la définition de la classe dérivée. Or, comme il faut pouvoir initialiser tous les attributs et que l'héritage rend inaccessible les membres privés de la classe de base, seul le constructeur de la classe de base peut initialiser ces attributs. Dans le cas d'un héritage ayant plus d'une *couche* (classe dérivée d'une classe dérivée...), l'appel des constructeurs se fait du plus général (la classe *racine*) au plus particulier.

Lors de la destruction d'un objet d'une classe dérivée, on appelle tout d'abord le destructeur de la classe dérivée puis seulement après le destructeur de la classe de base. L'ordre d'appel des destructeurs est ainsi l'opposé de celui des constructeurs.

Voici l'exemple d'une classe B fille d'une classe A :

Exemple 2.4.4 (Héritage et Constructeurs/Destructeurs)**.**

```
// Dans les fichiers interfaces (.H)
...
class A {
public:
  A(int val1=0);
  ...
  virtual ~A();
private:
  int arg1;
};

class B: public A {
public:
  B(int = 0, int = 0);
  ...
  virtual ~B();
private:
  int arg2;
};

// Dans les fichiers d'implementation (.cpp)
...
A::A(int val1) {arg1 = val1;  cout << ``constructeur A'';}
A::~A() {cout << ''destructeur A'';}
```

```
B::B(int val1, int val2) : A(val1) {arg2 = val2; cout << ''constructeur B'';}
B::~B() {cout << ''destructeur B'';}

// Utilisation
...
int main(){
B objB1(1);
// Resultat: constructeur A
//           constructeur B
B *pobjB;

pobjB = new B(objB1); // appel des constructeurs par recopie par defaut
                      // de la classe A puis de la classe B
delete pobjB;
// Resultat: destructeur B
//           destructeur A
...}
```

On notera que grâce à l'opérateur ' :', on peut préciser quel constructeur de la classe de base appeler. Si aucun n'est spécifié, ce sera le constructeur par défaut qui sera appelé (attention si le constructeur par défaut de la classe mère n'existe pas).

2.4.5 Polymorphisme

Dans le cas d'une dérivation, certaines conversions implicites sont définies par défaut. Si B est une classe dérivée de A, alors les conversions suivantes sont implicites :

— un objet de type B vers un objet de type A,
— un pointeur sur un objet de type B vers un pointeur sur un objet de type A,
— une référence sur un objet de type B vers une référence sur un objet de type A.

Ces trois conversions sont tout à fait naturelles puisqu'un objet de type B est également un objet de type A. La première conversion est une conversion d'objet où seuls les attributs présents dans le type A sont pris en compte alors que les autres (ceux qui n'appartiennent qu'au type B) sont ignorés. Dans les deux autres cas, les objets pointés ou référencés ne sont pas modifiés, c'est juste leur type qui change (en particulier les attributs propre au type B ne sont pas définitivement perdus). De plus, ces conversions de type impliquent qu'un pointeur sur type A peut pointer vers un objet de type B et qu'on peut retrouver un pointeur sur type B à condition de faire une conversion explicite. Cette *versatilité* des pointeurs est appelée *polymorphisme*. Voici un exemple de polymorphisme utilisant les classes *point2d* et *point3d* déclarées précédemment (on suppose que les constructeurs nécessaires ont été définis).

Exemple 2.4.5 (Exemples de polymorphisme)**.**

```
#include "point2d.H"
#include "point3d.H"

int main(){
  // objets
  point2d p1(1,2);
  point3d p2(3,4,5);
  double d;

  d = p1.distance_origine(); // appel de point2d::distance_origine()
```

```
  d = p2.distance_origine(); // appel de point3d::distance_origine()
  p1 = p2; // conversion d'objet
  d = p1.distance_origine(); // appel de point2d::distance_origine() => sqrt(3^2+4^2)

  // pointeurs
  point2d *pp1 = new point2d(1,2);
  point3d *pp2 = new point3d(3,4,5);

  d = pp1->distance_origine(); // appel de point2d::distance_origine()
  d = pp2->distance_origine(); // appel de point3d::distance_origine()
  pp1 = pp2;
  d = pp1->distance_origine(); // appel de point2d::distance_origine()
  pp2 = (point3d *) pp1; // correct uniquement car pp1 pointe bien
                         // vers un point3d
  d = pp2->distance_origine(); // appel de point3d::distance_origine()
  ...
}
```

Dans notre exemple, nous procédons à la conversion explicite d'un pointeur sur un objet de type *point2d* vers un pointeur sur un objet de la classe fille (ie plus spécifique avec plus d'attributs). Cette conversion n'est permise que parce que l'objet pointé est en réalité du type *point3d* et possède en particulier tous les attributs nécessaires. Ce genre de conversion peut s'avérer très risquée et il peut être utile de créer dans la classe de base un attribut permettant d'identifier la *place* de l'objet manipulé dans la hiérarchie d'héritage. Un tel attribut peut être utilisé afin de vérifier que la conversion utilisée a un sens.

Une autre manière de convertir un pointeur sur un objet d'une classe mère vers un pointeur sur un objet d'une classe dérivée est d'utiliser de nouvelles spécifications du C++, à savoir les nouveaux *casts* (un *cast* est une conversion explicite) :$static\_cast < T > (expr)$, $const\_cast < T > (expr)$, $dynamic\_cast < T > (expr)$ ou encore $reinterpret\_cast < T > (expr)$. Ces *casts* sont basés sur la fonctionnalité RTTI (RunTime Type Identification) et permettent des conversions de type moins ambiguë et en toute sécurité. Nous n'entrerons cependant pas dans ces détails car ils dépasseraient la portée d'une simple introduction au C++.

2.4.6 Liaison Dynamique

Dans l'exemple de la section précédente, l'appel de la méthode *distance_origine* sur le pointeur *pp1* de type *point2d** mais pointant en réalité sur un objet de type *point3d* a appelé la méthode *point2d : :distance_origine*. Le compilateur a décidé d'appeler cette méthode plutôt que celle de *point3d* en se basant sur le type déclaré de *pp1* sans tenir compte de son type réel.

Il peut cependant être préférable de ne pas choisir la méthode à appliquer lors de la compilation mais lors de l'exécution (dans le même esprit que pour les allocations dynamiques). Ceci peut par exemple être le cas dans la manipulation d'une liste d'objet appartenant à une même hiérarchie. Par exemple une liste d'objet de type *point2d* et *point3d* qui sera déclarée comme une liste d'objets de *point2d*. Pour l'instant, si on voulait créer une telle liste et en afficher chaque membre à l'aide d'une méthode *affiche*, la décision du compilateur serait d'appeler la méthode *point2d*::*affiche* sur tous les membres de la liste, même ceux de type *point3d*. Une solution, inélégante, pour palier à ce problème est d'utiliser le polymorphisme en convertissant explicitement les éléments de la liste dans le type approprié. Ainsi, si on rajoute à notre classe un attribut permettant de connaître le type de l'objet (2d ou 3d) ainsi qu'une méthode donnant le type, une solution statique est la suivante :

Exemple 2.4.6 (Solution statique à l'application d'une méthode à une famille d'objet)**.**

```
...
const int TAILLE = 100;

int main(){
  point2d* tabpts[TAILLE];
  point3d *aux;
  // Initialisation du tableau avec des pointeurs de types point2d et point3d

  for (int i=0; i<TAILLE; i++) {
    switch (tabpts[i]->type()) {
    case 2d:
      tabpts[i]->affiche();
      break;
    case 3d:
      aux = (point3d*) tabpts[i];
      aux->affiche();
      break;
    default:
      break;
    }
  }
}
```

Cette solution n'est pas très élégante et doit être modifiée après tout enrichissement de la hiérarchie d'héritage, comme par exemple si on rajoutait une classe fille de *point2d* ou *point3d*. La solution statique pose donc de vrais problèmes de maintenance.

Une solution plus élégante consiste à utiliser la *liaison dynamique* du C++. Pour ce faire, il suffit de déclarer la méthode qu'on souhaite exécuter de manière dynamique à l'aide du mot-clé *virtual*. La déclaration d'une méthode comme virtuelle se fera dans la classe de base et toutes les redéfinitions de cette méthode seront automatiquement virtuelles. Une fois une méthode déclarée comme virtuelle, c'est lors de l'exécution et non de la compilation qu'il sera décidé quelle méthode appliquer en fonction de l'objet sur lequel elle est appelée. Notre solution dynamique sera alors :

Exemple 2.4.7 (Solution dynamique de l'application d'une méthode à une famille d'objets)**.**

```
// dans l'interface de la classe point2d
class point2d{
  public:
    virtual void affiche() const;
    ...
}

// dans l'interface de la classe point3d, rien ne change
class point3d : public point2d{
  public:
    void affiche() const; // on ne rappelle pas 'virtual'
    ...
}

// Exemple liaison dynamique
...
const int TAILLE = 100;
```

```
int main(){
  point2d* tabpts[TAILLE];
  // Initialisation du tableau avec des points de types point2d et point3d

  for (int i=0; i<TAILLE; i++) {
      tabpts[i]->affiche();}
}
```

Ce choix dynamique explique pourquoi on déclare toujours un destructeur comme virtuel. Si ce n'était pas le cas, on risquerait de ne pas appeler le bon destructeur sous prétexte de polymorphisme.

2.4.7 Méthodes Virtuelles Pures

Nous savons comment déclarer une méthode comme virtuelle. Le C++ nous permet d'aller plus loin en déclarant des *méthodes virtuelles pures* dont la définition n'est pas donnée. L'intérêt est de déclarer une classe de base, possédant une méthode qu'on souhaite que toutes les classes filles possèdent. Par exemple, pour une classe *Figure* :

Exemple 2.4.8 (Déclaration d'une méthode virtuelle pure)**.**

```
class Figure {
  public:
    Figure(); //Constructeur
    virtual ~Figure(); //Destructeur
    virtual void affiche() const = 0; //Affichage
  ...
}
```

On a ainsi déclaré une classe *Figure* qui servira de cadre générique. Un objet d'une classe dérivée de *Figure* pourra toujours être affiché.

La déclaration d'une méthode virtuelle pure doit **toujours** être suivi de '= 0', comme illustrée dans l'exemple.

La fonction *affiche* n'a pas de définition dans la classe *Figure*. Cette définition devra être donnée dans les classes dérivées.

Une méthode virtuelle pure étant avant tout une méthode virtuelle, elle peut être utilisée dans le cadre de la liaison dynamique.

Attention, le fait de déclarer une méthode virtuelle pure dans une classe **interdit toute instanciation de cette classe**. En particulier, dans notre exemple, il est impossible de créer un objet de type **Figure**. On peut cependant créer un pointeur sur un objet de type **Figure**, mais on ne fera jamais d'allocation mémoire correspondant au type **Figure**. On fera plutôt du polymorphisme en affectant au pointeur de type **Figure***, de la mémoire pour un objet d'une classe fille de **Figure**, qui elle ne possèdera aucune méthode virtuelle pure et en particulier aura donné une définition à la méthode *affiche*.

2.5 Entrées/Sorties

ios_base et ses filles

En *C++*, les entrées/sorties sont gérées grâce à la classe de base *ios_base* et à ses filles. Toute la gestion est basée sur la notion abstraite de flux (stream) qui représente un médium (l'écran, le clavier, un fichier ou même une chaîne de caractère *string*) sur lequel sont effectuées les opérations. Les classes les plus utiles pour les manipulations de flux sont :

— **iostream** : pour l'écriture et la lecture sur la sortie standard (écran) et à partir de l'entrée standard (clavier).
— **ifstream** : pour la lecture dans un fichier (Input File Stream).
— **ofstream** : pour l'écriture dans un fichier (Output File Stream).
— **fstream** : pour la lecture/écriture dans un fichier (File Stream).
— **sstream** : pour la manipulation d'objet de classe *string* comme s'il s'agissait de flux (String Stream).

Lecture/Ecriture à partir du clavier ou sur l'Ecran

Nous avons déjà vu l'utilisation la plus courante de la bibliothèque *iostream* (Input Output Stream). Cette utilisation consiste simplement à utiliser le flux *std*::*cin* en conjonction avec l'opérateur >> pour lire à partir de l'entrée standard (le clavier). Pour l'écriture, on utilise les flux *std*::*cout* ou *std*::*cerr* (pour un message d'erreur) en conjonction avec l'opérateur << pour écrire sur la sortie standard (l'écran).

Lecture dans un fichier

Pour lire un fichier, il faut le placer dans un flux appartenant à la classe *ifstream*. Pour pouvoir utiliser cette classe (ainsi que la classe pour l'écriture : *ofstream*), il faut inclure l'entête *fstream*, ie $\#include\ <fstream>$

Notre objet *ifstream*, appartiendra, tout comme *cin, cout, cerr*, à l'espace de nommage *std*, donc il est conseillé d'utiliser la commande habituelle pour ne pas avoir à répéter *std* à chaque fois, ie : *using namespace std ;*.

Pour qu'un flux soit associé à un fichier, on a 2 possibilités. Soit on fait l'association lors de la déclaration du flux (grâce au constructeur, puisque le flux est un objet), soit on utilise la méthode *open*. Que vous utilisiez le constructeur ou la méthode, les paramètres seront les même, à savoir le nom du fichier (une chaîne de caractères) et le mode d'ouverture (en cas de lecture, ce sera toujours au moins *ios*::*in*, ie l'attribut *in* de la classe *ios*). De plus, une fois un flux ouvert, il est conseillé de vérifier que le flux l'a été correctement, en faisant simplement un test dessus, ie *if (flux) ....* Voici un exemple d'ouverture en lecture de deux fichiers :

Exemple 2.5.1 (Ouverture de fichiers en lecture)**.**

```
#include <iostream>
#include <fstream>

using namespace std;

int main(){
  // Ouverture avec le constructeur
  ifstream f("test1.txt",ios::in);
  if (f) {
    //operations sur le fichier
    f.close();}
  else
   cerr << "Erreur d'ouverture du fichier\n";

  // Ouverture avec open
  string nom = "test2.txt";
  f.open(nom.c_str(),ios::in);
  if (f) //... idem ...
```

```
}
```

On voit dans cet exemple qu'il faut également refermer le flux une fois qu'on n'en a plus besoin, et ce à l'aide de la méthode *close()*. On notera que lors de l'ouverture, le mode d'ouverture *ios*::*in* est facultatif car il est en fait le mode d'ouverture par défaut pour *ifstream*.

Dans notre exemple, on peut remarquer que si on souhaite utiliser une chaîne de caractère *string* comme argument pour l'ouverture (que ce soit avec le constructeur ou avec la méthode open), il faut la convertir en un *char** à l'aide de la méthode *c_str()*.

Une fois un fichier ouvert en lecture, on a principalement 3 fonctions pour effectivement lire dedans (on en a bien plus mais celles-ci seront les plus utilisées) :

- *std*::*ifstream flux*>> *variable* : récupération à partir du fichier (enfin le flux associée) jusqu'à un délimiteur (espace, nouvelle ligne).
- *getline(std*::*ifstream flux, string s[, char c])* qui lit le contenu du fichier jusqu'à rencontrer le caractère *c* et place le résultat dans la chaîne de caractère *s*. Le dernier argument est facultatif et vaut par défaut '\n', ie que par défaut on lit toute la ligne du fichier.
- *flux.get(char)* qui lit un seul caractère et le place dans la variable. Attention, les espaces, les sauts à la ligne ... sont considérés comme des caractères.

Voici un exemple d'utilisation de ces 3 fonctions, sur le fichier test.txt :

Exemple 2.5.2 (Exemple de lecture d'un fichier)**.**

```
/*****   LE FICHIER test.txt  ***********/
Ceci est un fichier de test
a45 13 machin
/*****  fin du fichier (cette ligne n'en fait pas parti) ********/

// code
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main(){
  ifstream f("test.txt",ios::in);
  if (f) {
    string s;
    getline(f,s);
    cout << s << endl; //ecrit "Ceci est un fichier de test"
    char c;
    f.get(c); // c vaut le caractere 'a'
    int n, m;
    f >> n >> m >> s; // n vaut 45, m vaut 13 et s vaut "machin"
    f.close();
  }
  else
    cerr << "Impossible d'ouvrir le fichier\n";
  // Affichage de tout le fichier
  f.open("test.txt",ios::in);
  if (f) {
    string stmp;
    while(getline(f,stmp))
```

```
      cout << stmp << endl;
  }
  else
    cerr << "Impossible d'ouvrir le fichier\n";
  return 1;
}
```

Dans la dernière partie de l'exemple, on voit qu'on peut tester la réussite de la fonction *getline* pour tester si on a atteint la fin du fichier ou non.

Ecriture dans un fichier

Comme pour la lecture, l'écriture dans un fichier passe par un flux mais cette fois de classe *ofstream*. Pour associer un flux à un fichier, on a encore 2 possibilités : le constructeur ou la méthode *open*. Les paramètres pour l'ouverture d'un fichier en écriture sont les mêmes à l'exception du mode d'ouverture. Pour les flux ofstream, il y a plusieurs modes d'ouvertures possibles qu'il faut utiliser intelligemment sous peine de perdre toutes les données se trouvant dans le fichier qu'on vient d'ouvrir. Ces modes sont :

- *ios*::*out* (pour output) : précise qu'on ouvre le fichier en écriture, il est obligatoire pour un flux ofstream mais est mis par défaut.
- *ios*::*app* (pour append) : lorsqu'on ouvre le fichier en écriture, on se trouve à la fin pour écrire des données à la suite du fichier (sans effacer le contenu, s'il y en a un). Avec ce mode d'ouverture, à chaque écriture, on est placé à la fin du fichier, même si on se déplace dans celui-ci avant (on verra comment se déplacer un peu plus tard.
- *ios*::*trunc* (pour truncate) : efface le contenu du fichier s'il est ouvert en écriture.
- *ios*::*ate* (pour at end) : positionne le curseur à la fin du fichier. La différence avec ios : :app est que si on se repositionne dans le fichier, l'écriture ne se fera pas forcément à la fin du fichier, contrairement à ios::app.

Il faut toujours utiliser le mode *ios*::*out* en conjonction avec un des 3 autres modes, ceci se fait en séparant les modes par l'opérateur ou bit à bit '|'. Ceci est très important, pour préciser ce qu'il faut faire si le fichier qu'on ouvre existe déjà.

Pour écrire dans le fichier, une fois le flux ouvert, on peut utiliser plusieurs fonctions, en voici deux :

- *flux* << *variable*, comme pour *cout*.
- *flux.put(char c)*, écrit le caractère *c* dans le fichier.

Comme la méthode *put* est juste un moyen plus compliqué que l'utilisation de <<, en général on n'utilise que ce dernier opérateur.

Voici un exemple d'écriture dans un fichier :

Exemple 2.5.3 (Ecriture dans un fichier)**.**

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main(){
  ofstream f("monfichier.dat",ios::out | ios::trunc);
  if (f) {
    string phrase = "Ma suite de fibonacci commence par";
    f << phrase << endl;
```

```
    int fibo[5] = {1,1,2,3,5};
    for (int i=0;i<5;i++)
      f << fibo[i] << ' ';
    f << endl;
    f.close();
  }
  else
    cerr << "Impossible d'ouvrir le fichier\n";
  return 1;
}
```

On voit que l'écriture dans un fichier ne diffère pas beaucoup de l'écriture à l'écran, la seule différence étant la destination (un *ostream* ou *ofstream*).

Positionnement du curseur dans un fichier

La position du curseur vous dit où dans le fichier vous allez lire ou écrire, sauf dans le cas d'ouverture de fichier en écriture avec *ios*::*app* auquel cas la position du curseur n'a aucune influence. Ceci n'est en général utile que si vous connaissez bien la structure du fichier que vous manipulez et que par exemple, vous savez que l'information qui vous intéresse se trouve à un endroit précis du fichier.

Tout d'abord, pour savoir où le curseur se trouve dans le fichier, on utilise les méthodes *flux.tellg()* si flux est de la classe *ifstream* et *flux.tellp()* si flux est de la classe *ofstream*. Ces deux méthodes renvoient la position courante dans le fichier, ie à combien d'octets du début du fichier on en est.

Pour se déplacer dans le fichier, on utilise une des deux méthodes *flux.seekg(int,ios : :[beg|cur|end])* si flux est un *ifstream* et *flux.seekp(int,ios*::*[beg|cur|end])* si flux est un *ofstream*. Le premier argument donne le nombre d'octets à sauter, le second nous dit à partir de quelle position il faut sauter :

— *ios*::*beg* (pour begin) : on saute depuis le début du fichier.
— *ios*::*cur* (pour current) : on saute depuis la position actuelle du curseur.
— *ios*::*end* (pour end) : on saute (en arrière) depuis la fin du fichier.

Le second argument par défaut est *ios*::*beg*. Donc par exemple, l'instruction :

flux.seekg(20,ios::cur)

positionne le curseur au 20 ème octet du fichier. Mais attention, ceci ne correspond pas forcément au 20 ème caractère, car le fichier n'est pas forcément écrit en ASCII.

Divers

Ouverture d'un fichier en lecture et écriture Il est possible d'ouvrir un fichier en lecture et en écriture à la fois. Ceci n'est probablement utile que quand on utilise le positionnement du curseur ou quand on veut lire tout un fichier pour ensuite écrire à sa suite (ou par dessus). L'ouverture d'un fichier en lecture et écriture se fait l'aide d'un flux de classe *fstream*. La syntaxe du constructeur est la même que pour la lecture seule ou l'écriture seule, il n'y a que le mode d'ouverture qui diffère. La syntaxe typique est :

fstream flux("nomfichier",ios::in | ios : :out | [ios::trunc | ios::ate]) ;

Dans ce cas, on ne peut pas utiliser le mode *ios*::*app*. Par contre, il faut absolument utiliser *ios*::*trunc* ou *ios*::*ate*. De plus, le fichier qu'on ouvre doit impérativement exister.

Ensuite, la lecture et l'écriture se font comme pour les fichiers ouverts en lecture seule ou en écriture seule.

Quelques fonctions utiles Voici quelques méthodes qui peuvent s'avérer utiles :

- **flux.eof()** qui renvoie *true* si on se trouve à la fin du fichier. En réalité, elle renvoie la valeur du bit *eofbit* qui est passée à *true* quand le curseur atteint la fin du fichier.
- **flux.clear()** remet tous les drapeaux (comme *eofbit* par exemple) à leur valeur d'origine. Surtout utile pour revenir au début du fichier.
- **flux.fail()** renvoie *true* si le fichier n'a pas été ouvert correctement (mais un test direct sur le flux fait la même chose).

2.6 La Bibliothèque STL

Les conteneurs, généralités

Un conteneur est un objet qui contient d'autres objets (ses éléments). Tous les conteneurs sont implémentés comme des classes templates, ce qui veut dire qu'on peut y stocker n'importe quel type d'objets, à la condition que ces derniers possèdent certaines propriétés (par exemple une relation d'ordre pour les conteneurs stockant les éléments d'une manière ordonnée). Ceci permet une grande flexibilité et explique pourquoi ils sont si populaires. Les conteneurs s'occupent eux-même de la gestion de l'espace mémoire nécessaire à leur existence, et ce à travers des méthodes qui permettent de les manipuler.

Les conteneurs implémentent des structures de données très courantes en programmation, comme par exemple les listes chaînées (list), les vecteurs/tableaux (vector) les ensembles (set), les piles (stack) ...etc. La plupart des conteneurs partagent les même fonctionnalités et le choix de l'un plutôt que de l'autre dépend fortement de ce qu'on souhaite en faire et des performances qu'on souhaite atteindre (certains conteneurs permettent des accès plus rapide à leurs éléments mais au prix d'une moins grande flexibilité).

Nous ne présenterons les conteneurs que de manière assez succincte et nous renvoyons donc le lecteur intéressé au manuel de référence de son choix.

On distingue trois familles de conteneurs. La première est celle des conteneurs séquentiels (sequence containers) :

- **vector** : vecteur
- **deque** : file à deux bouts
- **list** : une liste (chaînée)

La seconde famille est celle des conteneurs adaptatifs (il s'agit en fait d'un sous-ensemble des 'sequence containers'). Ils sont dit adaptatifs car ils ne sont qu'une 'adaptation' des conteneurs séquentiels. Il s'agit de :

- **stack** : une pile LIFO (Last In First Out)
- **queue** : une file FIFO (First In First Out)
- **priority_queue** : une file avec priorités

La dernière famille est celle des conteneurs associatifs (associative containers). Contrairement aux conteneurs séquentiels, ils ne classent pas leurs éléments suivant leurs positions. Il s'agit de :

- **set** un ensemble dont les éléments sont stockés dans un ordre dépendant d'une clef (d'un critère) donné lors de la déclaration (par défaut dans l'ordre croissant donné par <).
- **multiset** comme **set** mais permet d'avoir plusieurs éléments identiques.
- **map** stocke des paires (clef,valeur) et on retrouve la valeur à l'aide de la clef correspondante.
- **multimap** carte multiple.
- **bitset** ensemble de bits. Conteneurs conçus pour ne stocker que des variables binaires (qui ne peuvent prendre que 2 valeurs).

Pour pouvoir utiliser une des classes de conteneurs de la STL, il suffit d'inclure le fichier d'entête correspondant qui porte simplement le même nom que la classe. Par exemple, pour pouvoir utiliser une *list*, il suffit de faire un *#include <list>*.

Pour se déplacer dans un conteneur, on utilise ce qu'on appelle un itérateur (*iterator*) qui est une sorte de pointeur sur un des éléments du conteneur. Nous en verrons quelques exemples dans les sections suivantes.

Vecteur : le conteneur *vector*

Pour utiliser ce conteneur, il faut inclure la bibliothèque <*vector*>. Un vecteur est un tableau dont l'espace de stockage est géré dynamiquement et dont les éléments sont stockés dans un espace contigu de la mémoire. La gestion de la mémoire est effectuée automatiquement et l'utilisateur peut donc aisément raccourcir ou rallonger son tableau, sans souci de fuite de mémoire ou d'allocation de mémoire manuelle (par *new*, *malloc* ou ...).

Voici un exemple d'utilisation de ce type de conteneurs.

Exemple 2.6.1 (Exemple d'utilisation de < *vector* > :)**.**

```
vector<int> v1; // vecteur vide d'entiers
vector<int> v2(10,5); // un vecteur de 10 entiers valant tous 5
vector<int> v3(v2); // copie de v2
vector<int>::iterator it; // un iterateur pour vecteur d'entier
int i=0;
v1.resize(8,2); // v1 est maintenant de taille 8 avec uniquement des 2
                // (0 par defaut)
for(it=v1.begin();it!=v1.end();it++) // iteration du debut a la fin de v1
  *it = i++; // l'iterateur permet egalement de modifier une valeur
             // il se comporte presque comme un pointeur
// v1 contient maintenant 0, 1, 2, 3, 4, 5, 6, 7
cout << "Taille de v1 = " << v1.size() << endl;
cout << "Element 5 de v1 = " << v1[4] << endl;
cout << "Element 3 de v1 = " << v1.at(2) << endl;
cout << "Premier element de v1 = " << v1.front() << endl;
cout << "Dernier element de v1 = " << v1.back() << endl;
v1.reserve(100); // augmente la capacite de v1 a 100 entiers sans pour autant
                 // l'utiliser => pour ne pas perdre de temps au cas ou on
                 // aurait besoin de cette place
v2.clear(); // vide v2
if (v2.empty()) // Teste si v2 est vide
   cout << "v2 est vide!\n";
v1.erase(v1.begin()+2,v1.begin()+5); // efface les elements 3 a 5 de v1
v1.push_back(10); // ajoute 10 a la fin de v1
v1.pop_back(); // enleve le dernier element de v1
v1.insert(v1.begin()+2,v3.begin(),v3.begin()+3); // insert les 3 premiers
      // elements de v3 dans v1 a la 3eme position
```

Comme tous les conteneurs, un *vector* à une taille qui correspond aux nombres d'éléments qu'il contient et qu'on obtient à l'aide de la méthode *size*. Mais un *vector* possède également ce qu'on appelle une capacité et qu'on obtient grâce à la méthode *capacity*. Cette capacité correspond à l'espace que le *vector* réserve vraiment en mémoire. Cet espace est en général supérieur à celui dont le *vector* a besoin mais permet d'anticipé les éventuelles réallocation dynamique et ainsi d'être plus performant. Par exemple, si on déclare un *vector* de 100 entiers, le programme pourra choisir de réserver 150 espaces mémoire contigus de la taille d'un entier au cas ou l'utilisateur veut accroître la taille du *vector*.

Liste : le conteneur *list*

Comme un *vector*, le conteneur de classe *list* est un conteneur séquentiel mais celui-ci est implémenté comme une liste doublement chaînée dont chaque élément possède un lien vers celui qui le suit et celui qui le précède. Donc contrairement à un *vector*, les éléments d'une liste ne sont pas stockés de manière contiguë en mémoire. Ceci à les avantages suivants :

— L'insertion d'un élément dans la liste devient aisée et efficace.
— Manipulation efficace d'un bloc d'éléments de la liste et ce même entre listes différentes.
— Etendre la liste n'est pas aussi coûteux que pour un vecteur.

Comparées aux autres conteneurs séquentiels, les listes sont en général plus efficace dans les opérations d'insertion, d'extraction et de déplacement d'éléments d'une position de la liste à une autre. Les listes sont donc plus adaptées pour les opérations de tri par exemple.

L'inconvénient principal des listes par rapport aux autres conteneurs séquentiels est qu'elles ne possèdent pas d'accès direct aux éléments suivant leur position. Les seuls éléments rapidement accessibles sont le premier et le dernier.

Voici un petit exemple d'utilisation de la classe *list* :

Exemple 2.6.2 (Exemple d'utilisation d'une *list*)**.**

```
int T[5] = {1,4,2,8,3};
list<int> l1(T,T+5); // copie T dans une liste
list<int> l2(l1); // constructeur par copie
list<int> l3(10,1); // liste de 10 elements valant tous 1
list<int>::iterator it;
int i = 0;
for(it=l1.begin();it!=l1.end();it++)
  *it = i++; // l1 recoit {0,1,2,3,4}
l2.sort(); // ordonne l2 de facon croissante
l1.remove(0); // enleve tous les 0 de l1
l2.erase(2); // retire le troisieme elements de l2
l1.push_front(-1); // on ajoute -1 au debut de l1
l1.push_back(100); // on ajoute 100 a la fin de l1
cout << "Premier element de l1 = " << l1.front() << endl;
cout << "Dernier element de l1 = " << l1.back() << endl;
l1.pop_front(); // on retire le premier element de l1
l1.pop_back(); // on retire le dernier element de l1
it = l1.begin();
l1.insert(it+2,0); // insert 0 comme 3eme element de l1
```

Ceci n'est qu'un exemple d'utilisation et nous n'avons de loin pas utilisé toutes les possibilités des listes. On notera que la plupart des méthodes appliquables sur un *vector* le sont également sur une *list* à l'exception notable des accessions directes [] et *at*.

File et pile : les conteneurs *stack*, *queue*, *deque* et *priority_queue*

Les files et piles sont des conteneurs dont seuls quelques éléments sont directement accessibles. Voici un rapide exposé des propriétés des 4 conteneurs pouvant être assimilés à une pile ou une file :

- **stack :** un conteneur de type *stack* est implémenté comme une pile LIFO (last-in first-out) où les éléments ne sont insérés et extrait que d'un seul bout du conteneur. L'ajout ou retrait d'un élément de la pile ne se fait que par le dessus de la pile et c'est donc toujours le dernier élément ajouté qui sera retiré. Un exemple d'utilisation d'une *stack* est :
  Exemple 2.6.3 (Exemple d'utilisation d'une *stack*)**.**
  ```
  // Avec une classe Point definie comme il faut
  ```

```
stack<Point> pile; // une pile de Points vide
pile.push(Point(1,2)); // ajoute (1,2) sur la pile
pile.push(Point(0,1)); // ajoute (0,1) sur la pile
cout << pile.top() << endl; // affiche les coord. de (0,1)
pile.pop(); // retire (0,1) de la pile
cout << pile.top() << endl; // affiche les coord. de (1,2)
pile.pop(); // retire (1,2) de la pile
if (pile.empty()) // teste si la pile est vide => c'est le cas
  cout << "La pile est maintenant vide\n"
```

On notera que la méthode *pop* ne renvoie rien et ne peut donc pas être utilisée pour lire l'élément du dessus de la pile tout en le retirant.

- **queue :** un conteneur de type *queue* est implémenté comme une file de type FIFO (first-in first-out) où les éléments sont ajoutés à un bout du conteneur et extrait à l'autre bout. On insère les éléments à l'arrière (back) de la file et on les retire à l'avant (front). Un exemple d'utilisation :

  Exemple 2.6.4 (Exemple d'utilisation d'une *queue*)**.**

```
// avec une classe Point definie comme il faut
queue<Point> Q;
Q.push(Point(1,2));
Q.push(Point(3,4));
Q.push(Point(5,6));
cout << "Dernier element ajoute = " << Q.back() << endl; // concerne (5,6)
cout << "Taille de la file: " << Q.size() << endl;
while (!Q.empty()) { // tant que Q n'est pas vide
  cout << Q.front() << endl; // on affiche le devant de la file
  Q.pop(); // On retire le devant de la file
} // Affichera les coord de: (1,2), puis (3,4), puis (5,6)
```

- **dequeue :** Ce nom est un acronyme pour **d***ouble-***e***nded* **queue** et est une sorte de conteneur séquentiel. Ce qui la rapproche d'une file (*queue*) est que les premier et dernier éléments d'une *dequeue* sont les plus rapide à être accèdés. Par contre, contrairement à une file ou pile, tous les éléments d'une *dequeue* sont accessibles directement, on peut également itérer sur une *dequeue* à l'aide d'une itérateur sans pour autant devoir vider la *dequeue*. En particulier, on peut utiliser les méthodes d'accès [] et *at*, comme pour un *vector*.
- **priority_queue :** Ce conteneur correspond à une file prioritaire où le premier élément est toujours le plus grand que la file contient. La relation d'ordre permettant de définir la notion de *plus grand* est soit fournie lors de la déclaration soit est l'opérateur $<$ si celui-ci est défini pour le type des éléments à stocker. Attention, la bibliothèque à inclure pour pouvoir utiliser une file prioritaire est $<$ *queue* $>$. Un exemple d'utilisation :

  Exemple 2.6.5 (Exemple d'utilisation d'une *priority_queue*)**.**

```
priority_queue<int> PQ; // file prioritaire vide, d'entier
PQ.push(1);
PQ.push(2);
PQ.push(0);
while (!PQ.empty()) {
  cout << PQ.top() << " ";
  PQ.pop();
} // affiche: 2 1 0
```

  Pour l'utilisation d'une file prioritaire avec une fonction de comparaison fournie par l'utilisateur, nous renvoyons le lecteur à un manuel de référence.

Ensembles : les conteneurs *set* et *multiset*

Les conteneurs de types *set* et *multiset* représentent des conteneurs associatifs où les éléments sont stockés suivant une relation d'ordre sur leur *clef*. Pour ces classes, les éléments sont leur propre clefs. La différence majeure entre un *set* et un *multiset* est que le *set* ne permet pas la coexistence d'éléments identiques (donc de clefs identiques) alors qu'un *multiset* le permet. A chaque insertion d'un élément dans un *set* ou *multiset*, ce dernier est ordonné dans le conteneur. La lecture d'un élément ne peut se faire que par un itérateur. Voici un exemple d'utilisation d'un *set* et d'un *multiset* :

Exemple 2.6.6 (Exemple d'utilisation d'un *set* et *multiset*)**.**

```
set<int> S;
multiset<int> MS;
set<int>::iterator itS; // iterateur sur set
set<int>::reverse_iterator ritS; // iterateur inverse sur set
multiset<int>::iterator itMS; // iterateur sur multiset
S.insert(1); // S = {1}
S.insert(1); // S = {1} car pas de doublons dans un set
MS.insert(1); // MS = {1}
MS.insert(1); // MS = {1,1}
S.insert(0); // S = {0,1}
S.insert(3); // S = {0,1,3}
cout << "Nombre de '1' dans MS = " << MS.count(1) << endl; // retourne 2
cout << "{";
for(ritS = S.rbegin();ritS!=S.rend();ritS++)
  cout << *ritS << " ";
cout << "}\n";
// Affiche: {3 1 0}
itS = S.lower_bound(1); // renvoie iterateur sur 1: premier >= a 1
itS = S.upper_bound(1); // renvoie iterateur sur 3: premier > a 1
```

On notera que la méthode *count* pour un *set* revient à un méthode de test de présence car la résultat est soit 1 (l'élément s'y trouve) soit 0 (il ne s'y trouve pas).

Attention, pour utiliser un conteneur de type *multiset*, il faut inclure la bibliothèque $<set>$ et pas $<multiset>$.

Cartes : les conteneurs *map* et *multimap*

Ces conteneurs sont des conteneurs associatifs, tout comme *set* et *multiset* à la différence que les *clefs* des éléments sont maintenant fournies en plus des éléments.

Les algorithmes

L'inclusion de l'entête <algorithm> permet d'utiliser tout un éventail de fonctions pouvant agir sur un tableau ou quelques uns des conteneurs de la STL. Il est important de noter que ces algorithmes n'agissent que sur les éléments des conteneurs (ou du tableau) et ne peuvent en aucun cas en modifier la structure (pas de rajout d'éléments par exemple).

Voici une sélection de ces fonctions :

- **Algorithmes ne modifiant pas le conteneur :**
  - *Function for_each (InputIterator first, InputIterator last, Function f)* : Applique la fonction f à tous les éléments de l'ensemble $[first, last]$. à une collection d'éléments.
  - *InputIterator find (InputIterator first, InputIterator last, const T& value)* : Renvoie un itérateur sur le premier élément de l'ensemble $[first, last]$ égale à *value*. Si un tel élément n'existe pas, renvoie l'itérateur sur l'elément juste après *last*.

- *InputIterator find_if (InputIterator first, InputIterator last, Predicate pred)* : Renvoie un itérateur sur le premier élément de l'ensemble $[first, last]$ qui satisfait le prédicat *pred*.
- *ForwardIterator1 find_end (ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred)* : Cherche dans l'ensemble $[first1, last1]$ la dernière occurrence de la suite de valeurs $[first2, last2]$ et renvoie un itérateur sur le premier élément de cette dernière occurrence. La relation de comparaison utilisée est *pred* ou par défaut l'opérateur ==.
- *ForwardIterator1 find_first_of (ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred)* : Trouve le premier élément de l'ensemble $[first1, last1]$ qui soit aussi élément de $[first2, last2]$ et renvoie un itérateur sur l'élément de $[first1, last1]$ trouvé. La comparaison est faite grâce à *pred* ou == par défaut.
- *ForwardIterator adjacent_find (ForwardIterator first, ForwardIterator last, BinaryPredicate pred)* : Trouve dans l'ensemble $[first, last]$ les 2 premiers éléments égaux entre eux et renvoie un itérateur sur le premier des 2 éléments. La comparaison est faite par *pred* ou == par défaut.
- *int count(ForwardIterator first, ForwardIterator last, const T& value)* : Renvoie le nombre d'éléments de l'ensemble $[first, last]$ qui sont égaux à *value* (le type renvoyé n'est pas exactement *int* mais *typename iterator_traits¡InputIterator¿ : :difference_type*).
- *count_if(ForwardIterator first, ForwardIterator last, Predicate pred)* : Renvoie le nombre d'éléments de l'ensemble $[first, last]$ qui satisfont le prédicat *pred*. Même remarque que pour *count* sur le type de retour.
- *pair¡InputIterator1,InputIterator2¿ mismatch (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, BinaryPredicate pred)* : Renvoie une paire d'itérateurs pointant sur les premières position pour lesquelles les ensemble $[first1, last1]$ et $[first2, ?]$ diffèrent. La comparaison est faite avec le prédicat *pred* ou avec == si *pred* n'est pas fourni.
- *bool equal(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, BinaryPredicate pred)* : Teste si l'ensemble $[first1, last1]$ et celui commençant en *first2* sont égaux. Le teste d'égalité est fourni par *pred* ou par défaut est pris comme l'opérateur ==.
- *ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred)* : Renvoie un itérateur sur le premier élément de l'ensemble $[first1, last1]$ qui commence la suite d'éléments défini par $[first2, last2]$. Le teste de comparaison est fourni par *pred* ou est pris comme étant l'opérateur ==.
- *ForwardIterator search_n(ForwardIterator first, ForwardIterator last, Size count, const T& value. BinaryPredicate pred)* : Cherche dans l'ensemble $[first, last]$, une succession de *count* élément ayant pour valeur *value*. Renvoie un itérateur sur la première de ces successions de valeurs.

- **Algorithmes modifiant le conteneur :**
  - *OutputIterator copy(InputIterator first, InputIterator last, OutputInterator result)* Copie les valeurs pointées entre *first* vers la zone mémoire pointée par *result*.
  - *void swap(T& a,T& b)* Intervertie les 2 variables (de type T) a et b.
  - *OutputIterator transform(InputIterator first1, InputIterator last1, OutputIterator result, UnaryOperator op)* Applique la fonction unaire *op* à tous les éléments compris entre *first1* et *last1* et stocke le résultat à partir de *result*. Une version avec un opérateur binaire existe aussi, auquel cas il faut rajouter un paramètre *first2* (un *last2* est inutile car on connaît déjà la taille).
  - *void replace (ForwardIterator first, ForwardIterator last, const T& old_value, const T& new_value)* : Remplace toutes les valeurs comprisent entre *first* et *last* qui sont égales à *old_value* par la *new_value*.
  - *void replace_if (ForwardIterator first, ForwardIterator last, Predicate pred, const T& new_value)* :

Remplace toutes les valeurs comprises entre *first* et *last* qui satisfont le prédicat *pred* et ce par la valeur *new_value*.

— *void generate (ForwardIterator first, ForwardIterator last, Generator gen)* : Remplace les valeurs comprises entre *first* et *last* par celles générées par l'appel consécutifs de la fonction *gen*.

— *ForwardIterator remove (ForwardIterator first, ForwardIterator last, const T& value)* : Retire de l'ensemble pointé par *first* et *last*, toutes celles égales à *value*. La fonction retourne un itérateur/pointeur sur la fin du nouvelle ensemble (qui est plus court que celui d'origine).

— *ForwardIterator remove_if (ForwardIterator first, ForwardIterator last, Predicate pred)* : Même idée que le *remove* mais cette fois on retire un élément qui satisfait le prédicat *pred*.

— *ForwardIterator unique (ForwardIterator first, ForwardIterator last, BinaryPredicate pred)* : Retire les éléments qui sont égaux à leur voisin. Le teste d'egalité se fait avec la relation de comparaison *pred* (par défaut cette relation est l'opérateur ==).

- **Algorithmes de tri et de recherche :**
  — *void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp)* : Trie les éléments compris entre les pointeurs *first* et *last* suivant la relation de comparaison *comp*. Par défaut cette dernière est l'opérateur <.
  — *ForwardIterator lower_bound (ForwardIterator first, ForwardIterator last, const T& value, Compare comp)* : Retourne un pointeur sur le premier élément de l'ensemble trié [*first*,*last*] qui ne soit pas inférieur à la valeur *value*. La comparaison se fait grâce à la fonction *comp* qui par défaut est prise comme l'opérateur <.
  — *ForwardIterator upper_bound (ForwardIterator first, ForwardIterator last, const T& value, Compare comp)* : Retourne un pointeur sur le premier élément de l'ensemble trié $[first, last]$ qui soit supérieur à *value*. La comparaison se fait grâce à la fonction *comp* qui part défaut est prise comme étant l'opérateur <.
  — *bool binary_search (ForwardIterator first, ForwardIterator last, const T& value, Compare comp)* : Teste si la valeur *value* est présente dans l'ensemble $[first, last]$. Le teste d'égalité est effectué avec la fonction de comparaison *comp* ou par défaut avec l'opérateur <.
- **Algorithmes de fusion :**
  — *OutputIterator merge (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp)* : Fusionne les deux ensembles ordonnés $[first1, last1]$ et $[first2, last2]$ en un nouvelle ensemble ordonné commençant en *result*. La relation de comparasion utilisée est *comp* ou < par défaut.
  — *bool includes (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, Compare comp)* : Teste si tous les éléments de $[first2, last2]$ se trouvent dans $[first1, last1]$, en utilisant la relation de comparaison *comp* ou < par défaut.
  — *OutputIterator set_union (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp)* : Réalise l'union entre les ensembles ordonnés $[first1, last1]$ et $[first2, last2]$ et la place en *result*. La relation utilisée est *comp* ou par défaut <. La fonction renvoie l'itérateur/pointeur sur le dernier élément de l'union.
  — *OutputIterator set_intersection (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp)* : Réalise l'intersection entre les ensembles ordonnés $[first1, last1]$ et $[first2, last2]$ et place l'intersection en *result*. La comparaison est faite grâce à *comp* ou à < si *comp* n'ets pas spécifié. La fonction retourne l'itérateur/pointeur sur le dernier élément de l'intersection.
- **Algorithme d'extrémalité :**
  — *const T& min (const T& a, const T& b, Compare comp)* : Retourne le minimum entre les deux valeurs *a* et *b*. La relation de comparaison est *comp* ou < si *comp* n'est pas spécifié.
  — *const T& max (const T& a, const T& b, Compare comp)* : Idem que *min* mais renvoie le

maximum.
— *ForwardIterator min_element (ForwardIterator first, ForwardIterator last, Compare comp)* : Renvoie un itérateur sur le plus petit élément de l'ensemble $[first, last]$.
— *ForwardIterator max_element (ForwardIterator first, ForwardIterator last, Compare comp)* : Idem que *min_element* mais renvoie la valeur maximum.

2.7 Autres bibliothèques utiles

Les chaînes de caractères : string

Les objets de la classe *string* sont en fait des conteneurs spéciaux fait pour contenir et manipuler des suites de caractères. Contrairement aux chaînes de caractères du *C* (les *char **) qui ne sont en fait que des tableaux de caractères, ie des caractères stockés contigüment en mémoire, la classe *string* possède de nombreuses fonctions et opérateurs permettant de manipuler ses objets de façons beaucoup plus intuitive.

La classe *string* est en réalité une instanciation de la classe template *basic_string* et est définie comme :

$$typedef\ basic\_string < char > \ string$$

Tous les objets *string* se trouve dans l'espace de nommage standard *std*. Il faut donc bien penser à utiliser le *using namespace std ;* avant le code ou a préfixer toutes les références à un objet ou méthode de *string* par *std* ::.

Voici un aperçu des méthodes disponibles pour la classe *string* :

- **Constructeur(s) :** On peut construire un objet de type *string* de plusieurs manière différente, en voici les plus utiles :
```
string s1("Aloha le monde"); // ca devrait etre clair
string s2 = "Aloha the world"; // ca aussi
string s3(s1); // constructeur par copie
string s4(s2,6); // copie des 6 premiers caracteres de s2
string s5(s2,7,9); // copie des caracteres 7 a 9 de s2
char s[10] = "bonjour";
string s6(s); // ca devrait etre limpide
```
- **Affectation :** L'opérateur d'affectation est bien entendue définie : $s2 = s1$ si $s1$ et $s2$ sont des *string*. Mais aussi si $s1$ est un *char** ou simplement un *char*.
- **Itérateurs :** Un *string* étant un conteneur, on peut le manipuler avec des itérateurs qui pointeront sur un des caractères de la chaîne. On a pour ça les méthodes suivantes :
```
string s("Aloha");
string::iterator it; // un iterateur
it = s.begin(); // iterateur sur le premier caractere
it = s.end(); // iterateur sur le caractere juste apres le dernier
string::reverse_iterator rit; // un iterateur inverse
rit = s.rbegin(); // iterateur inverse sur le dernier caractere
rit = s.rend(); // iterateur inverse sur l'emplacement juste avant le premier de s
for(it=s.begin();it!=s.end();it++)
  cout << *it; // pour afficher la chaine 's'
for(rit=s.rbegin();rit!=s.rend;rit--)
  cout << *rit; // afficher la chaine 's' a l'envers
cout << endl << "Je viens de faire un palindrome!\n";
```
- **Capacités :** On peut aisément connaître, manipuler ou tester la capacité d'un *string*. Un exemple est le suivant :

```
string s("Aloha");
cout << "Taille (par size) = " << s.size() << endl; // retourne 5
cout << "Taille (par length) = " << s.length() << endl; // idem
cout << "Taile maximum = " << s.max_size() << endl; // taille maximum
cout << "Capacite = " << s.capacity() << endl; // capacite
s.resize(10); // on reserve de la memoire pour 5 caracteres supplementaires
s.resize(4); // la chaine ne comporte plus que 4 caracteres, les 4 premiers
s.resize(10,'a'); // on reserve de la memoire pour 6 caracteres de plus
                  // et ces 6 caracteres sont initialises a 'a'
s.clear(); // vide la chaine de caracteres
if (s.empty())  // teste si 's' est vide (c'est le cas)
  cout << "La chaine est vide!\n";
```

La taille maximum retournée par *max_size* dépend de l'état actuel de la mémoire mais est en général très importante (de quoi contenir un bouquin). La capacité donnée par *capacity* correspond, comme pour un *vector* à l'espace mémoire que le constructeur à réservé pour stocké *s* et qui est en général un tout petit peu plus important que le strict minimum nécessaire en prévision des manipulations à venir.

- **Accès aux éléments :** Pour accéder à un élément d'une chaîne de caractère, on peut (en plus des itérateurs) utiliser [] et *at*. Attention, on reste dans la logique des tableaux et le premier élément est en position 0.

```
string s("Aloha");
cout << s[0]; // premier element
sout << s.at(1); // second element
```

- **Modifications :** On possède plusieurs méthodes ou opérateurs pour modifier un *string*. Voici un exemple d'utilisation de ces opérateurs :

```
string s1("Aloha"), s2("le"), s3("Monde");
string s;
s = s1 + " " + s2; // s = "Aloha le"
s += " "; // s = "Aloha le "
s.append(s3); // s = "Aloha le Monde"
s.push_back('!'); // ajoute le caractere '!', donc s = "Aloha le Monde!"
s.assign(s1,4); // s recoit les 4 premiers caracteres de s1, ie s = "aloh"
s.assign(s1,2,3); // s recoit les 3 caracteres de s1 a partir du second,
                  // donc s = "loh"
s.assign(6,'*'); // s recoit 5 fois le caractere '*', ie s = "******"
s.insert(4,s1); // insert s1 a la 4eme position de s, ie s = "***Aloha***"
                // fonctionne egalement avec des iterateurs
s.erase(4,5); // efface les 5 caracteres en commencant a la position 4,
              // donc s = "******"
              // fonctionne egalement avec des iterateurs
```

On a également les méthodes *replace*, *copy* et *swap* qui peuvent être utiles et qui peuvent être appelées de plusieurs façon différentes.

- **Opérations :** Voici quelques opérations utiles sur les *string* :

```
char *cs; // une chaine de caracteres C
string s("Oh la jolie phrase");
cs = new char[s.size()+1];
strcpy(cs,s.c_str()); // transforme s en char*, vous connaissez deja strcpy
string s1("phrase");
size_t ici;
ici = s.find(s1); // cherche "phrase" dans la chaine s, renvoie la position
```

```
s1 = s.substr(3,8); // extrait "la jolie" de s
if (s == s1) // retourne true si s et s1 sont identiques (pas le cas ici)
  cout << "s et s1 sont les memes\n";
```