Programmation en C++

Notions de base

par

Olivier Bernard
Thomas Grenier
olivier.bernard@creatis.insa-lyon.fr
thomas.grenier@creatis.insa-lyon.fr

Département Génie Electrique
Institut National des Sciences Appliquées de Lyon
Villeurbanne - France

Table des matières

1 Développement en C++

1.1 Rappels sur la compilation

Avant de donner plus de détails sur les notions de base en C++, il est important de rappeler les différentes étapes intervenant lors de la création d'un exécutable (ou d'une librairie) à partir de fichiers C++. Figure 1 donne un schéma simplifié de la création d'un exécutable lors de la compilation en C++.

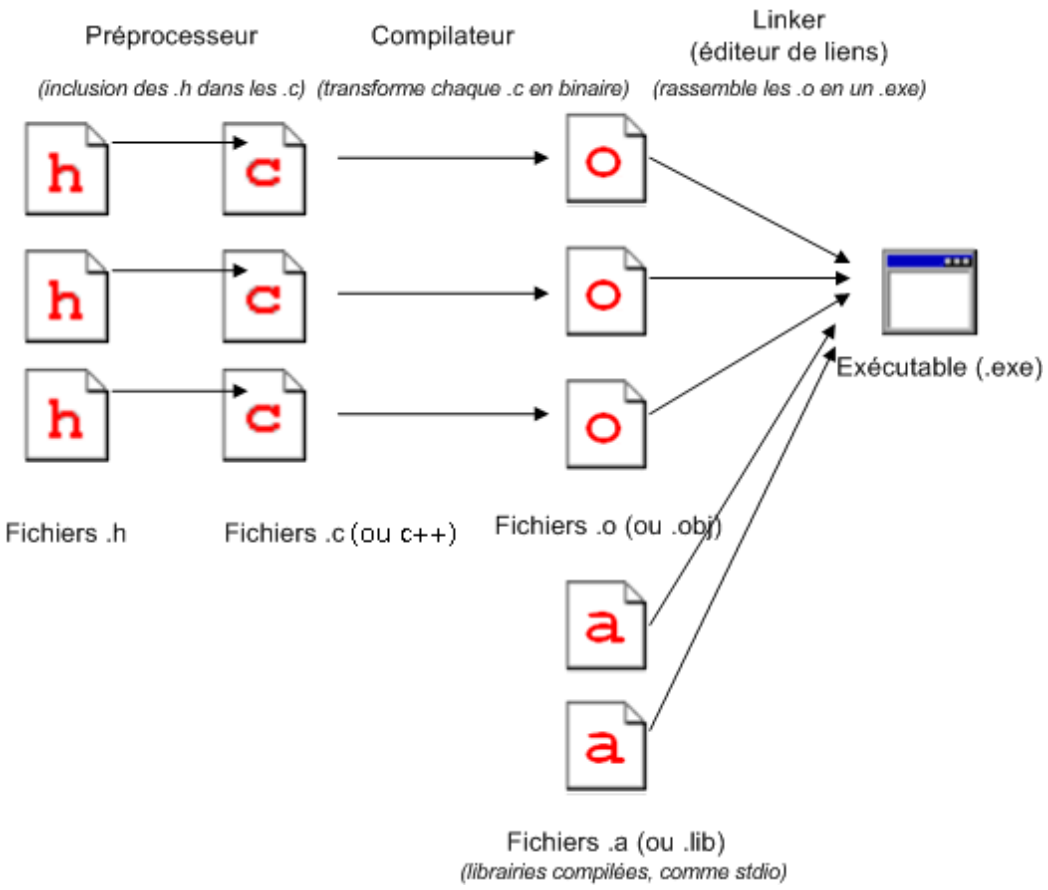

FIGURE 1 – Création d'un exécutable lors de la compilation en C++

De façon simplifiée, trois étapes sont réalisées lors de la création d'un exécutable.

1.1.1 Pré-processeur

Le pré-processeur ou pré-compilateur est un utilitaire qui traite le fichier source avant le compilateur. C'est un manipulateur de chaînes de caractères. Il retire les parties de commentaires, qui sont comprises entre `/*` et `*/`. Il prend aussi en compte les lignes

du texte source ayant un `#` en première colonne pour créer le texte que le compilateur analysera. Ses possibilités sont de trois ordres :
– inclusion de fichiers (mot clé `#include`). Lors de cette étape, le pré-compilateur remplace les lignes `#include` par le fichier indiqué ;
– définition d'alias et de macro-expression (mots clés `#define`, `macro`) ;
– sélection de parties de texte (mot clé `inline`).
A la fin de cette première étape, le fichier *.cpp* (ou *.c*) est complet et contient tous les prototypes des fonctions utilisées.

1.1.2 Compilateur

Cette étape consiste à transformer les fichiers sources en code binaire compréhensible par l'ordinateur. Le compilateur compile chaque fichier *.cpp* (ou *.c*) un à un. Le compilateur génère un fichier *.o* (ou *.obj*, selon le compilateur) par fichier *.cpp* compilé. Ce sont des fichiers binaires temporaires. Ces fichiers sont supprimés ou gardé selon les options utilisées pour la compilation. Enfin, il est à noter que ces fichiers peuvent contenir des références insatisfaites qui seront résolues par l'éditeur de liens.

1.1.3 Editeur de liens

L'éditeur de liens prend chaque fichier généré par le compilateur (*.o*) et les associe pour créer un module chargeable (l'exécutable). Il se sert de bibliothèques pour résoudre les références insatisfaites.

1.2 Environnement de développement

Le mot clé en anglais est Integrated Development Environment (IDE). Un environnement de développement est un programme regroupant généralement tous les outils necessaires au developpement. Les plus efficaces sont ceux qui permettent :
– la création et la gestion de projets (plusieurs fichiers, plusieurs exécutables, plusieurs librairies, ...) ;
– l'édition des fichiers avec une reconnaissance de la syntaxe ;
– la compilation, l'exécution et le débugage des projets.
Il existe de nombreux IDE C++. Voici une liste non exhaustive d'IDE (gratuits) :
– MS Visual Express : http ://msdn.microsoft.com/fr-fr/express/aa975050.aspx
– Dev C++ : http ://sourceforge.net/projects/dev-cpp/
– wxDevC++, la version de Dev C++ avec un outil de création d'interface graphique (RAD) : http ://wxdsgn.sourceforge.net
– Eclipse, supportant de nombreux langages : http ://www.eclipse.org/
– CodeBlocks : http ://www.codeblocks.org/downloads
Nous présentons ici une démarche de développement en se basant sur un "vieux" IDE mais très classique, ***Microsoft Visual Studio C++*** (version 6.0) sous *Windows*.

1.2.1 Environnement Visual Studio C++ 6

La figure 2 donne un aperçu de l'interface (bien évidemment, la disposition des outils peut être différente).

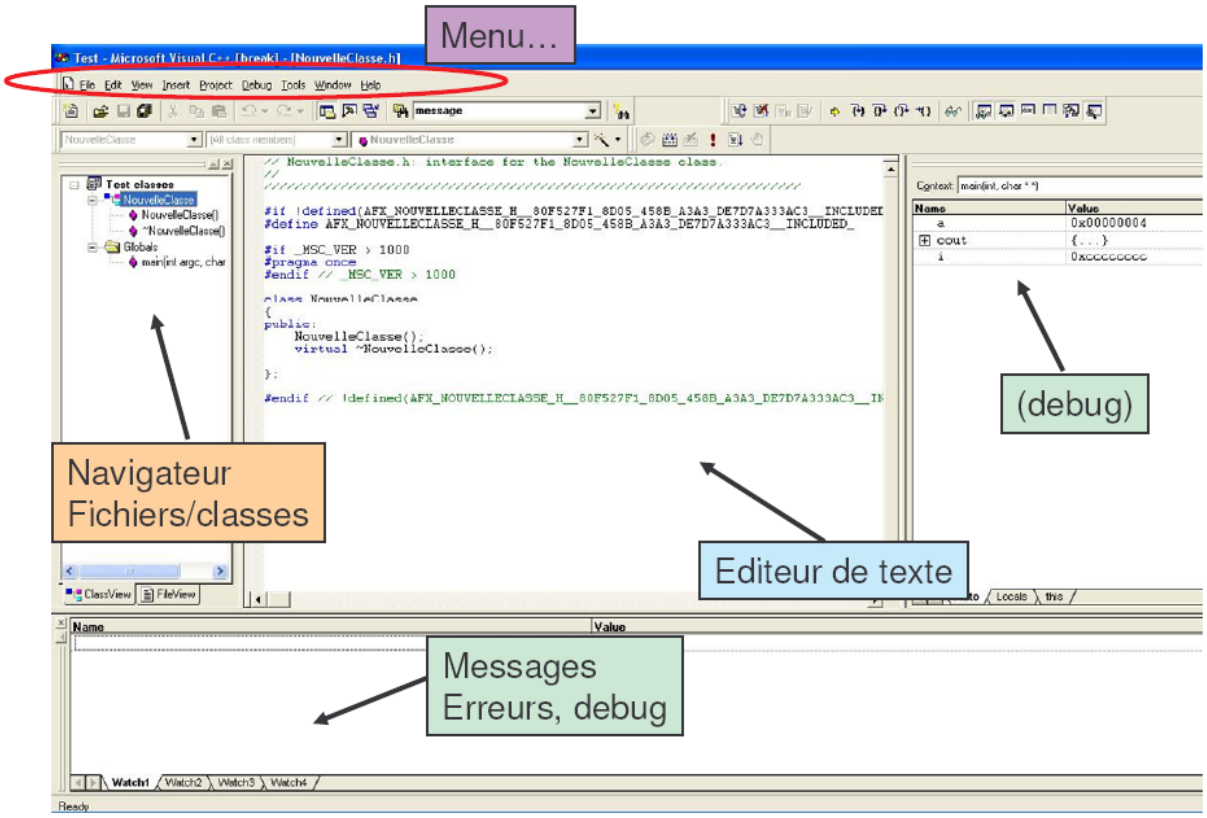

FIGURE 2 – interface type de Visual Studio C++ 6

Dans ce qui suit, nous décrivons par la suite les étapes clées afin d'être rapidemment autonome sur cet IDE. Notons que les démarches et les terminologies pour les autres IDE sont très proches de celles présentées ci-après.

1.2.2 Création d'un projet

La première étape consiste à créer un projet. Pour cela, il suffit de réaliser les étapes suivantes :
– à partir du menu faire : ***File*– >*New*** ;
– dans la fenête qui apparait choisir : ***Projects*** ;
– dans le cadre d'une utlilisation basique choisir ***Win32 Console Application*** ;
– enfin, donner un nom à votre projet ainsi que son emplacement.
La figure 3 donne un aperçu de la fenêtre de dialogue de sélection du type de projet.

1.2.3 Compilation et exécution

Il existe différentes échelles de compilation :
- compilation du fichier sélectionné. Ceci permet de corriger les erreurs de syntaxe éventuelles. Cette étape s'effectue de la façon suivante : à partir du menu faire : ***Build*** $->$ ***Compile*** ;
- compilation de l'ensemble du projet. Un projet est généralement constitué par un ensemble de fichiers (.h, .cpp, .cxx). Cette compilation permet de tester la syntaxe de l'ensemble des fichiers ainsi que leurs interractions. De plus, avant la compilation, tous les fichiers sont enregistrés et les modifications récentes sont prises en compte par le compilateur. Cette étape s'effectue de la façon suivante : à partir du menu faire : ***Build*** $->$ ***Build*** ;

Lors de la compilation, les erreurs de syntaxe détectées sont signalées dans l'onglet *Build* de l'outil *messages* (*cf.* figure 2). Il est important de noter que l'erreur est systématiquement décrite et qu'un double clic sur cette description envoie le curseur à la ligne correspondant à l'erreur détectée dans le fichier concerné. Ceci facilite grandement la résolution d'erreur de syntaxe dans les fichiers de code. La figure 4 donne un exemple de résultat de compilation avec un message d'erreur.
Une fois l'ensemble du projet compilé (sans erreur de compilation), il est possible d'exécuter le programme de la façon suivante : à partir du menu faire : ***Build*** $->$ ***Execute*** ;

1.2.4 Debugage

Il existe plusieurs mode de compilation. Par défaut, Visual Studio C++ 6 utilise le mode de compilation ***debug***. Contrairement au mode ***release***, le mode ***debug*** permet d'effectuer le debogage d'un programme. Le debogage d'un projet permet d'interagir avec le programme pendant son exécution. Cela permet notamment de surveiller les valeurs des variables, contrôler les passages dans les tests, l'exécution de boucles, l'allocation des objets, modifier les valeurs de variables ...). Le debugage d'un programme s'effectue de la façon suivante :
- positionner dans le code un ou plusieurs ***breakpoint***. Cette étape s'effectue de la façon suivante : à partir de la ligne de code où l'on souhaite insérer un breakpoint faire : ***Clic droit*** $->$ ***Insert/remove Breakpoint*** ;
- exécuter le code en mode ***debug***. Cette étape s'effectue de la façon suivante : à partir du menu faire : ***Build*** $->$ ***Start*** $->$ ***Go*** *ou* **F5**.

L'excution du programme sera alors mise en pause juste avant l'exécution de la ligne où est positioné le *breakpoint*. Une fois le programme mis en pause, on peut interagir avec les variables et la progression de l'exécution en utilisant les bouttons de la barre d'outil ***Step Over*** (*F10*) et ***Step Into*** (*F11*). La figure 5 donne un exemple de debugage d'un programme.

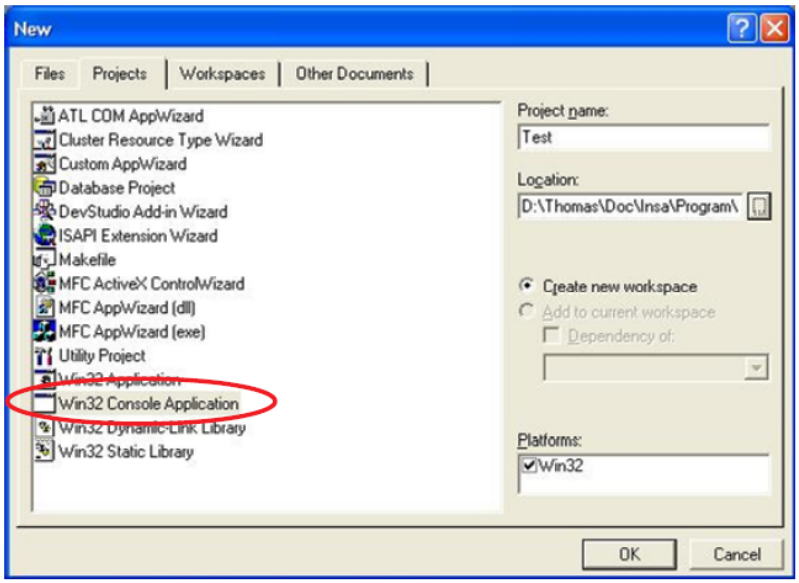

FIGURE 3 – fenêtre de dialogue de sélection du type de projet sous Visual Studio C++ 6

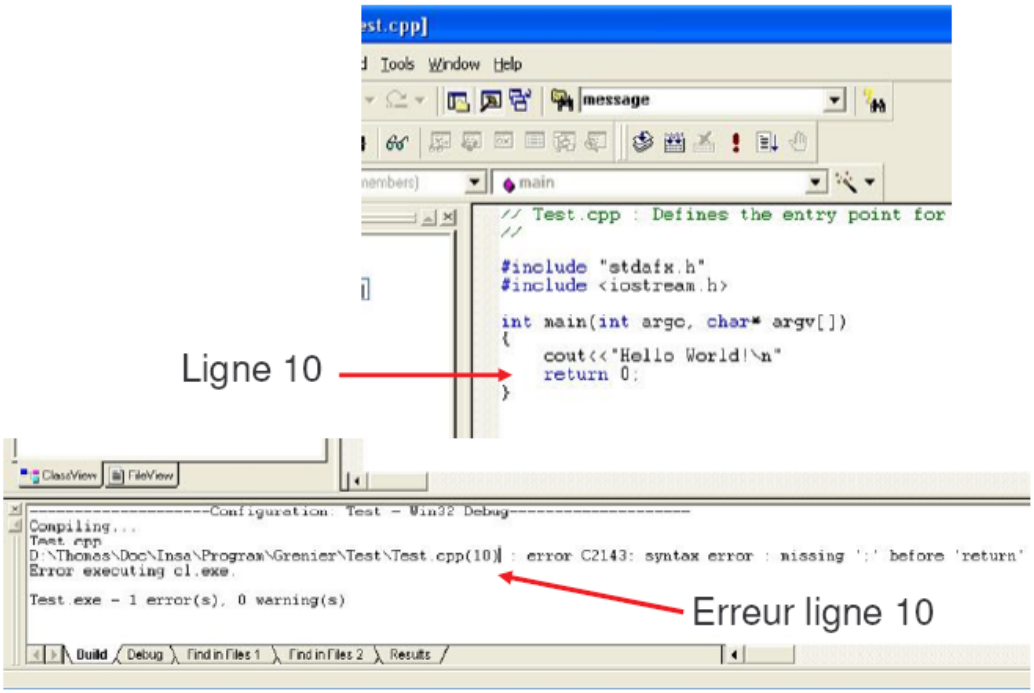

FIGURE 4 – Exemple de résultat de compilation avec un message d'erreur sous Visual Studio C++ 6

1.2.5 Ajout de classe

Afin d'ajouter une classe à un projet, il suffit d'effectuer l'étape suivante : à partir du menu faire ***Insert*** $->$ ***New Class***. Deux fichiers sont ainsi créés dans le repertoire du projet : *NouvelleClasse.h* et *NouvelleClasse.cpp*. Ces deux fichiers sont également ajoutés au projet.
Il est à noter la présence de majuscules/minuscules aux noms de fichiers. Cependant *Windows* ne différentie pas la casse pour les noms de fichier contrairement à *Linux* et à *Mac OS*. Ainsi, afin d'être le plus portable possible, il est important de respecter la casse pour les noms de fichiers. Dans l'exemple précédent, afin de pouvoir utiliser la nouvelle classe, il faut donc ajouter la ligne suivante :

```
#include "NouvelleClasse.h"
```

1.2.6 Remarques

Lors de la première compilation, vous pouvez avoir des erreurs du type : « *Error cannot find ... .pch* » ou « *unexpected end of file ... precompiled header* ». Ces erreurs sont liées aux options de compilation de votre projet. Pour modifier l'option responsable de ces erreurs, il est nécessaire de changer l'option de précompilation des entêtes. Pour cela, il suffit de faire à partir du menu : ***Project*** $->$ ***Settings*** $->$ ***Onglet*** « ***C/C++*** », puis dans la boite « *Category* » choisir « *Pre compiled headers* ». Ensuite choisir l'option « *not using precompiled headers* » (cf. tableau 6). Finalement il faut supprimer du projet les deux fichiers *stdafx.h* et *stdafx.cpp* ainsi que la ligne :

```
#include "stdafx.h"
```

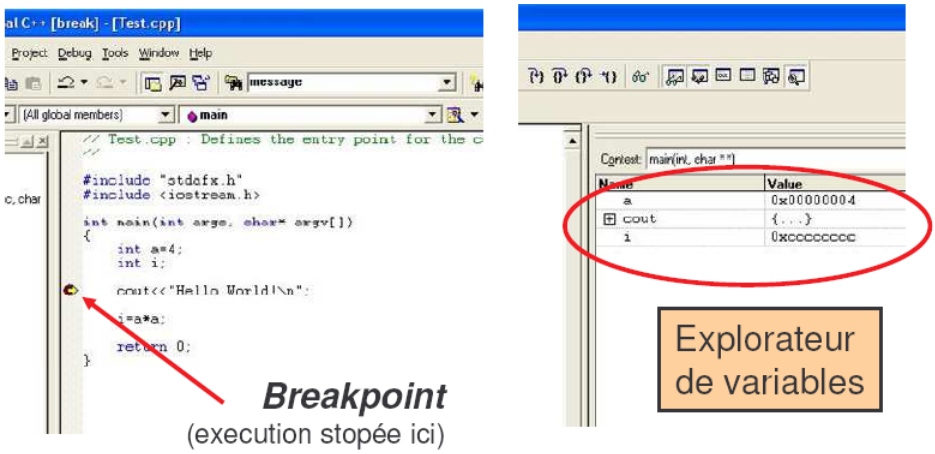

FIGURE 5 – Exemple de debugage d'un programme sous Visual Studio C++ 6

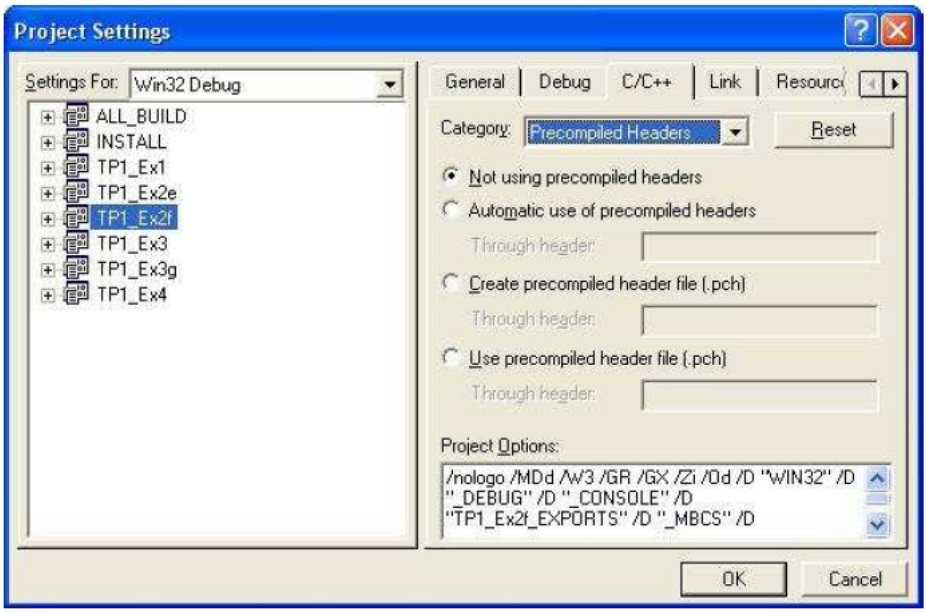

FIGURE 6 – Fenêtre d'option d'un projet afin de supprimer des entêtes précompilées sous Visual Studio C++ 6

2 Mémento de syntaxe en C++

2.1 Syntaxe élémentaire

Dans cette section, nous donnons les principales règles d'écriture d'un programme en C++. En effet, pour qu'un compilateur puisse interpréter un code en C++, il est necessaire de respecter un certain nombre de règles résumées dans les trois paragraphes suivants.

2.1.1 Instructions

Toutes les instructions en C++ se terminent par « ; ». Le saut de ligne « entrée »ne permet pas de signaler la fin d'une instruction. De manière générale, le saut de ligne, les espaces ainsi que la tabulation, peuvent être ajoutés n'importe où, sauf pour :
– l'écriture d'un chiffre (100000 et non 100 000) ;
– le nom des fonctions, opérateurs et variables ;
– les mots clés.
La définition d'un bloc d'instructions (plusieurs instructions) se fait avec des accolades :

```
{
    instruction1;
    instruction2;
    ...;
}
```

La durée de vie (création et destruction) ainsi que la portée (possibilité d'utilisation / d'accès ...) de tout ce qui est défini à l'intérieur d'un bloc est limitée à ce bloc (sauf pour les variables *static* et les variables allouées dynamiquement).

2.1.2 Commentaires

Deux styles de commentaire sont disponibles en C++ :
– « // »permet de mettre en commentaire tout ce qui est situé après ce caractère et jusqu'au prochain saut de ligne ;
– « /* »et « */ »permettent de délimiter un bloc de commentaires (plusieurs lignes en général).
Voici un exemple d'utilisation de commentaires en C++ :

```
int b;     // variable utilisée pour compter des notes
nb = 10;   /* par défaut on connait
              le nombre de notes qui est :
              10... et toutes ces lignes sont en commentaires */
```

2.1.3 Casse et mots réservés

Le compilateur prend en compte la casse (majuscule ou minuscule) pour
– les noms des fichiers ;
– les noms des variables et des fonctions ;
– les mots clés du langage C++ et les directives du préprocesseur.
Il est à noter que les mots clés et les directives du préprocesseur ne peuvent pas être utilisés par le programmeur pour nommer les variables, les objets et fonctions. Les tableaux 1 et 2 fournissent une liste complète des mots clés et directives du préprocesseur.

asm	auto				
break					
case	catch	char	class	const	continue
default	delete	do	double		
else	enum	extern			
float	for	friend			
goto					
if	inline	int			
long					
new					
operator					
private	protected	public			
register	return				
short	signed	sizeof	static	struct	switch
template	this	throw	try	typedef	
unsigned	union				
virtual	void	volatile			
while					

TABLE 1 – Liste des 48 mots clés du langage C++

#define			
#elif	#else	#endif	#error
#if	#ifdef	#ifndef	#include
#line			
#pragma			
#undef			

TABLE 2 – Liste des 13 directives du préprocesseur

Voici un exemple d'utilisation de mots clés et de directives du préprocesseur :

```
// Ces 3 directives sont necessaire a tous fichiers ".h", la
// variable "_MaClasse_H_" doit etre differente pour chaque fichier
#ifndef _MaClasse_H_
#define _MaClasse_H_

#define PI 3.14 // pas de type devant la variable ni de "=" ni de ";"

// le reste du fichier .h
class MaClasse
{
...
};

#endif
```

2.2 Types

Dans cette section, nous donnons les principales notions à connaître sur les types et la déclaration de variables.

2.2.1 Types fondamentaux en C++

Le tableau 3 fournit une liste des principaux types utilisés en programmation C++.

Type	Taille	Intervalle de valeurs	Précision
`bool`	1 octet	*false* ou *true*	-
`char`	1 octet	-128 à 127	-
`unsigned char`	1 octet	0 à 255	-
`short`	2 octets	-32768 à 32767	-
`unsigned short`	2 octets	0 à 65535	-
`int`	4 octets	-2147483648 à 2147483647	-
`unsigned (int)`	4 octets	0 à 4294967295	-
`long`	4 octets	-2147483648 à 2147483647	-
`unsigned (long)`	4 octets	0 à 4294967295	-
`float`	4 octets	$+/- 3.4 * 10^{-38}$ à $3.4 * 10^{38}$	6 chiffres
`double`	8 octets	$+/- 1.7 * 10^{-308}$ à $1.7 * 10^{308}$	15 chiffres
`long double`	10 octets	$+/- 1.2 * 10^{-4932}$ à $1.2 * 10^{4932}$	18 chiffres

TABLE 3 – Types fondamentaux en C++

2.2.2 Déclaration de variables

La syntaxe de déclaration d'une variable en C++ est la suivante :

```
Type variable;
Type var1, var2, var3;
```

Les noms de variables doivent commencer par une lettre ou par « _ ». Les accents sont interdits. La casse (majuscule ou minuscule) est prise en compte (`var1` est différent de `Var1`). En C++, tous les types sont des classes et toutes les classes sont des types, même les classes créées par l'utilisateur. Exemples :

```
int i,j,k;
float Valeur;
```

2.2.3 Déclaration de variables avec affectation

La déclaration d'une variable avec affectatiton d'une valeur (ou initialisation) se fait de la façon suivante :

```
type var1 = 0;
```

qui est équivalent à [1] :

```
type var1;
var1 = 0;
```

Voici une liste d'exemples de déclaration de variables avec affectation :

```
float a=3.002;
unsigned short b=1,c=2;
double un_mille(1.609e3);
```

L'opération d'affectation peut se faire entre des variables de même type

```
int a,b;
a=4;
b=a;  // la valeur de a est copiée dans b: b vaut 4
```

ou entre des variables de type différent (dans ce cas, attention aux pertes lors de la conversion) :

```
float a=3.1415;
char b;
double c;
b = a;     // b vaut 3, un warning est généré à la compilation
c = a;     // c vaut 3.1415, pas de warning ici car la précisison
           // du type double est supérieure à la précision du
           // type float (pas de perte de précision)
```

1. algorithmiquement... pas en terme d'exécution

2.3 Opérateurs

Nous abordons dans cette partie les définitions et les différentes propriétés liées aux opérateurs en C++.

2.3.1 Opérateur unaire, binaire et ternaire

Un opérateur unaire est un opérateur ne possédant qu'un seul opérande. Voici un exemple d'un tel opérateur :

```
int a; // déclaration de a
&a;    // & est un opérateur unaire,
       // il renvoie l'adresse mémoire de a
```

Un opérateur binaire possède deux opérandes (à ne pas confondre avec les opérateurs réalisant des opérations en binaire). Voici un exemple d'un tel opérateur :

```
int a; // déclaration de a
a = 1; // = est un opérateur binaire, il affecte la valeur 1 à a
```

Il n'existe qu'un seul opérateur ternaire, l'opérateur de condition « ? : ». Voici un exemple d'utilisation de cet opérateur :

```
#include <iostream>
int main (void)
{
  int limitation = 50 ;
  int vitesse;
  int nb_point_en_moins;
  // demander la vitesse
  std::cin >> vitesse;
  nb_point_en_moins = (vitesse > 50) ? 3 : 0;
  // affichage
  std::cout <<" Nombre de points en moins : ";
  std::cout << nb_point_en_moins << std::endl;
return 0;
}
```

2.3.2 Priorité

Les opérateurs en C++ sont répartis selon 17 niveaux de priorité. La priorité la plus basse vaut 1 et la priorité la plus haute vaut 17. La règle de priorité des opétateurs est la suivante : si dans une même instruction sont présents des opérateurs avec des niveaux de priorité différents, les opérateurs ayant la plus haute priorité sont exécutés en premier. Par exemple :

```
A op1 B op2 C;
```

si la priorité de *op*2 est supérieure à la priorité de *op*1, on a :

```
A op1 ( B op2 C );
```

De manière générale, l'ajout de parenthèse dans les expressions enlève les ambiguïtés de priorité.

2.3.3 Associativité

L'associativité désigne la direction dans laquelle sont exécutés les opérateurs d'un même niveau de priorité. Par exemple, la multiplication et la division ont les mêmes niveaux de priorité et leur associativité est de gauche à droite :

```
int a = 3 * 4 / 2 * 3;    // a = 18
int b = 3 * 4 / ( 2 * 3); // b = 2
```

De manière générale, l'ajout de parenthèse dans les expressions enlève les ambiguïtés d'associativité. Tableau 2.3.3 fournit une liste détaillée des opérateurs en C++ ainsi que leur niveau de priorité et leur associativité.

Prio.	Opér.	Signification	Associativité	Exemple
17	::	Résolution de portée (unaire)	droite à gauche	
17	::	Résolution de portée (binaire)	droite à gauche	
16	`()`	Parenthèse de	gauche à droite	`MaFonction(x,y)`
16	`()`	Construction d'objet	gauche à droite	`MaClass(x,y)`
16	`[]`	Indexation de tableau	gauche à droite	`Tab[i]`
16	`.`	Sélection de composant	gauche à droite	`objet.methode();`
16	`->`	Sélection de composant	gauche à droite	`this->methode();`
15	()	conversion de type explicite	droite à gauche	`(double) x;` `double(x);`
15	`sizeof`	Taille en octets	droite à gauche	`sizeof(int);`
15	&	Fournit l'adresse mémoire	droite à gauche	$\&a$
15	`*`	Déférentiation	droite à gauche	
15	`~`	Négation binaire	droite à gauche	`~a;`
15	`!`	Négation logique	droite à gauche	`!a;`
15	`+`	Addition (unaire)	droite à gauche	`+a;`
15	`-`	Soustraction (unaire)	droite à gauche	`-a;`
15	`++`	Incrément	droite à gauche	`++a;`
15	`--`	Décrément	droite à gauche	`--a;`
15	`new`	Allocation mémoire	droite à gauche	`double *t=new;`
15	`delete`	Dé-allocation mémoire	droite à gauche	`delete[] t;`
14	`.*`	Sélection de composant		
14	->*	Sélection de composant (pointeur)	gauche à droite	
13	`*`	Multiplication	gauche à droite	`a*b;`
13	`/`	Division	gauche à droite	`a/b;`
13	`%`	Modulo	gauche à droite	`a%b;`
12	`+`	Addition (binaire)	gauche à droite	`a+b;`
12	`-`	Soustraction (binaire)	gauche à droite	`a-b;`

Prio.	Opér.	Signification	Associativité	Exemple
11	>>	Décalage des bits à droite	gauche à droite	`a>>2;`
11	<<	Décalage des bits à gauche	gauche à droite	`a<<2;`
10	>	Test strictement supérieur	gauche à droite	`a>2;`
10	>=	Test supérieur ou égal	gauche à droite	`a>=2;`
10	<	Test strictement inférieur	gauche à droite	`a<2;`
10	<=	Test inférieur ou égal	gauche à droite	`a<=2;`
9	==	Test d'égalité	gauche à droite	`if(choix=='o')`
9	!=	Test de non égalité	gauche à droite	`if(pt!=NULL)`
8	&	ET binaire	gauche à droite	`a=1&3; //1`
7	^	OU exclusif binaire	gauche à droite	`a=1^3; //2`
6	|	OU binaire	gauche à droite	`a=1|2; //3`
5	&&	ET logique	gauche à droite	`if(a=1&&b<3)`
4	||	OU logique	gauche à droite	`if(a=1||b<3)`
3	? :	Opérateur de condition (ternaire)	gauche à droite	
2	=	Affectation simple |	droite à gauche	`a=b=2;`
2	+=	Affectation combinée +	droite à gauche	`a+=2; //a=a+2`
2	-=	Affectation combinée −	droite à gauche	`a-=2; //a=a-2`
2	*=	Affectation combinée ∗	droite à gauche	`a*=2; //a=a*2`
2	/=	Affectation combinée /	droite à gauche	`a/=2; //a=a/2`
2	%=	Affectation combinée %	droite à gauche	`a%=2; //a=a%2`
2	<<=	Affectation combinée <<	droite à gauche	`a<<=2; //a*=4`
2	>>=	Affectation combinée >>	droite à gauche	`a>>=1; //a/=2`
2	&=	Affectation combinée &	droite à gauche	`a&=1;`
2	^=	Affectation combinée ^	droite à gauche	`a^=0xFF;`
2	|=	Affectation combinée |	droite à gauche	`a|=0xFE;`
1	,	Séquence d'expressions	gauche à droite	`int a,b,c;`

TABLE 4 – Liste et propriétés des opérateurs en C++

2.4 Structures conditionnelles

Les conditions servent à comparer (ou évaluer) des variables ou des valeurs retournées par des fonctions.

2.4.1 if / else

La syntaxe d'une condition *if* est la suivante (les crochets signalent des blocs non nécessaires) :

```
if ( test ) // test est une variable booléenne (true ou false)
  {
  ...       // instructions effectuées si test est vrai
  }
[
else        // bloc else: n'est pas obligatoire
  {
  ...       // instructions effectuées si test est faux
  }
]
```

La variable `test` est forcément de type booléen (bool). Après l'instruction `if`, les accolades sont nécessaires si plusieurs instructions sont déclarées. Le bloc `else` n'est pas obligatoire. Il est à noter que la syntaxe `if(a)` est équivalent à `if( a!=0 )`. Voici un exemple d'utilisation de structure `if` en C++ :

```
bool a,b;
if( a==true )  // il s'agit bien de 2 signes "=="
{
   a = false;
   // std::cout permet d'afficher à l'ecran un message
   std::cout << "Initialisation de la variable à false";
}
double moyenne;
if ( moyenne >= 16.0 )
   std::cout << "Mention très bien";
else
    if ( moyenne >= 14.0 )
       std::cout << "Mention bien";
    else
        if ( moyenne >= 12.0 )
           std::cout << "Mention assez bien";
        else
            if ( moyenne >= 10.0 )
```

```
        std::cout << "Mention passable";
    else
        std::cout << "Non admis...";
```

2.4.2 switch / case

L'imbrication de `if` / `else` peut être remplacée par une structure `switch case` quand la variable à tester est de type entier, c'est à dire `char`, `short`, `int`, `long` et `string`. Dans l'exemple précédent, il n'est donc pas possible d'utiliser une telle structure.
La syntaxe d'une condition `switch case` est la suivante (les crochets signalent des blocs non nécessaires) :

```
switch ( variable )
  {
  case constante_1:       // faire attention au ":"
  [ instructions réalisées si variable == constante_1 ]
  [ break ; ]
  case constante_2:
  [ instructions réalisés si variable == constante_2 ]
  [ break ; ]
  ...
  [ default:
    instructions réalisées si aucun des
    cas précédents n'a pas été réalisé]
  }
```

Les blocs d'instructions n'ont pas besoin d'accolade. Généralement chaque bloc se termine par l'instruction `break;` qui saute à la fin de la structure `switch` (l'accolade fermante). Si un bloc `case` ne se termine pas par une instruction `break;`, le compilateur passera au bloc `case` suivant. Voici un exemple d'utilisation de structure `switch case` en C++ :

```
char choix;
// cout permet d'afficher à l'ecran un message
cout << "Etes vous d'accord ? O/N";
// cin permet de récupérer une valeur saisie au clavier
cin >> choix;
switch( choix )
{
case 'o':
case 'O':
    cout << "Vous etes d'accord";
```

```
    break;
case 'n':
case 'N':
    cout << "Vous n'etes pas d'accord";
    break;
default:
    cout << "Répondre par o/n ou O/N";
}
```

2.5 Structures de boucles

Comme en C, les boucles en C++ servent à répéter un ensemble d'instructions. Il existe 3 types de boucle.

2.5.1 for

La syntaxe d'une boucle `for` en C++ est la suivante :

```
for ( [initialisations]; [condition]; [post-instructions] )
  {
  // Instructions de boucle répétées tant que la condition est vraie
  }
```

L'exécution d'une boucle `for` se passe en 3 étapes.

1. Le bloc [initialisations], déclaré en premier, est toujours exécuté une seule fois,
2. Le bloc [condition] est évalué à chaque répétition,
3. Si la condition est vrai, les instructions de boucle puis le bloc [post-instructions] sont exécutés puis l'étape 2 est répétée. Si l'expression est fausse, aucune instruction de boucle ni le bloc [post-instrucitions] ne sont exécutés et la boucle s'arrète.

Les blocs [initialisations] et [post-instructions] peuvent contenir aucune ou plusieurs instructions séparées par des virgules. Le bloc [condition] peut être vide, dans ce cas on a généralement affaire à une boucle sans fin.
Voici deux exemples d'utilisation d'une boucle `for` :

```
int Taille=3, titi=1, toto;

// qu'une instruction de boucle: accolades non obligatoire
  for ( int i=0; i < Taille; i++ )
    cout << i << endl;  //  Afficher i à l'écran
//-------------------------------
for ( int i=0, int j=0; i < Taille; i++, j=2*i, toto=titi )
```

```
{ //exemple plus compliqué...
    titi = i + j;
    cout << i << endl;     //  Afficher i à l'écran
    cout << toto << endl;  //  Afficher toto à l'écran
}
```

2.5.2 while

La syntaxe d'une boucle `while` est la suivante :

```
while ( [condition] )
  {
  // Instructions de boucle répétées tant que la condition est vraie
  }
```

Il est à noter que le test du bloc [condition] se fait en début de boucle. Voici deux exemples d'utilisation d'une boucle `while` en C++ :

```
int i=0, Taille=3, j;
// Résultat identique à la boucle \verb$for$ ci-dessus
while ( i < Taille )
{
    cout << i << endl;  //  Afficher i à l'écran
    i++;
}
//-------------------------------
// Multi conditions
while ( ( i< Taille ) && ( j != 0 ) )
{
    cout << "Ok" << endl;
    j = 2 * i - i * ( Taille - i );
    i++;
}
```

2.5.3 do / while

Contrairement aux boucles `for` et `while`, la boucle `do while` effectue les instructions de boucle avant d'évaluer l'expression d'arrêt (le bloc condition). La syntaxe d'une boucle `do while` est la suivante :

```
do
  {
  // Instructions de boucle répétées tant que la condition est vrai
  }
while ( [condition] )
```

Voici un exemple d'utilisation d'une boucle `do while` en C++ :

```
char choix;
do
{
    cout << "Quel est votre choix: (q pour quitter) ?";
    cin >> choix;
    (...)
}
while ( choix != 'q' );
```

2.6 Flux d'entrée cin et de sortie cout

En C++, la manière la plus efficace pour gérer les entrées/sorties est l'utilisation de flux de données (*stream*). Les flux d'entrée/sortie ne sont pas incluent dans le langage C++. Ils se trouvent dans le *standard stream input/output library* aussi appelé *iostream*. Pour utiliser les flux dans un programme, il faut inclure le fichier d'entête de cette librairie (la librairie est incluse par défaut par le compilateur) :

```
  #include <iostream>
...
using namespace std; // pour ne pas écrire "<std::">
                     // devant chaque fonction/classe de la std
```

Sous Visual Studio 6, la compilateur impose une formulation différente (norme) :

```
  #include <iostream.h>
// pas besoin de using namespace
```

2.6.1 Flux de sortie pour l'affichage : cout

`cout` désigne le flux de sortie standard qui est dirigé par défaut sur l'écran. La syntaxe est la suivante :

```
[std::]cout << donnée_à_afficher [ << autres_données ];
```

Voici des exemples d'utilisation de flux de sortie :

```
cout << "Hello world";
cout << "Hello " << "world"; //idem que précédent
cout  << "Hello world" << endl; //idem que précédent + saut de ligne
int longueur = 12;
cout << "La longueur est de : " << longeur << " metres \n";
double longeur = 12.323;
cout << "La longueur est de : "
```

```
     << longeur
     << " metres
     << endl;
// Affichage en bases différentes
cout << dec << 16 << endl; // base décimale
cout << hex << 16 << endl; // base hexadécimale
cout << oct << 16 << endl; // base octale
cout << 24; // la dernière base utilisée reste active !!!
```

Les 4 dernières lignes provoquent l'affichage suivant :

```
16
10
20
30
```

2.6.2 Flux d'entrée clavier : cin

`cin` est le flux d'entrée standard sur lequel est dirigée par défaut le clavier. La syntaxe est la suivante :

```
[std::]cin >> variable;
cin >> variable1 >> variable2;
```

Voici des exemples d'utilisation de flux d'entrée :

```
#include <iostream.h>
int main()
{
   int age;
   cout << "Entrer votre age ";
   cin >> age;
   cout << "Vous avez " << age << " ans" << endl;
   return 0;
}
```

2.7 Pointeurs / Références

2.7.1 Pointeurs

En langage C et C++, chaque variable est stockée à une (et unique) adresse physique. Un pointeur est une variable contenant l'adresse d'une autre variable. Un pointeur est typé : il pointe vers des variables d'un certain type et ainsi permet de manipuler correctement ces variables. Avant d'être utilisé, un pointeur doit obligatoirement être initialisé à l'adresse d'une variable ou d'un espace mémoire. Il est possible d'obtenir l'adresse d'une variable à partir du caractère `&` :

```
int a;
cout << &a; // affichage de l'adresse de a
```

Les pointeurs ont un grand nombre d'intérêts :
- ils permettent de manipuler de façon simple des données pouvant être importantes : au lieu de passer à une fonction un élément de grande taille, on pourra lui fournir un pointeur vers cet élément,
- ils permettent de manipuler les tableaux (un tableau est en fait un pointeur sur un espace mémoire réservé),
- en C++ (programmation orientée objet) les pointeurs permettent aussi de réaliser des liens entre objets (cf. cours).

Un pointeur est une variable qui doit être définie, précisant le type de variable pointée, de la manière suivante :

```
type * Nom_du_pointeur;
```

Le type de variable pointée peut être aussi bien un type primaire (tel que `int`, `char`...) qu'un type complexe (tel que `struct` ou une classe). La taille des pointeurs, quelque soit le type pointé, est toujours la même. Elle est de 4 octets avec un OS [2] 32 bits (et 8 en 64 bits). Un pointeur est typé. Il est toutefois possible de définir un pointeur sur `void`, c'est-à-dire sur quelque chose qui n'a pas de type prédéfini (`void * toto`). Ce genre de pointeur sert généralement de pointeur de transition, dans une fonction générique, avant un transtypage qui permetra d'accéder aux données pointées. Le polymorphisme (cf. cours) proposé en programmation orientée objet est souvent une alternative au pointeur `void *`.

Pour initialiser un pointeur, il faut utiliser l'opérateur d'affectation « = » suivi de l'opérateur d'adresse `&` auquel est accollé un nom de variable :

```
int a=2;
int *p1;
p1 = &a;  // initialisation du pointeur!
```

2. Système d'Exploitation

Après (et seulement après) avoir déclaré et initialisé un pointeur, il est possible d'accéder au contenu de l'adresse mémoire pointée par le pointeur grâce à l'opérateur `*`. La syntaxe est la suivante :

```
int a=2;
int *p1;
p1=&a;
cout << *p1; // Affiche le contenu pointé par p1, c'est à dire 2
*p1=4;       // Affecte au contenu pointé par p1 la valeur 4
             // A la fin des ces instructions, a=4;
```

Attention à ne pas mélanger la signification des symboles `*` :
– il sert déclarer un pointeur (int *p1),
– il sert à accéder au contenu pointé par un pointeur (*p1),
– il s'agit de l'opérateur de multiplication (mettre *p1 au carré ...)

2.7.2 Références

Par rapport au C, le C++ introduit un nouveau concept : les références. Une référence permet de faire « référence » à des variables. Le concept de référence a été introduit en C++ pour faciliter le passage de paramètre à une fonction, on parle alors de passage par référence. La déclaration d'une référence se fait simplement en intercalant le caractère `&`, entre le type de la variable et son nom :

```
type & Nom_de_la_variable = valeur;
```

Il existe deux contraintes à l'utilisation des références :
– une référence doit obligatoirement être initialisée lors de sa déclaration,
– une référence ne peut pas être déréférencée d'une variable à une autre.
Voici un exemple de déclaration d'une référence suivie d'une **affectation** :

```
int A = 2;
int &refA = A; //initialisatoin obligatoire!
refA++; // maintenant A=3

// déréférence impossible ==> modification de la valeur contenue:
int B = 5;
refA = B; // signifie A = 5 !!!!
refA++;
// on a : A = 6, B = 5, et refA = 6
```

Les références permettent d'alléger la syntaxe du langage C++ vis à vis des pointeurs. Voici une comparaison de 2 codes équivalents, utilisant soit des pointeurs soit des références.

```
/* Code utilisant un pointeur */
 int main()
 {
    int age = 21;
    int *ptAge = &age;
    cout << *ptAge;
    *ptAge = 40;
    cout << *ptAge;
 }
```

```
/* Code utilisant une référence */
 int main()
 {
    int age = 21;
    int &refAge = age;
    cout << refAge;
    refAge = 40;
    cout << refAge;
 }
```

Bien que la syntaxe soit allégée, les références ne peuvent pas remplacer les pointeurs. En effet, contrairement aux pointeurs, les références ne peuvent pas faire référence à un tableau d'éléments et ne peuvent pas faire référence à une nouvelle variable une fois qu'elles ont été initialisées. Ainsi, en fonction des situtations, il sera plus avantageux d'utiliser soit des pointeurs soit des références : c'est un choix de programmation. L'usage le plus commun des références est le passage de paramètre aux fonctions.

2.8 Tableaux

Un tableau est une variable composée de données de même type, stockées de manière contiguë en mémoire (les unes à la suite des autres). Un tableau est donc une suite de cases (espaces mémoires) de même taille. La taille de chacune des cases est conditionnée par le type de donnée que le tableau contient. Les éléments du tableau peuvent être :
– des données de type simple : int, char, float, long, double... ;
– des pointeurs, des tableaux, des structures et des classes

2.8.1 Tableaux statiques

Lorsque la taille du tableau est connue par avance (avant l'exécution du programme), il est possible de définir un tableau de la façon suivante :

```
Type variable_tableau[nb_éléments];
Type variable_tableau[] = {élément1, élément2, élément3...};
```

Voici un exemple d'une telle initialisation :

```
float notes[10];
int premier[] = {4,5,6};  // le nombre d'éléments est 3
```

Les tableaux statiques sont alloués et détruits de façon automatique. Les éléments d'un tableau sont indicés à partir de 0. La lecture ou l'écriture du ième élément d'un tableau se fait de la façon suivante :

```
A=variable_tableau[i]; // lecture du ième élément
variable_tableau[i]=A; // écriture du ième élément
```

Remarque : La norme ISO du C++ interdit l'usage des *variable-size array*.

2.8.2 Tableaux et pointeurs

La syntaxe suivante :

```
int tab[] = {4,5,6};
cout << tab;
```

affiche une adresse. Ceci montre qu'il existe un lien étroit entre les tableaux et les pointeurs. Plus précisemment une variable tableau (dans l'exemple précédent `tab`) renvoie comme valeur l'adresse de la première case du tableau. Ainsi, le code :

```
int tab[] = {4,5,6};
cout << *tab;  // on affiche le contenu pointé par tab, c'est à dire
               // la valeur de la première case du tableau
```

renvoie la valeur de la première case du tableau, ici 4. Dans l'exemple suivant, nous proposons différentes opérations sur les éléments de tableaux :

```
int tab[] = {4,5,6};
cout << &(tab[1]);  // affiche l'adresse de la deuxième case du tableau
cout << tab+1;      // affiche l'adresse de la deuxième case du tableau
cout << tab[2];     // affiche 6
cout << *(tab+2);   // affiche 6
```

2.8.3 Tableaux dynamiques

Lorsque la taille d'un tableau n'est pas connue lors de la compilation, il est possible en C++ d'allouer un espace mémoire à la taille voulue en cours de programme. On parle d'allocation dynamique. Pour faire un tableau dynamique, il faut d'abord créer un pointeur, puis allouer l'espace mémoire souhaité par la commande `new`. A la fin de l'utilisation du tableau, il ne faut pas oublier de libérer l'espace alloué par la commande `delete[]`. Voici un récapitulatif des instructions à utiliser en C++ pour la gestion dynamique de tableaux :

– Déclaration d'un pointeur

```
type *variable_tableau;
```

– Allocation puis affectation

```
variable_tableau = new type[nb_élément];
//ou encore
type *variable_tableau = new type[nb_élément];
```

– Libération de l'espace alloué

```
delete[] variable_tableau;
```

Voici un exemple de comparaison de déclaration, allocation, destruction d'un tableau dynamique en C et en C++

```
/* Allocation dynamique en C  */
{
int Nb = 4;
int *Array = NULL;
// Allocation de la mémoire
Array = malloc(sizeof(int)*Nb);
// libération de la mémoire
free(Array);
// ou plus simple :
// variable-size array
}
```

```
/* Allocation dynamique en C++ */
{
int Nb = 4;
int *Array = NULL;
// Allocation de la mémoire
Array = new int[Nb];
// libération de la mémoire
delete[] Array;
}
```

Cas particulier : tableau dynamique avec un seul élément :

Pour allouer dynamiquement l'espace pour un seul élément, l'instruction :

```
Type *variable_tableau = new type[1];
```

se simplifie :

```
Type *variable_tableau = new type;
```

de même pour la suppression :

```
delete variable_tableau;
```

L'intérêt de l'écriture `Type *variable_tableau = new type;` est de pouvoir initialiser la variable avec une valeur particulière, par exemple :

```
float *tab = new float(1.3);  // un tableau d'une case de type float
                              // est alloué en mémoire et sa valeur
                              // est initialisée à 1.3
```

2.8.4 Tableaux multidimensionnels

Les tableaux multidimensionnels sont des tableaux de tableaux. Par exemple, en dimension 3, un tableau statique sera déclaré de la façon suivante :

```
type volume[L][C][P]; // L,C,P, nb de ligne, colonne et profondeur
```

En dimension 2, un tableau dynamique sera déclaré de la façon suivante :

```
type **matrice;
matrice = new type*[L];
for( int i=0; i<L; i++ )
   matrice[i] = new type[C];
```

et un tableau dynamique sera détruit de la façon suivante :

```
// A partir de l'exemple précédent
for( int i=0; i<L; i++ )
   delete[] matrice[i];
delete[] matrice;
```

2.9 Fonctions

Une fonction est un sous-programme qui permet d'effectuer un ensemble d'instructions par simple appel à cette fonction. Les fonctions permettent d'exécuter dans plusieurs parties du programme une série d'instructions, ce qui permet de simplifier le code et d'obtenir une taille de code minimale. Il est possible de passer des variables aux fonctions. Une fonction peut aussi retourner une valeur (au contraire des procédures, terminologie oblige...). Une fonction peut faire appel à une autre fonction, qui fait elle meme appel à une autre, etc. Une fonction peut aussi faire appel à elle-même, on parle alors de fonction récursive (il ne faut pas oublier de mettre une condition de sortie au risque sinon de ne pas pouvoir arrêter le programme).

2.9.1 Prototype d'une fonction

Comme les variables, avant d'être utilisée, une fonction doit être déclarée. Le prototype (ou déclaration) d'une fonction correspond à cette déclaration. Il s'agit de la description de la fonction, c'est à dire donner son nom , indiquer le type de la valeur renvoyée et les types d'arguments. Cette fonction sera définie (les instructions qu'elle exécute) plus loin dans le programme. On place le prototype des fonctions en début de programme (généralement le plus tôt possible). Cette déclaration permet au compilateur de « vérifier » la validité de la fonction à chaque fois qu'il la rencontre dans le programme. Contrairement à la définition de la fonction, le prototype n'est pas suivi du corps de la fonction (contenant les instructions à exécuter), et ne comprend pas obligatoirement le nom des paramètres (seulement leur type). Une fonction est déclarée de la façon suivante :

```
type_retour NomFonction( [type Arg1, type Arg2,...] );
```

Le prototype est une instruction, il est donc suivi d'un point-virgule. Les entêtes (fichiers .h) contiennent les déclarations des fonctions. Par exemple `#include <math.h>` permet de déclarer toutes les fonctions de la libraire math.

2.9.2 Définition d'une fonction

La définition d'une fonction est l'écriture du programme exécuté par cette fonction. Une fonction est définie par l'ensemble *nom de fonction*, *nombre d'arguments en entrée*, *type des arguments en entrée* et *liste d'instructions* . La définition d'une fonction se fait selon la syntaxe suivante :

```
type_retour NomFonction( [type Arg1, type Arg2,...] )
{
   liste d'instructions;
}
```

Remarques :

– `type_retour` représente le type de valeur que la fonction peut retourner (char, int, float...), il est obligatoire même si la fonction ne renvoie rien [3] ;
– si la fonction ne renvoie aucune valeur, alors `type_retour` est `void` ;
– s'il n'y a pas d'arguments, les parenthèses doivent rester présentes.

De plus, le nom de la fonction suit les mêmes règles que les noms de variables :
– le nom doit commencer par une lettre ;
– un nom de fonction peut comporter des lettres, des chiffres et le caractère _ (les espaces ne sont pas autorisés) ;
– le nom de la fonction, comme celui des variables est sensible à la casse (différenciation entre les minuscules et majuscules)

2.9.3 Appel de fonction

Pour exécuter une fonction, il suffit de lui faire appel en écrivant son nom (en respectant la casse) suivie de parenthèses (éventuellement avec des arguments) :

```
NomFonction(); // ou
NomFonction1( Arg1, Arg2,... );
```

2.9.4 Passage d'arguments à une fonction

Il existe trois types de passage de paramètres à une fonction : le passage par valeur, par pointeur et par référence. Ces passages sont décrits ci-dessous.

Passage par valeur

Les arguments passés à la fonction sont copiés dans de nouvelles variables propres à la fonction. Les modifications des valeurs des arguments ne sont donc pas propagés au niveau de la portée de la fonction appelante. Ce type de passage de paramètre par valeur est aussi appelé passage par copie. Voici la syntaxe de passage par valeur :

– Déclaration :

```
type_retour NomFonction( [type Arg1, type Arg2,...] );
```

– Définition :

```
type_retour NomFonction( [type Arg1, type Arg2,...] )
{
   // instructions de la fonction
   // pouvant utiliser les valeurs des paramètres
   // et déclarer d'autres variables
   return variable_de_type_retour;
}
```

3. excepetion faite des constructeurs et destructeurs

Voici un exemple d'utilisation de telles fonctions :

```
void Affichage(int a, int b)
{
  cout<<"Valeur 1 = "<<a<<endl;
  cout<<"Valeur 2 = "<<b<<endl;
}

int Somme(int a, int b)
{
   int c = a + b;
   return c;
}
```

```
// suite du programme
int main()
{
   int nb1=2, nb2=3, nb3=4;
   int somme;
   Affichage( nb1, nb2);
   Affichage( Somme(nb1,nb2),3);
   somme = Somme( nb2, nb3);
   return 0;
}
```

Passage par pointeur

Dans le cas où l'on souhaite modifier dans la fonction une variable passée en paramètre, il est nécessaire d'utiliser en C++ soit un passage par pointeur soit un passage par référence. Dans le cas du passage par pointeur, les variables passées en arguments de la fonction correspondent à des adresses de variables (pointeur ou &variable). Voici la syntaxe du passage par pointeur :

– Déclaration :

```
type_retour NomFonction( [type *Arg1, type *Arg2,...] );
```

– Définition :

```
type_retour NomFonction( [type *Arg1, type *Arg2,...] )
{
   // instructions de la fonction
   // les variables passées par pointeur s'utilisent
   // comme les autres variables
   return variable_de_type_retour;
}
```

Voici un exemple d'utilisation :

```
 void CopieTab(double *origine,
               double *copie,
               int taille)
{
   for (int i=0; i<taille, i++)
       copie[i] = origine[i];
}

void ModifTailleTab(double **tab,
                    int taille)
{
   // suppression du tableau
   if ( *(tab) != NULL )
      delete[] *(tab);
   *(tab) = new double[taille];
   // il faut définir **tab pour
   // que l'allocation dynamique
   // soit prise en compte au
   // niveau de la fonction
   // appelante
}
```

```
// suite du programme
int main()
{
   int dim=20;
   double tab1[20];
   double tab2[20];
   double *p = NULL;
   // Initialisation
   // de tab2
   CopieTab(tab1,tab2,dim);
   // Allocation
   // dynamique de p
   ModifTailleTab(&p,dim);
   delete[] p;
   return 0;
}
```

Passage par référence

Un paramètre passé par référence n'est pas copié en local pour la fonction : l'adresse mémoire du paramètre (sa référence) est passée à la fonction. Pour l'utilisateur de la fonction ainsi que pour le programmeur, ce processus est complétement transparent et est syntaxiquement identique au passage par valeur (à un `&` près...). Le passage par référence est utile pour :
– modifier et préserver les changements d'un paramètre passé en argument ;
– gagner du temps en ne copiant pas le paramètre pour la fonction ;
– ne pas exécuter le constructeur de copie de l'objet passé.
Voici la syntaxe du passage par référence :

– Déclaration :

```
type_retour NomFonction( [type &Arg1, type &Arg2,...] );
```

– Définition :

```
type_retour NomFonction( [type &Arg1, type &Arg2,...] )
```

```
{
   // instructions de la fonction
   // les variables passées par référence s'utilisent
   // comme les autres variables
   return variable_de_type_retour;
}
```

Voici un exemple de comparaison d'utilisation de fonctions avec passage par pointeur ou avec passage par référence.

```
/* Passage par pointeur  */
struct Point
{
   int x,y;
};

void remiseAzero(Point *ptNew)
{
   ptNew->x = 0;
   ptNew->y = 0;
}

int main()
{
   Point pt;
   remiseAzero(&pt);
   return 0;
}
```

```
/* Passage par référence */
struct Point
{
   int x,y;
};

void remiseAzero(Point &ptNew)
{
   ptNew.x = 0;
   ptNew.y = 0;
}

int main()
{
   Point pt;
   remiseAzero(pt);
   return 0;
}
```

2.9.5 Surcharge de fonction

Une des nouveautés intéressantes du C++ par rapport au C est la possibilité de nommer plusieurs fonctions avec le même nom, à condition que celles-ci aient leurs arguments différents (en type et/ou en nombre). Ce principe est appelé surcharge de fonction. Le nom de la fonction n'est donc plus limité à un type de paramètre et il est possible de définir une fonction réalisant des opérations différentes en fonction des arguments qui lui sont passés.

3 Programmation Orientée Objet en C++

Le langage C est un langage procédural, c'est-à-dire un langage permettant de définir des données grâce à des variables, et des traitements grâce aux fonctions. L'apport principal du langage C++ par rapport au langage C est l'intégration du concept d'objet, afin d'en faire un langage orienté objet.

3.1 Diagramme UML

L'UML (*Unified Modeling Language*, que l'on peut traduire par « langage de modélisation unifié ») est une notation permettant de modéliser une application sous forme de concept (les **classes**) et d'interaction entre les instances de ces concepts (les objets) . Cette modélisation consiste à créer une réprésentation des éléments du monde réel auxquels on s'intéresse, sans se préoccuper de leur réalisation (ou implémentation en informatique). Ainsi en informatique cette modélisation est indépendante du langage de programmation. Modéliser une application n'est pas évident. La première difficutlé est de déterminer les objets (et les concepts) présents dans l'application (une sorte de découpage données/fonctions de l'application) puis les interactions qu'ils existent entre les différents concepts. Ensuite il faut passer à la description précise de chaque concept : décrire les données et les fonctions qu'il utilise pour fonctionner correctement.

3.1.1 UML et C++

Le C++ étant un langage orienté objet, l'UML permet bien évidemment de décrire des objets qui seront implémentés en C++. Cependant, le langage C++ possède quelques mécanismes et fonctionnalités particuliers au langage qui ne sont pas forcément décrits en conception UML. Il s'agit notamment :
- la construction et destruction des objets ;
- les processus de copie d'objet (constructeur de copie et opérateur d'affectation) ;
- les règles d'héritage (en UML `public` uniquement, le C++ ajoute `protected` et `private`).

Pour les deux premiers points, le compilateur C++ ajoute automatiquement des méthodes par défaut si celle-ci ne sont pas déjà présentes :
- le constructeur de copie ;
- le constructeur par défaut (sans argument) ;
- l'opérateur d'affecation (ou de copie) `=` ;
- le destructeur (par défaut déclaré avec le mot clé `virtual`).

Il arrive très souvent que ces méthodes par défaut soient utilisées par l'application alors que la conception UML ne les faisait pas apparaître. Ainsi, passer de la conception UML à une implémentation C++ nécessitera des ajustements de la conception. Afin d'illustrer ce paragraphe, voici une comparaison de deux diagrammes UML (l'un naturel, l'autre

correspondant aux ajouts du C++) d'une même classe `Point` permettant de représenter et de manipuler (addition) des points du plan.

Point
- X :double - Y :double
+ GetX() :double + GetY() :double + SetX(x :double) :void + SetY(y :double) :void + Add(pt : Point) :Point + Afficher() :void

Point
- X :double - Y :double
+ GetX() :double + GetY() :double + SetX(x :double) :void + SetY(y :double) :void + Add(pt : Point) :Point + Afficher() :void `<<create>>` + Point(pt :Point) `<<create>>` + Point() + operator=(pt :Point) :Point `<<destroy>>` + Point()

TABLE 5 – Diagramme de classe UML (à gauche) et diagramme complet de l'implémentation C++ (à droite). Les trois dernières méthodes du deuxième diagramme sont automatiquement créées en implémentation C++.

La déclaration en C++ de la classe `Point` du diagramme UML (TAB. 5) est la suivante :

```
class Point
{
private:
    double X;
    double Y;
public:
    Point Add(const Point &pt);
    void Afficher();
    double GetX();
    void SetX(const double &x);
    double GetY();
    void SetY(const double &y);
};
```

3.2 Concept de classe

On appelle *classe* la structure d'un objet, c'est-à-dire la déclaration de l'ensemble des entités qui composeront un objet. Un objet est donc « issu » d'une classe. Plus rigoureusement, on dit qu'un objet est une instance (une réalisation) d'une classe. Pour désigner un objet, on parle aussi d'instance ou d'occurrence.

3.2.1 Définition de classe

Avant de manipuler un objet, il faut définir la classe dont il provient, c'est-à-dire décrire de quoi est composé la classe : fonctions et données (cf. paradigme d'encapsulation du cours). La définition d'une classe s'effectue dans un fichier d'entête (header) dont l'extension commence généralement par un « h »(.h, .hxx). Le nom de ce fichier est souvent le même que celui de la classe qu'il défini. Cette définition en C++ se fait de la manière suivante (les crochets signalent des blocs non nécessaires) :

```
class NomClasse [ : type_héritage NomClasseHéritée]
{
[ public:
     // définition des champs et méthodes publics
               ]
  [   NomClasse();  ] // constructeur
  [   ~NomClasse(); ] // Destructeur
[ protected:
     // définition des champs et méthodes protégés
]
[ private:
     // définition des champs et méthodes privées
]

};
```

Le mot clé `type_héritage` permet de définir un type d'héritage. Il existe trois types d'héritage possibles : `public`, `protected` ou `private` (cf. paragraphe 3.4). Le point virgule situé à la fin du bloc de définition de la classe est obligatoire.

3.2.2 Visibilité et accesseurs

L'utilisateur d'une classe n'a pas forcément besoin de savoir de quelle façon sont structurées les données dans l'objet. Cela signifie qu'un utilisateur n'est pas obligé de connaitre les détails de l'implémentation des méthodes. Ainsi, en interdisant l'utilisateur de modifier directement les données, il est obligé d'utiliser les fonctions définies pour les modifier. Si ces fonctions sont bien faites, il est alors possible de garantir l'intégrité des données et un fonctionnement cohérent de la classe. Les niveaux de visibilité permettent de restreindre l'accès à chaque donnée (ou fonction) suivant que l'on accède à cette donnée par une méthode de la classe elle-même, par une classe fille (ou dérivée), ou bien d'une classe ou fonction quelconque. En C++ il existe 3 niveaux de visibilité des éléments de la classe. :

– `public` : Aucune restriction. Il s'agit du plus bas niveau de restriction d'accès, toutes les fonctions membres (de n'importe quelle classe) peuvent accéder aux champs ou aux méthodes public,

– `protected` : l'accès aux variables et fonctions est limité aux fonctions membres de la classe et aux fonctions des classes filles,
– `private` : Restriction d'accès le plus élevé. L'accès aux données et fonctions membres est limité aux méthodes de la classe elle-même.

3.2.3 Données membres

Les données membres (ou attributs ou champs) sont les variables stockées au sein d'une classe. Elles doivent être précédées de leur type et d'un champs précisant leur portée, c'est-à-dire leur niveau de protection (`public`, `protected` ou `private`). Il est recommandé d'utiliser des champs `private` ou `protected`. Ainsi l'accès à ces données ne peut se faire qu'au moyen de fonctions membres (ou méthodes) qui s'assurent de la bonne utilisation de ces variables. Il est à noter que tout objet peut être utilisée comme un attribut de la classe, sauf un objet du type de la classe en construction (dans ce cas il faut passer par un pointeur) ou un objet de la hierarchie avale (car les objets fils n'existent pas encore). Voici un exemple de déclaration d'une classe `Point` avec des attributs ayant différents niveaux de protection :

```
class Exemple
{
    double X;  // par défaut un attribut est de champ private
    double Y;
protected:
    char *Txt;
public:
    double Val;
private:
    Exemple *A;  // champs privé, pointeur sur Exemple
};
```

3.2.4 Fonctions membres

Les données membres permettent de conserver des informations relatives à la classe, tandis que les fonctions membres (ou méthodes) représentent les traitements qu'il est possible de réaliser avec les objets de la classe. On parle généralement de fonctions membres ou méthodes pour désigner ces traitements. Il existe deux façons de définir des fonctions membres :
– en définissant la fonction (prototype et corps) à l'intérieur de la classe en une opération ;
– en déclarant la fonction à l'intérieur de la classe et en la définissant à l'extérieur (généralement dans un autre fichier).
Pour des questions de lisibilité, la seconde solution est préférée. Ainsi, puisque l'on définit la fonction membre à l'extérieur de sa classe, il est nécessaire de préciser à quelle classe cette dernière fait partie. On utilise pour cela l'opérateur de résolution de portée,

noté ::. A gauche de l'opérateur de portée figure le nom de la classe, à sa droite le nom de la fonction. Cet opérateur sert à lever toute ambiguïté par exemple si deux classes ont des fonctions membres portant le même nom. Voici un exemple de déclaration d'une classe `Valeur` avec attribut et méthodes :

```
//fichier Valeur.h
/* Déclaration des méthodes
à l'intérieur de la classe */
class Valeur
{
private:
    double X;
public:
    void SetX(double);
    double GetX();
};
```

```
// fichier Valeur.cpp
/* Définition des méthodes à
l'extèrieur de la classe */
void Valeur::SetX(double a)
{
  X = a;
}

double Valeur::GetX()
{
  return X;
}
```

3.2.5 Objets

En C++, il existe deux façons de créer des objets, c'est-à-dire d'instancier une classe :
– de façon statique,
– de façon dynamque.

Création statique
La création statique d'objets consiste à créer un objet en lui affectant un nom, de la même façon qu'une variable :

```
NomClasse NomObjet;
```

Ainsi l'objet est accessible à partir de son nom. L'accès à partir d'un objet à un attribut ou une méthode de la classe instantiée se fait grâce à l'opérateur « . » :

```
// Fichier Valeur.h
class Valeur
{
private:
    double X;
public:
    void SetX(double a)
    { X = a; }
    double GetX()
    { return X; }
};
```

```
#include "Valeur.h"
#include <iostream.h>

int main()
{
  Valeur a;
  a.SetX(3);
  cout << "a.X=" << a.GetX();
  return 0;
}
```

Création dynamique

La création d'objet dynamique se fait de la façon suivante :
– définition d'un pointeur du type de la classe à pointer ;
– création de l'objet « dynamique » grâce au mot clé `new`, renvoyant l'adresse de l'objet nouvellement créé ;
– affecter cette adresse au pointeur.

Voici la syntaxe à utiliser pour la création d'un objet dynamique en C++ :

```
NomClasse *NomObjet;
NomObjet = new NomClasse;
```

ou directement

```
NomClasse *NomObjet = new NomClasse;
```

Tout objet créé dynamiquement devra impérativement être détruit à la fin de son utilisation grâce au mot clé `delete`. Dans le cas contraire, une partie de la mémoire ne sera pas libérée à la fin de l'exécution du programme. L'accès à partir d'un objet créé dynamquement à un attribut ou une méthode de la classe instantiée se fait grâce à l'opérateur « `->` »(il remplace (*pointeur).Methode()) :

```
// Fichier Valeur.h
class Valeur
{
private:
    double X;
public:
    void SetX(double a)
    { X = a; }
    double GetX()
    { return X; }
};
```

```
#include "Valeur.h"

int main()
{
  Valeur *a = new Valeur;
  a->SetX(3); // ou (*a).SetX(3);
  delete a;
  return 0;
}
```

3.2.6 Surcharge de méthodes

Toute méthode peut être surchargée[4]. On parle de surcharge de méthode lorsque l'on déclare au sein d'une classe une méthode ayant le même nom qu'une autre méthode. La différence entre les deux méthodes (ou plus) réside dans les arguments de ces méthodes : chaque fonction doit avoir une particularité dans ces arguments (type, nombre ou protection constante) qui permettra au compilateur de choisir la méthode à exécuter. Le type de retour d'une fonction ne constitue pas une différence suffisante pour que le compilateur fasse la différence entre plusieurs fonctions. Voici un exemple de surcharge de méthode :

```
class Point
{
private:
    double X;
    double Y;
public:
    void SetX(int);           // définition d'une fonction
    void SetX(int,int);       // bonne surcharge
    void SetX(double);        // bonne surcharge
    double SetX(double);      // mauvaise surcharge
    void Set(Point &a);       // bonne surcharge
    void Set(const Point &a); // bonne surcharge
};
```

3.3 Méthodes particulières

3.3.1 Constructeurs

Le constructeur est la fonction membre appelée automatiquement lors de la création d'un objet (en statique ou en dynamique). Cette méthode est la première fonction membre à être exécutée. Elle est utilisée généralement pour initialiser les attributs de la classe. Un constructeur possède les propriétés suivantes :
– un constructeur porte le même nom que la classe dans laquelle il est défini ;
– un constructeur n'a pas de type de retour (même pas `void`) ;
– un constructeur peut avoir des arguments.
La définition de cette fonction membre spéciale n'est pas obligatoire (si vous ne souhaitez pas initialiser les données membres par exemple) dans la mesure où un constructeur par défaut (appelé parfois constructeur sans argument) est défini par le compilateur C++ si la classe n'en possède pas. Voici un exemple d'utilisation d'un constructeur pour initialiser les attributs d'une classe :

4. Il n'existe qu'une seule méthode qui ne peut être surchargée : le destructeur. Cette méthode ne possède pas d'arguments... assez logique, non ?

```
class Point
{
private:
    double X;
    double Y;
public:
    Point(double a=0,double b=0);  // valeurs par défaut {0,0}
};

Point::Point(double a, double b)
{
   X = a;
   Y = b;
}
```

Comme pour n'importe quelle fonction membre, il est possible de surcharger les constructeurs, c'est-à-dire définir plusieurs constructeurs avec un nombre/type d'arguments différents. Ainsi, il sera possible d'initialiser différemment un même objet, selon la méthode de construction utilisée.

3.3.2 Constructeur de copie

Le but de ce constructeur est d'initialiser un objet lors de son instanciation à partir d'un autre objet appartenant à la même classe. Toute classe dispose d'un constructeur de copie par défaut (et naïf) généré automatiquement par le compilateur. Naïf car il copie une à une les valeurs des champs de l'objet à copier dans les champs de l'objet à instancier. Toutefois, ce constructeur par défaut ne conviendra pas toujours, et le programmeur devra parfois en fournir un explicitement, typiquement quand la classe possède des champs dynamiques. La définition du constructeur de copie se fait comme celle d'un constructeur normal. Le nom doit être celui de la classe, et il ne doit y avoir aucun type de retour. Dans la liste des paramètres cependant, il devra toujours y avoir une référence sur l'objet à copier. Voici un exemple d'utilisation d'un tel constructeur :

```
class Valeur
{
private:
    double X;
public:
    // constructeur
    Valeur(double a);
    { X = a; }
    // constructeur de copie
    Valeur(Point &pt)
    { X = pt.X; }
};
```

```
#include "Valeur.h"
#include <iostream.h>

int main()
{
  // constructeur
  Valeur v1(2.3);
  // constructeur par copie
  Valeur v2(v1);
  // constructeur par copie
  Valeur v3 = v2;
  return 0;
}
```

3.3.3 Destructeurs

Le destructeur est une fonction membre qui s'exécute automatiquement lors de la destruction d'un objet (pas besoin d'un appel explicite à cette fonction pour détruire un objet). Cette méthode permet d'éxécuter des instructions nécessaire à la destruction de l'objet. Cette méthode possède les propriétés suivantes :
– un destructeur porte le même nom que la classe dans laquelle il est défini et est précédé d'un tilde ~ ;
– un destructeur n'a pas de type de retour (même pas `void`) ;
– un destructeur n'a pas d'argument ;
– la définition d'un destructeur n'est pas obligatoire lorsque celui-ci n'est pas nécessaire.
Le destructeur, comme dans le cas du constructeur, est appelé différemment selon que l'objet auquel il appartient a été créé de façon statique ou dynamique.
– le destructeur d'un objet créé de façon statique est appelé de façon implicite dès que le programme quitte la portée dans lequel l'objet existe ;
– le destructeur d'un objet créé de façon dynamique sera appelée via le mot clé delete.
Il est à noter qu'un destructeur ne peut être surchargé, ni avoir de valeur par défaut pour ses arguments, puisqu'il n'en a tout simplement pas. Voici un exemple de destructeur :

```
class Point
{
private:
    double X;
    double Y;
    double *tab; // champs dynamique !!!
public:
    Point(double a=0,double b=0);  // constructeur
    ~Point();                      // destructeur
```

```
};

Point::Point(double a, double b)
{
   X = a;
   Y = b;
   tab = new double[2];
}

Point::~Point()
{
   delete[] tab;
}
```

3.3.4 Opérateurs

Par convention, un opérateur `op` reliant deux opérandes `a` et `b` est vu comme une fonction prenant en argument un objet de type `a` et un objet de type `b`. Ainsi, l'écriture `a op b` est traduite en langage C++ par un appel `op(a,b)`. Par convention également, un opérateur renvoie un objet du type de ses opérandes. Cela permet pour de nombreux opérateurs de pouvoir intervenir dans une chaîne d'opérations (par ex. `a=b+c+d`).

La fonction C++ correspondant à un opérateur est réprésentée par un nom composé du mot operator suivi de l'opérateur (ex : `+` s'appelle en réalité `operator+()`).

Un opérateur peut être définit en langage C++ comme une fonction membre, ou comme une fonction amie indépendante. Ici, nous nous limiterons au cas des fonctions membres. Tableau (6) donne une liste de l'ensemble des opérateurs qui peuvent être surchargés en C++ :

+	-	*	/	%	^	&
|	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	|=
<<	>>	>>=	<<=	==	!=	<=
>=	&&	||	++	--	->*, ,	
->	[]	()	new	delete	type()	

TABLE 6 – Liste des opérateurs que l'on peut surcharger en C++

Surcharge d'opérateur avec une fonction membre

Le langage C++ étend la notion de surcharge de fonctions aux opérateurs du langage. Dans ce type de surcharge d'opérateur, le premier opérande de notre opérateur, correspondant au premier argument de la fonction, va se trouver transmis implicitement. Ainsi l'expression `a op b` sera interprétée par le compilateur comme `a.operator op(b)` (exemple : `a+b;` est en fait `a.operator+(b);`). Le prototype de surcharge d'opérateur avec une fonction membre est le suivant (les crochets signalent des blocs non nécessaires) :

```
NomClasse operator SymboleOpérateur( NomClasse );
```

Voici un exemple de surcharge de l'opérateur `+` :

```
class Valeur
{
private:
    double X;
public:
    Valeur(double a=0) { X=a; }
    Valeur operator+( Valeur pt)
    {
       Valeur result;
       result.X = X + pt.X;
       return result;
    }
};
```

```
#include "Valeur.h"

int main()
{
  Valeur a(2);
  Valeur b(3);
  // surcharge de l'opérateur
  Valeur c = a + b;
  return 0;
}
```

Transmission par référence

Dans le paragraphe précédent, nous avons présenté comment effectuer une surcharge d'opérateur avec une transmission d'arguments et de valeur de retour par valeur. Dans le cas de manipulation d'objet de grande taille, il sera préférable d'utiliser une transmission d'arguments (voir de valeur de retour) par référence. Le prototype de surcharge d'opérateur avec une fonction membre, avec une transmission par référence, est le suivant (les crochets signalent des blocs non nécessaires) :

```
NomClasse operator SymboleOpérateur( [const] NomClasse & );
```

Voici un exemple de surcharge de l'opérateur + :

```
class Valeur
{
private:
    double X;
public:
    Valeur(double a=0) { X=a; }
    Valeur operator+(
     const Valeur &pt)
    {
       Valeur result;
       result.X = X + pt.X;
       return result;
    }
};
```

```
#include "Valeur.h"

int main()
{
  Valeur a(2);
  Valeur b(3);
  // surcharge de l'opérateur
  Valeur c;
  c = a + b;
  return 0;
}
```

3.4 Héritage

L'héritage est un principe propre à la programmation orientée objet, permettant de créer une nouvelle classe à partir d'une classe existante. Le nom d'héritage (pouvant parfois être appelé dérivation de classe) provient du fait que la classe dérivée (la classe nouvellement créée) contient les attributs et les méthodes de sa classe mère (la classe dont elle dérive). L'intérêt majeur de l'héritage est de pouvoir définir de nouveaux attributs et de nouvelles méthodes pour la classe dérivée, qui viennent s'ajouter à ceux et celles héritées.

Par ce moyen, une hiérarchie de classes de plus en plus spécialisées est créée. Cela a comme avantage majeur de ne pas avoir à repartir de zéro lorsque l'on veut spécialiser une classe existante. De cette manière il est possible de récupérer des librairies de classes, qui constituent une base, pouvant être spécialisées à loisir.

3.4.1 Type d'héritage

En C++, il existe trois types d'héritage définis par les mots clés `public`, `protected`, `private`. Un type d'héritage définie les accès possibles qu'une classe dérivée a vis à vis des attributs et méthodes de la classe de base. Le prototype de l'héritage se fait à la définition de la classe :

```
class NomClasseDérivée : TypeHériatage NomClasseDeBase
{
// definition des attributs et méthodes
// de la classe dérivée
...
};
```

Héritage public

L'héritage public se fait en C++ à partir du mot clé `public`. C'est la forme la plus courante de dérivation. Les principales propriétés liées à ce type de dérivation sont les suivantes :

- les membres (attributs et méthodes) publics de la classe de base sont accessibles par « tout le monde », c'est à dire à la fois aux fonctions membres de la classe dérivée et aux utilisateurs de la classe dérivée ;
- les membres protégés de la classe de base sont accessibles aux fonctions membres de la classe dérivée, mais pas aux utilisateurs de la classe dérivée ;
- les membres privées de la classe de base sont inaccessibles à la fois aux fonctions membres de la classe dérivée et aux utilisateurs de la classe dérivée.

De plus, tous les membres de la classe de base conservent dans la classe dérivée le statut qu'ils avaient dans la classe de base. Cette remarque n'intervient qu'en cas de dérivation

d'une nouvelle classe à partir de la classe dérivée.

Héritage privé

L'héritage privé se fait en C++ à partir du mot clé `private`. Les principales propriétés liées à ce type de dérivation sont les suivantes :

– les membres (attributs et méthodes) publics de la classe de base sont inaccessibles à la fois aux fonctions membres de la classe dérivée et aux utilisateurs de cette classe dérivée ;
– les membres protégés de la classe de base restent accessibles aux fonctions membres de la classe dérivée mais pas aux utilisateurs de cette classe dérivée. Cependant, ils seront considéré comme privés lors de dérivation ultérieures.

Il est cependant possible dans une dérivation privée de laisser public un membre de la classe de base en le redéclarant explicitement public.

Héritage protégé

L'héritage protégé se fait en C++ à partir du mot clé `protected`. Cette dérivation est intermédiaire entre la dérivation publique et la dérivation privée. Ce type de dérivation possède la propriété suivante :

– les membres (attributs et méthodes) publics de la classe de base seront considérés comme protégés lors de dérivation ultérieures.

Récapitulation

Le tableau 7 récapitule toutes les propriétés liées aux différents types de dérivation.

Classe de base	Dérivée publique		Dérivée protégée		Dérivée privée	
Statut initial	Nouveau statut	Accès utilisateur	Nouveau statut	Accès utilisateur	Nouveau statut	Accès utilisateur
public	public	O	protégé	N	privé	N
protégé	protégé	N	protégé	N	privé	N
privé	privé	N	privé	N	privé	N

TABLE 7 – Les différents types de dérivation en C++

Voici un exemple d'utilisation d'un héritage public :

```
//fichier Point.h

  class Point
  {
  protected:
      float X,Y;
  public:
      Point(float a, float b)
      {
        X=a;
        Y=a;
      }
      void SetX(double a) {X=a;}
      void SetY(double b) {Y=b;}
  };
```

```
//fichier PointColor.h

  class PointColor : public Point
  {
  private:
      short couleur;
  public:
      void SetCouleur(short a)
      { couleur=a; }
      void Afficher()
      {
         cout << "La couleur en ";
         cout << X << " " << Y;
         cout << " = " << couleur;
      }
  };
```

```
  #include "PointColor.h"
  #include <iostream.h>

  int main()
  {
    PointColor pt;     // Objet instantié

    pt.setX(3.4);      // Utilisation d'une méthode
    pt.SetY(2.6);      // de la classe de base

    pt.SetCouleur(1);  // Utilisation de méthodes
    pt.Afficher();     // de la classe dérivée

    return 0;
  }
```

3.4.2 Appel des constructeurs et des destructeurs

Hiérarchisation des appels

Soit l'exemple suivant :

```
class A
{   .....
public:
    A()
    ~A()
    .....
};
```

```
class B : public A
{   .....
public:
    B()
    ~B()
    .....
};
```

Pour créer un objet de type B, il faut tout d'abord créer un objet de type A, donc faire appel au constructeur de A, puis le compléter par ce qui est spécifique à B en faisant appel au constructeur de B. Ce mécanisme est pris en charge par le C++ : il n'y aura pas à prévoir dans le constructeur de B l'appel au constructeur de A.

La même démarche s'applique aux destructeurs : lors de la destruction d'un objet de type B, il y aura automatiquement appel du destructeur de B, puis appel de celui de A. Il est important de noter que les destructeurs sont appelés dans l'ordre inverse de l'appel des constructeurs.

Transmission d'informations entre constructeurs

En C++, il est possible de sécifier, dans la définition d'un constructeur d'une classe dérivée, les informations que l'on souhaite transmettre au constructeur de la classe de base. Voici un exemple d'héritage avec transmission d'informations entre constructeurs :

```
class Point
{
protected:
    float X,Y;
public:
    Point(float a, float b)
    {
      X=a;
      Y=a;
    }
};
```

```
class PointColor : public Point
{
private:
    short couleur;
public:
    PointColor(float a, float b,
       short c) : Point(a,b)
    {
      couleur=c;
    }
};
```