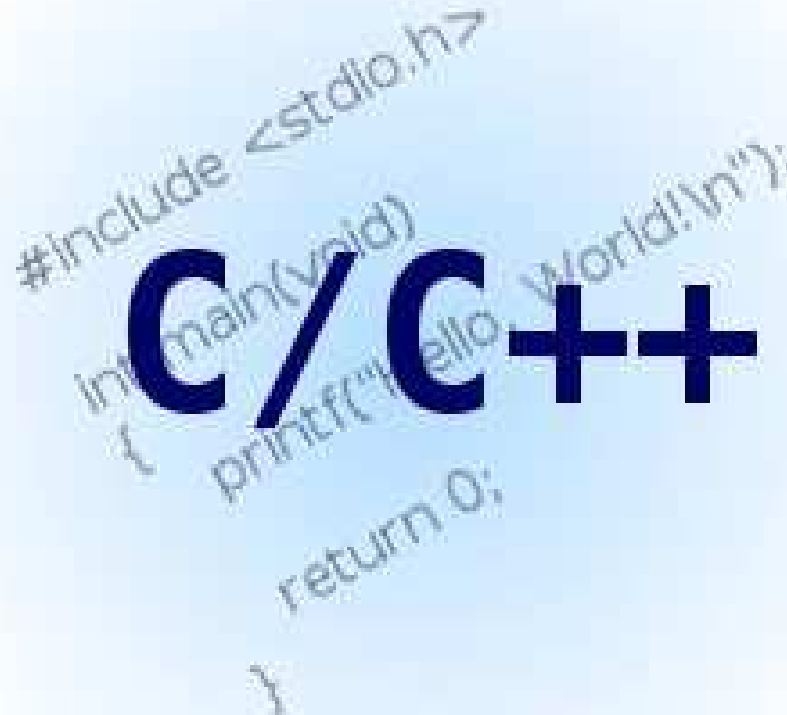


Programmation Orientée Objet


The logo for C/C++ is displayed in a large, bold, dark blue font. The 'C' is followed by a diagonal slash, another 'C', and then two plus signs. In the background, there is a faint, light blue circular graphic containing C code snippets: '#include <stdio.h>', 'int main(void)', '{', 'printf("Hello, World!\n");', 'return 0;', and '}'.

Sommaire

1. Généralités
2. La conception orientée objet
3. Un meilleur C
4. Les fonctions
5. Compilation
6. La notion d'objet en C++ (Classe)
7. Organisation d'un développement en C++
8. Héritage
9. La généricité
10. C++ 11 : Standard ANSI/ISO

1. Généralités

1.1 *Objectifs du cours*

- C++ et Java sont actuellement les deux langages à objets les plus utilisés dans l'industrie et doivent donc être maîtrisés par tout développeur. Les objectifs de ce cours sont :
- Apprendre les concepts et principes de base de la programmation et conception orientée objet.
- Apprendre à programmer en C++.
- Se faire une idée de l'utilisation de la Programmation Orientée Objet (POO) pour améliorer la fiabilité et la réutilisation des logiciels.
-  Pour pouvoir suivre avec profit ce cours sur le langage C++, il est indispensable de bien connaître le langage C. Les pages qui suivent ne présentent que les aspects nouveaux du langage C++ par rapport au langage C (ANSI).

1. Généralités

1.2 Historique

La programmation par objet est apparue dans le courant des années 60-70. Le premier véritable langage en la matière fut SMALLTALK. La programmation par objet définit ce qu'on appelle les « langages de quatrième génération »

- La première génération : ASSEMBLEUR, FORTRAN I, ...
 - *Zone de données commune et pas de sous-programme.*
- La deuxième génération : FORTRAN II, ALGOL 60, COBOL, ...
 - *Introduction de la notion de sous-programme et de données typées.*
- La troisième génération : ALGOL 68, PL/1, Pascal, C, ...
 - *Introduction de la notion de module (unité) et de compilation séparée.*
- La quatrième génération : ADA, Pascal Objet, C++, ...
 - *Introduction des concepts objets.*

1. Généralités

1.2 Historique

- Le langage de programmation C++ a été inventé par Bjarne Stroustrup au début des années 80.
- Lien : *<http://www.research.att.com/~bs/C++.html>*
- Le premier compilateur a été développé par les laboratoires AT&T.
- Le C++ est une extension du langage C dont il étend les possibilités et lui ajoute les caractéristiques d'un langage orienté objet.

1. Généralités

1.3 Bibliographie

Claude Delannoy Programmer en langage C++ Eyrolles

Philippe Laroque Exercices en langage C++ Eyrolles

Bjarne Stroustrup The C++ Programming Language (3rd Edition)
Mise à jour avec le standard ANSI/ISO



Stanley B. Lippman L'essentiel du C++ 2ème édition Addison Wesley

Améliorer son niveau en C++

James Coplien Programmation avancée en C++ Addison

Scott Myers Effective C++ CD
Effective C++ & More Effective C++ Wesley
(2nd Ed.)



En ligne <http://www.cppreference.com> et <http://www.cplusplus.com>

2. La conception orientée objet

2.1 Limites des langages procéduraux



- ❑ **Aucuns « guides » pour architecturer données et traitements**
 - ❖ Structures de données + sous-programmes indépendants
 - ❖ Seul l'auteur du programme s'y retrouve
 - ❖ On recommence souvent les mêmes sous-programmes
 - ❖ Evolution et maintenance difficiles

- ❑ **Aucune corrélation entre spécifications et réalisation**
 - ❖ Structure du programme indépendante des spécifications
 - ❖ La modélisation informatique n'a rien à voir avec la structure réelle du problème

2.La conception orientée objet

2.2 Objectifs des langages objets

- ❑ **Permettre une modélisation informatique représentative du monde réel :**
 - ❖ **Créer des familles et les hiérarchiser**
 - ❖ **Associer données et traitements**
- ❑ **Faciliter le passage spécification/réalisation**
 - ❖ **L'analyse se traduit en structures logicielles**
- ❑ **Faciliter les évolutions et la maintenance**
- ❑ **Faciliter la réutilisation du code**

2. La conception orientée objet

2.3 Définition d'un objet

- ❑ **Un objet est une structure logicielle permettant d'associer données et traitements**
 - ❖ Les données sont les grandeurs représentatives de l'objet
 - ❖ Les traitements sont les actions réalisables sur l'objet
- ❑ **Un objet est une abstraction du monde réel**
 - ❖ Un objet concret : un ascenseur, une voiture, ...
 - ❖ Un concept : une liste, une vitesse, ...
 - ❖ Une activité : un gestionnaire de tâches, une entrée, ...
 - ❖ Etc ...
- ❑ **Exemple simple d'un objet « cercle »**
 - ❖ Données : coordonnées du centre, rayon, couleur, etc...
 - ❖ Traitements : calcul du périmètre, déplacement, affichage, etc...

REMARQUE : données et traitements sont regroupés au sein d'une structure logicielle ce qui est logique dans la mesure où ces traitements s'appliquent uniquement à ces données. Dans l'exemple, la fonction de calcul du périmètre d'un cercle a nécessairement besoin du « rayon ». **Elle ne peut être utilisée en l'absence des données de l'objet « cercle ».**

2. La conception orientée objet

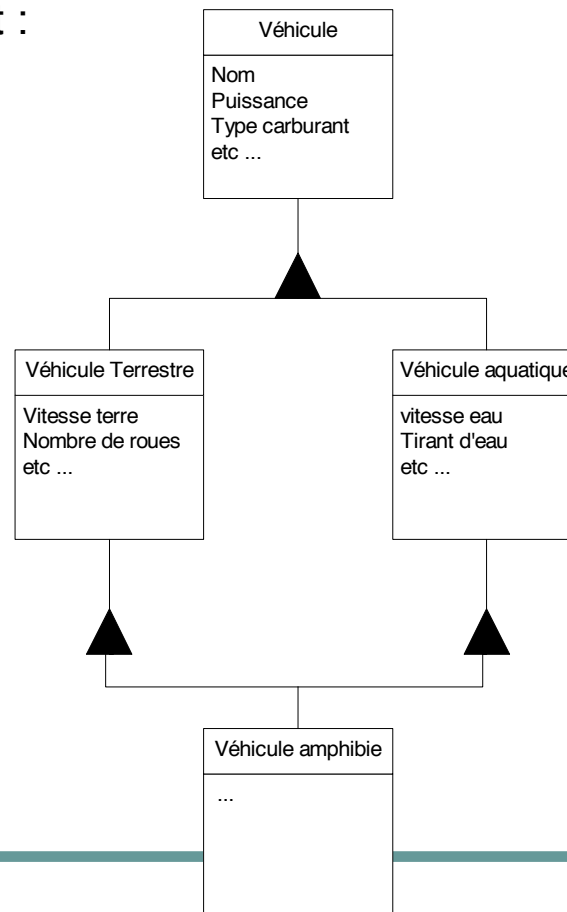
2.4 Concepts des langages objets

- ❑ **Typage abstrait**
 - ❖ La création de types autres que ceux du langage (définition d'un objet)
 - ❖ Une variable créée à partir de ce type est appelée une instance de l'objet
- ❑ **Encapsulation des données**
 - ❖ Réaliser des modules sans données globales, protéger les données privées
- ❑ **Héritage**
 - ❖ Pouvoir définir des « sous-types » de types plus généraux.
- ❑ **Polymorphisme**
 - ❖ Les noms de fonctions ne sont plus uniques pour un programme. Une fonction donnée peut être redéfinie pour chaque objet d'une hiérarchie de classes.
- ❑ **Surcharge**
 - ❖ Une fonction ne se différencie pas uniquement par son nom mais également par le type de ses paramètres d'appel.
- ❑ **Création dynamique d'objets**
 - ❖ Les instances d'objets peuvent être créées et détruites lors de l'exécution.

2.La conception orientée objet

2.5 Exemple de décomposition en objets

- ❑ Problème (simple): gestion d'un parc de véhicules composés de véhicules terrestres, aquatiques et amphibies. Une conception objet pourrait être symbolisée comme suit :



3. Un meilleur C

3.2 Les commentaires

Le langage C++ offre une nouvelle façon d'ajouter des commentaires. En plus des symboles `/*` et `*/` utilisés en C, le langage C++ offre les symboles `//` qui permettent d'ignorer tout jusqu'à la fin de la ligne. Exemple :

```
/* commentaire traditionnel
   sur plusieurs lignes
*/
void main()
{      // commentaire de fin de ligne
#if 0
           // une partie d'un programme en C ou C++
           // peut être ignorée par les directives
           // au préprocesseur #if .... #endif
#endif
}
```

3. Un meilleur C

3.4 Entrées/sorties avec *cin*, *cout* et *cerr*

La bibliothèque `iostream` et la surcharge des opérateurs offrent les facilités d'**entrées/sorties par flux** (ou flot / stream). Quatre flots sont prédéfinis dans le fichier d'en-tête ***iostream*** :

- ❖ `cout` qui correspond à la sortie standard
- ❖ `cin` qui correspond à l'entrée standard
- ❖ `cerr` qui correspond à la sortie standard d'erreur non tamponné
- ❖ `clog` qui correspond à la sortie standard d'erreur tamponné.

L'opérateur (surchargé) `<<` permet d'envoyer des valeurs dans un flot de sortie, tandis que `>>` permet d'extraire des valeurs d'un flot d'entrée.

3. Un meilleur C

3.4 Entrées/sorties avec cin , cout et cerr

Exemple :

```
#include <iostream>
using namespace std;
```

```
void main()
```

```
{
```

```
    int    i=123;    float f=1234.567; char  ch[80]="Bonjour\n", rep;
```

```
    cout << "i = " << i << "    f=" << f << "    ch=" << ch;
```

```
    cout << "i = ? "; cin >> i;    // lecture d'un entier
```

```
    cout << "f = ? "; cin >> f;    // lecture d'un réel
```

```
    cout << "rep = ? "; cin >> rep; // lecture d'un caractère
```

```
    cout << "ch = ? "; cin >> ch;  // lecture du premier mot d'une chaîne
```

```
    cout << "ch = " << ch;        // c'est bien le premier mot
```

```
}
```

```
/*-- résultat de l'exécution ----- */
i = 123  f=1234.57  ch=Bonjour
i = ? 12
f = ? 34.5
rep = ? y
ch = ? c++ is easy
Ch = c++
```

3. Un meilleur C

3.5 Définition de variables

En C++ vous pouvez déclarer les variables ou fonctions n'importe où dans le code. La portée de telles variables va de l'endroit de la déclaration jusqu'à la fin du bloc courant. Ceci permet :

- ❖ de définir une variable aussi près que possible de son utilisation afin d'améliorer la lisibilité (utile pour des grosses fonctions ayant beaucoup de variables locales)
- ❖ d'initialiser un objet avec une valeur obtenue par calcul ou saisie

```
#include <iostream>
using namespace std;
void main() {
    int i=0;                // définition d'une variable
    i++;                    // instruction
    int j=i;                // définition d'une autre variable
    j++;                    // instruction
    int somme(int n1, int n2); // déclaration d'une fonction
    cout << i << " " << j << " " << somme(i, j);
    cin >> i;
    const int k = i;        // définition d'une constante initialisée avec la valeur saisie
}
```

3. Un meilleur C

3.6 *Variable de boucle*

On peut déclarer une variable de boucle directement dans l'instruction for. Ceci permet de n'utiliser cette variable que dans le bloc de la boucle.

```
#include <iostream>
using namespace std;

void main()
{
    for (int i = 0; i < 10; i++)
        cout << i << ' ';

    // i n'est pas utilisé après la fin de la boucle

}
```

```
/*-- résultat de l'exécution -----*/
0 1 2 3 4 5 6 7 8 9
```


3. Un meilleur C

3.7 Les constantes

Les habitués du C ont l'habitude d'utiliser la directive du préprocesseur `#define` pour définir des constantes. Il est reconnu que l'utilisation du préprocesseur est une source d'erreurs difficiles à détecter. En C++, l'utilisation du préprocesseur se limite aux cas les plus sûrs :

- ❖ inclusion de fichiers (`#include`)
- ❖ compilation conditionnelle (`#if ... #endif`)

Le mot réservé ***const*** permet de définir une constante. L'objet ainsi spécifié ne pourra pas être modifié durant toute sa durée de vie. Il est indispensable d'initialiser la constante au moment de sa définition.

```
const int N = 10;           // N est un entier constant.
const int MOIS=12, AN=1995; // 2 constantes entières
int tab[2 * N];             // autorisé en C++, interdit en C
```

3. Un meilleur C

3.8 Les types composés

En C++, comme en langage C, le programmeur peut définir de nouveaux types en définissant des struct , enum ou union.

```
struct FICHE
{          // définition du type FICHE
    char *nom, *prenom;
    int  age;
};
FICHE adherent, *liste;


/*-----*/
enum BOOLEEN { FAUX, VRAI };
BOOLEEN trouve;
trouve = FAUX;
trouve = 0;          // ERREUR en C++ : verification stricte des types
trouve = (BOOLEEN) 0;    // OK
```

3. Un meilleur C

3.8 Les types composés

Chaque énumération enum est un type particulier, différent de int et ne peut prendre que les valeurs énumérées dans sa définition :

```
enum Jour {DIMANCHE, LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI};  
enum Couleur {NOIR, BLEU, VERT, CYAN, ROUGE, MAGENTA, BRUN, GRIS};
```

```
Jour j;           // définition d'une variable de type Jour  
j = LUNDI;        // OK  
int i = MARDI;     // légal, il existe une conversion implicite vers le type int
```

```
Couleur c;        // définition d'une variable de type Couleur  
c = j;            // ERREUR en C++
```

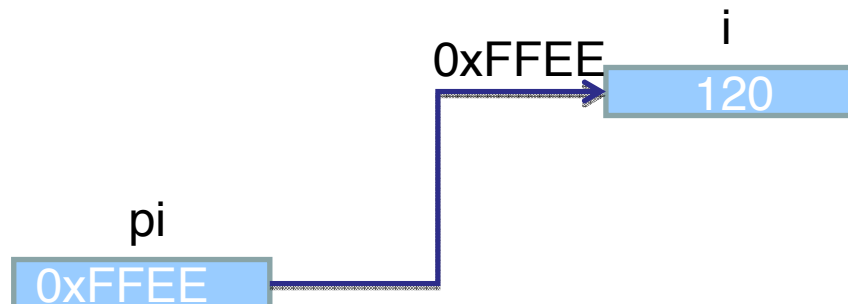
3. Un meilleur C

3.9 Pointeurs

C++ possède la notion de pointeur, héritée de C. Un pointeur est un variable (ou constante) qui contient l'adresse d'une variable de type de base ou d'une instance de classe :

```
int i = 120;  
int *pi = &i;
```

En C et C++, `*pi` désigne l'objet pointé ; dans notre exemple, `*pi` et `i` désignent donc la même variable entière.



3. Un meilleur C

3.10 Variables références

En plus des variables normales et des pointeurs, le C++ offre les variables références. Une variable référence permet de créer une variable qui est un "synonyme" d'une autre. Dès lors, une modification de l'une affectera le contenu de l'autre.

```
int i;  
int & ir = i; // ir est une référence à i  
int *ptr;  
  
i = 1;  
cout << "i= " << i << " ir= " << ir << endl; // affichage de : i= 1 ir= 1  
ir = 2;  
cout << "i= " << i << " ir= " << ir << endl; // affichage de : i= 2 ir= 2  
  
ptr = &ir;  
*ptr = 3;  
cout << "i= " << i << " ir= " << ir << endl; // affichage de : i= 3 ir= 3
```

Une variable référence doit obligatoirement être initialisée et le type de l'objet initial doit être le même que l'objet référence.

3. Un meilleur C

3.11 Allocation mémoire

C++ permet 3 modes d'allocation des objets :

- ❖ Statique pour un objet créé hors d'un bloc,
- ❖ Automatique pour un objet créé dans un bloc,
- ❖ Dynamique.

C++ dispose des opérateurs *new* et *delete* pour la création dynamique / destruction des objets. La durée de vie de l'objet créé est de :

- ❖ mode statique : destruction automatique à la fin de l'exécution du programme,
- ❖ mode automatique : destruction automatique à la fin du bloc (fin normale ou suite à une exception),
- ❖ mode dynamique : lors de l'appel à l'opérateur *delete* sur l'objet .

L'opérateur *new* réserve l'espace mémoire et l'initialise. Il retourne l'adresse de début de la zone mémoire allouée.

3. Un meilleur C

3.11 Allocation mémoire

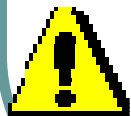
```
int *ptr1, *ptr2, *ptr3;

ptr1 = new int;           // allocation dynamique d'un entier
ptr2 = new int [10];      // allocation d'un tableau de 10 entiers
/*-----*/
struct date {int jour, mois, an; };
date *ptr4, *ptr5, *ptr6, d = {25, 4, 1952};

ptr4 = new date;          // allocation dynamique d'une structure

ptr5 = new date[10];      // allocation dynamique d'un tableau de structure
```

En cas d'erreur d'allocation par new, une exception `bad_alloc` est lancée s'il n'y a pas de fonction d'interception définie par l'utilisateur.



Attention à ne pas confondre la création d'un tableau de 10 entiers :

```
int *ptr = new int[10];
```

avec la création d'un entier initialisé à 10 :

```
int *ptr = new int(10);
```

3. Un meilleur C

3.11 Allocation mémoire

L'opérateur *delete* libère l'espace mémoire alloué par *new* à un seul objet, tandis que l'opérateur *delete[]* libère l'espace mémoire alloué à un tableau d'objets.

```
// libération d'un entier
delete ptr1;

// libération d'un tableau d'entier
delete[] ptr2;
```

L'application de l'opérateur *delete* à un pointeur nul est légale et n'entraîne aucune conséquence fâcheuse (l'opération est tout simplement ignorée).

A chaque instruction *new* doit correspondre une instruction *delete*.

- ❖ Il est important de libérer l'espace mémoire dès que celui ci n'est plus nécessaire.
- ❖ La mémoire allouée en cours de programme sera libérée automatiquement à la fin du programme.
- ❖ Tout ce qui est alloué avec *new[]*, doit être libéré avec *delete[]*

4 Les fonctions

4.1 Déclaration des fonctions

C++ dispose de trois « types » de fonctions :

- ❖ fonctions-membres d'instance,
- ❖ fonctions-membres statiques,
- ❖ fonctions ordinaires (non membres d'une classe, à portée globale ou de namespace).



La fonction *main()* par laquelle débute l'exécution d'un programme C++ doit être une de ces fonctions ordinaires et elle doit être unique dans tout le programme.

4 Les fonctions

4.2 Prototype de fonction

Le langage C++ impose au programmeur de déclarer le nombre et le type des arguments de la fonction.

La déclaration suivante `int f1();` où `f1` est déclarée avec une liste d'arguments vide est interprétée en C++ comme la déclaration `int f1(void);`

Une fonction dont le type de la valeur de retour n'est pas `void`, doit obligatoirement retourner une valeur. Sans type spécifié, le type retourné est `int`.

La déclaration de fonction possède deux mécanismes supplémentaires :

- ❖ la possibilité d'arguments ayant des valeurs par défaut ;
- ❖ la possibilité de déclarer une fonction en ligne (*inline*), suggestion au compilateur de remplacer l'appel de la fonction par une expansion en ligne de son corps, plutôt que de générer une séquence d'appel/retour.

4 Les fonctions

4.3 Passage par référence

En plus du passage par valeur et par pointeur, le C++ définit le passage par référence. Lorsque l'on passe à une fonction un paramètre par référence, cette fonction reçoit un "synonyme" du paramètre réel. Toute modification du paramètre référence est répercutée sur le paramètre réel.

```
void echange(int &n1, int &n2) { // n1 est un alias de i, n2 est un alias de j
    int temp = n1;
    n1 = n2; // toute modification de n1 est répercutée sur i
    n2 = temp; // toute modification de n2 est répercutée sur j
}

void main() {
    int i=2, j=3;
    echange(i, j);
    cout << "i= " << i << " j= " << j << endl;
}
```

```
/*-- résultat de l'exécution -----*/
i=3 j=2
```

Comme vous le remarquez, l'appel se fait de manière très simple.

- ❖ Utilisez les références quand vous pouvez,
- ❖ Utiliser les pointeurs quand vous devez.

4 Les fonctions

4.3 Passage par référence

Cette facilité doit être utilisée avec précaution, car elle ne protège plus la valeur du paramètre réel transmis par référence. L'utilisation du mot réservé *const* permet d'annuler ces risques pour les arguments ne devant pas être modifiés dans la fonction. Les règles :

L'argument ne doit pas être modifié

L'argument est un type intégré (*int*, *float*, ...)

- passage par valeur

L'argument est un objet

- passage par référence constante

L'argument doit être modifié

- passage par référence ou par pointeur

4 Les fonctions

4.4 Retour de fonction

```
Voiture *usine( int type, int equipements){  
    Voiture *v;  
    v = new Voiture(type);  
    v->ajouter(equipements);  
    return v;  
}
```

En C++, il est possible de retourner au choix une valeur, un pointeur ou une référence, quelle que soit la nature de l'objet.

Dans le cas du retour par pointeur ou par référence, le programmeur C++ doit s'assurer que l'objet pointé ou référencé a une durée de vie plus longue que l'exécution de la fonction elle même (c'est-à-dire que cet objet ne doit pas être un objet local automatique).

4 Les fonctions

4.5 Surcharge de fonctions

Une fonction se définit par :

- ❖ son nom,
- ❖ sa liste typée de paramètres formels,
- ❖ le type de la valeur qu'elle retourne.

Mais seuls les deux premiers critères sont discriminants. On dit qu'ils constituent la signature de la fonction. On peut utiliser cette propriété pour donner un même nom à des fonctions qui ont des paramètres différents.

```
int somme( int n1, int n2)          { return n1 + n2; }
int somme( int n1, int n2, int n3) { return n1 + n2 + n3; }
double somme( double n1, double n2) { return n1 + n2; }

void main()
{
    cout << "1 + 2 = " << somme(1, 2) << endl;
    cout << "1 + 2 + 3 = " << somme(1, 2, 3) << endl;
    cout << "1.2 + 2.3 = " << somme(1.2, 2.3) << endl;
}
```

4 Les fonctions

4.5 Surcharge de fonctions

Le compilateur sélectionnera la fonction à appeler en fonction du type et du nombre des arguments qui figurent dans l'appel de la fonction. Ce choix se faisant à la compilation, l'appel d'une fonction surchargée procure des performances identiques à un appel de fonction "classique". On dit que l'appel de la fonction est résolu de manière statique.

```
enum Jour {DIMANCHE, LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI};  
enum Couleur {NOIR, BLEU, VERT, CYAN, ROUGE, MAGENTA, BRUN, GRIS};  
  
void f1(Jour j);  
void f1(Couleur c); // OK : les types énumérations sont tous différents
```

```
int sommel(int n1, int n2)           {return n1 + n2;}  
int sommel(const int n1, const int n2) {return n1 + n2;}  
// Erreur : la liste de paramètres dans les déclarations des deux fonctions  
// n'est pas assez divergente pour les différencier.
```

4 Les fonctions

4.5 Surcharge de fonctions

```
int somme2(int n1, int n2) {return n1 + n2;}
int somme2(int & n1, int & n2) {return n1 + n2;}
// Erreur : la liste de paramètres dans les déclarations des deux fonctions
// n'est pas assez divergente pour les différencier.
```

```
int    somme3(int n1, int n2) {return n1 + n2;}
double somme3(int n1, int n2) {return (double) n1 + n2;}

// Erreur:      seul le type des paramètres permet de faire la distinction entre
// les fonctions et non pas la valeur retournée.
```

C++ permet la **surcharge des opérateurs sous forme de fonctions membres ou de fonctions ordinaires : le nom de la fonction doit commencer par le mot-clé *operator* suivi de la notation correspondant à l'opérateur (*operator+*, *operator[]*, *operator new*, etc.).**

5 Structure des programmes

5.1 Unité de compilation

Le source d'un programme est constitué de plusieurs fichiers, appelés unités de compilation. Chaque unité est compilée séparément, afin de produire des fichiers binaires (fichiers objets d'extension .o ou .obj en C++).



Une fois les fichiers binaires obtenus, il faut les regrouper pour obtenir un exécutable. Ce mécanisme est effectué statiquement en C++ par un « éditeur de liens ».

Afin de permettre cette compilation séparée, chaque unité doit comporter l'information nécessaire au compilateur pour effectuer sa traduction. En particulier, elle doit référencer explicitement les autres unités de compilation qu'elle utilise.

5 Structure des programmes

5.1.1 Unité de compilation et namespace en C++

En C++, l'unité de compilation est un fichier source, portant l'extension `.cpp`, ainsi que tous les fichiers d'en-tête (extension `.h`) qu'il inclut (`#include`), et ceci récursivement. Une telle unité de compilation peut contenir des définitions de types, de classes, de fonctions ordinaires, de fonctions membres, d'objets globaux...

Il n'y a pas de notion de package à proprement parler. La directive `#include` va chercher les fichiers d'inclusion dans le répertoire courant et/ou dans un ensemble de répertoires par défaut.

Néanmoins, afin d'éviter d'avoir un seul espace de nommage global, C++ définit la notion d'espace de noms (*namespace*).

5 Structure des programmes

5.1.1 Unité de compilation et namespace en C++

C++ a un unique espace de nom dans lequel tous les noms déclarés en global figurent. Lors de l'utilisation de bibliothèques, des collisions pouvaient arriver rendant l'écriture d'une application utilisant deux bibliothèques incompatibles impossible ! ex :

```
bibliothèque A :          bibliothèque B :  
class animal;  
class animal;
```

Le mécanisme des namespaces (à rapprocher du package de ADA) permet de remédier à ce problème :

```
bibliothèque A :          bibliothèque B :  
namespace lib_A {        namespace lib_B {  
    class animal;  
    class lion;  
}  
}
```

Utilisation : à l'aide de l'opérateur de portée

```
lib_A :: animal aa;      ou   using namespace lib_A :: animal;  
lib_B :: animal ab;      animal aa;          // == lib_A :: animal aa;  
                           lib_A :: lion la;
```

OU

```
using namespace lib_A;  
animal    aa;          // == lib_A :: animal aa;  
lion      la;          // == lib_A :: lion aa;
```

5 Structure des programmes

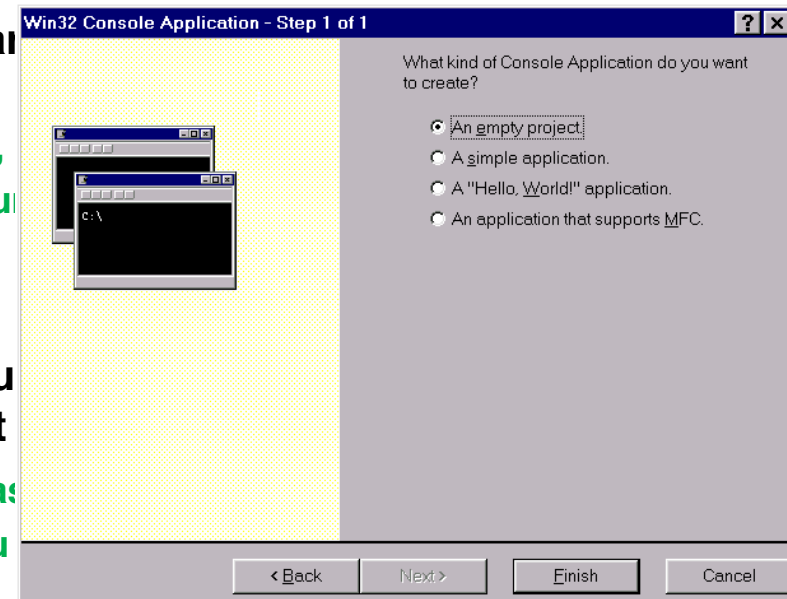
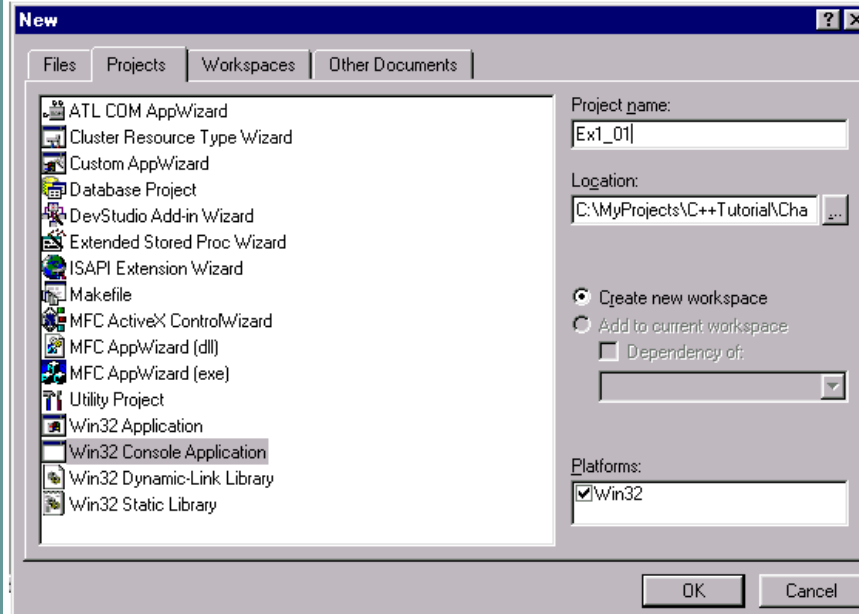
5.2 Préprocesseur

Le préprocesseur est un outil spécifique à C et C++. Il traite les directives `#include` et les directives de compilation conditionnelles (`#if`, `#ifdef`, `#ifndef`).

La plupart des utilisations de `#define`, qui sont inévitables en C, peuvent être avantageusement remplacées par des constructions C++ plus sûres : définition de constante, fonction en ligne, fonction générique.

5 Structure des programmes

5.3 Compilation sous Windows (Visual C++)



La compilation s'effectue par le menu build. La touche F5 permet de lancer la compilation suivie de l'exécution.

5 Structure des programmes

5.4 Compilation sous Linux à l'aide de g++

Compiler tous les fichiers code (.cpp) pour obtenir des fichiers objets (.o)

```
g++ -c fic.cpp      (génère fic.o)
```

Linker tous les fichiers objets pour obtenir un exécutable.

```
g++ -o NomExec fic1.o fic2.o .... ficN.o      (génère NomExec)
```

Il est possible de réunir ces deux étapes en une seule :

```
g++ -o NomExec fic1.cpp fic2.cpp .... FicN.cpp
```

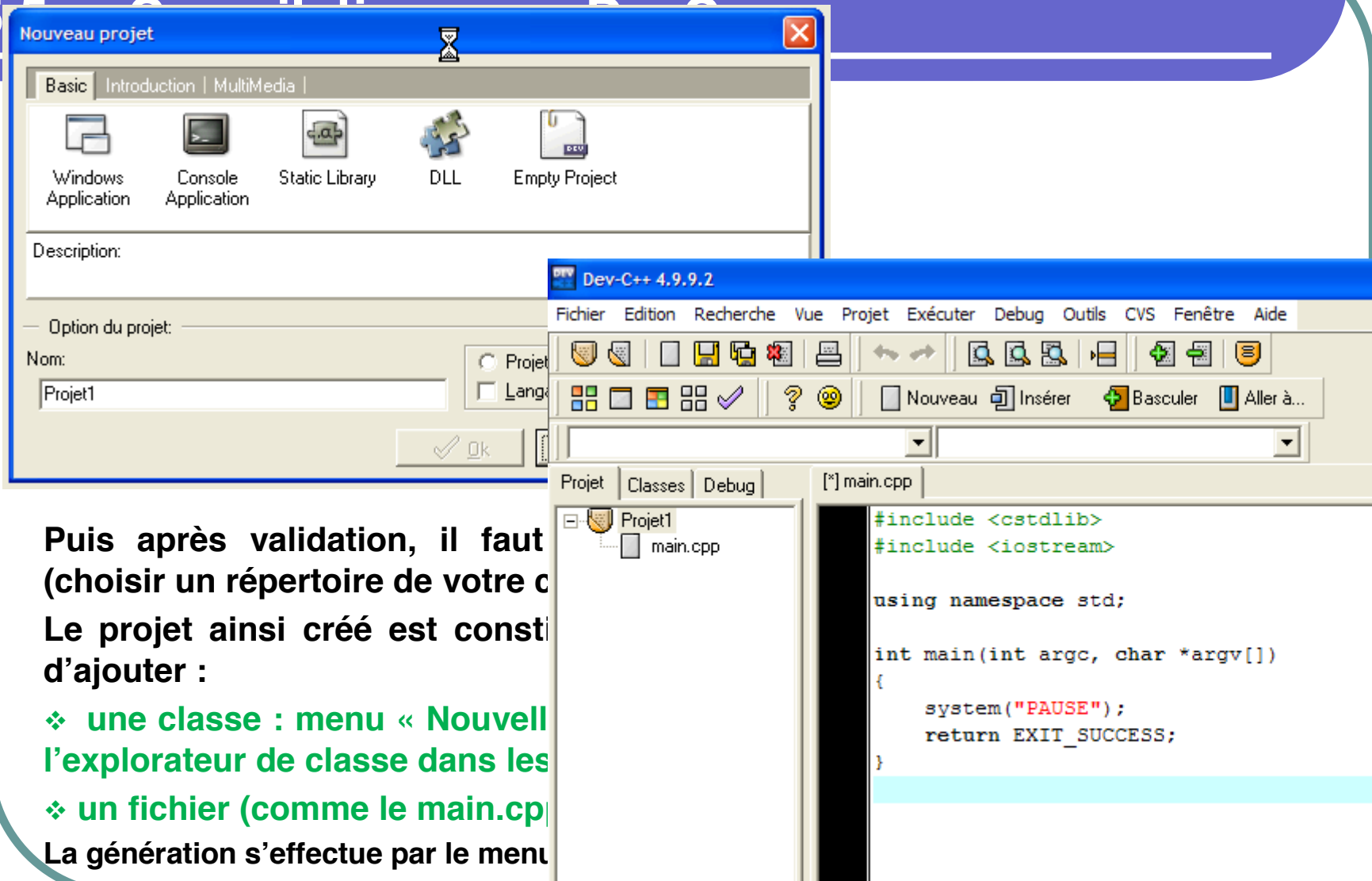
Compilation de l'exemple précédent :

```
g++ -o MainExemple Exemple.cpp Main.cpp
```

Le nombre de fichiers devenant généralement assez conséquent, l'utilisation du makefile devient obligatoire. Pour le déverminage de vos programmes, un débogueur graphique comme ddd (Data Display Debugger) ou Kdevelop (Linux) est recommandé. Il est alors nécessaire d'avoir compiler ses sources avec l'option « debug » :

```
g++ -c -g exemple.cpp
```

5 Structure des programmes



The screenshot shows the Dev-C++ 4.9.9.2 interface. The 'Nouveau projet' (New Project) dialog is open, displaying the 'Basic' tab with options: Windows Application, Console Application, Static Library, DLL, and Empty Project. The 'Description' field is empty. Under 'Option du projet:', the 'Nom:' (Name) field contains 'Projet1'. The 'Projet' radio button is selected. The 'Ok' button is highlighted. In the background, the main Dev-C++ window is visible, showing the menu bar, toolbar, and a project explorer on the left with 'Projet1' and 'main.cpp'. The main editor window displays the following C++ code:

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Puis après validation, il faut
(choisir un répertoire de votre c
Le projet ainsi créé est consti
d'ajouter :

- ❖ une classe : menu « Nouvell
 - l'explorateur de classe dans les
 - ❖ un fichier (comme le main.cp
- La génération s'effectue par le menu

6 La notion d'objet C++ (Classe)

6.1 Définition d'une classe

La classe décrit le modèle structurel d'un objet :

- ❖ attributs décrivant sa structure (appelés aussi champs ou données membres)
- ❖ opérations qui lui sont applicables (méthodes ou fonctions membres)

Les mots réservés **public** et **private** délimitent les sections visibles par l'application. Exemple :

```
class Avion {  
    private : // membres privées  
    // données membres privées  
        char _immatriculation[6], *_type;  
        float _poids;  
    // fonction membre privée  
        void erreur(char *message);  
    public : // fonctions membres publiques  
        void init(char im[], char *type, float p);  
        void affiche();  
}; // n'oubliez pas ce ; après l'accolade
```


6 La notion d'objet C++ (Classe)

6.2 Droits d'accès

L'encapsulation consiste à masquer l'accès à certains attributs et fonctions-membres d'une classe.

Elle est réalisée à l'aide des mots clés :

- ❖ **private** : les membres privés ne sont accessibles que par les fonctions membres de la classe.
- ❖ **protected** : les membres protégés sont comme les membres privés. Mais ils sont aussi accessibles par les fonctions membres des classes dérivées (voir l'héritage).
- ❖ **public** : les membres publics sont accessibles par tous. La partie publique est appelée interface.

Les mots réservés **private**, **protected** et **public** peuvent figurer plusieurs fois dans la déclaration de la classe.

Le droit d'accès ne change pas tant qu'un nouveau droit n'est pas spécifié.

6 La notion d'objet C++ (Classe)

6.3 Définition des fonctions membres

En général, la déclaration d'une classe contient simplement les prototypes des fonctions membres de la classe.

```
class Avion {  
    private :  
        char  _immatriculation[6], *_type;  
        float _poids;  
        void  erreur(char *message);  
    public :  
        void init(char [], char *, float);  
        void affiche();  
};
```

Les fonctions membres sont définies dans un module séparé ou plus loin dans le code source. Syntaxe de la définition hors de la classe d'une fonction-membre :

```
type_valeur_retournée Classe::nom_fonction( paramètres_formels )  
{  
    // corps de la fonction  
}
```

6 La notion d'objet C++ (Classe)

6.3 Définition des fonctions membres

**Dans la fonction membre on a un accès direct à tous les membres de la classe.
Exemple de définition de fonctions membres de la classe Avion :**

```
void Avion::init(char m[], char *t, float p) {
    if ( strlen(m) != 5 )
    {
        erreur("Immatriculation invalide");
        strcpy(_immatriculation, "?????");
    }
    else
        strcpy(_immatriculation, m);
    _type = new char [strlen(t)+1];
    strcpy(_type, t);
    _poids = p;
}
void Avion::affiche()
{
    cout << _immatriculation << " " << _type;
    cout << " " << _poids << endl;
}
```

6 La notion d'objet C++ (Classe)

6.4 Instanciation d'une classe

De façon similaire à une *struct* ou à une *union*, le nom de la classe représente un nouveau type de donnée.

On peut donc définir des variables de ce nouveau type. On dit alors que l'on crée des objets ou des instances de cette classe.

```
Avion av1;                // une instance simple (statique)
Avion *av2;               // un pointeur (non initialisé)
Avion compagnie[10];      // un tableau d'instances

av2 = new Avion;          // création (dynamique) d'une instance
```

6 La notion d'objet C++ (Classe)

6.5 Utilisation des objets

Après avoir créé une instance (de façon statique ou dynamique) on peut accéder aux attributs et fonctions membres de la classe.

Cet accès se fait comme pour les structures à l'aide de l'opérateur . (point) ou -> (tiret supérieur).

```
Avion av1;                // une instance simple (statique)
Avion *av2;                // un pointeur (non initialisé)
Avion compagnie[10];       // un tableau d'instances

av2 = new Avion;           // création (dynamique) d'une instance

av1.init("FGBCD", "TB20", 1.47);
av2->init("FGDEF", "ATR 42", 80.0);
compagnie[0].init("FEFGH", "A320", 150.0);
av1.affiche();
av2->affiche();
compagnie[0].affiche();
av1._poids = 0; // erreur, poids est un membre privé
```

6 La notion d'objet C++ (Classe)

6.6 Constructeurs

- Les données membres d'une classe ne peuvent pas être initialisées, il faut donc prévoir une fonction d'initialisation. Sans l'appel à cette fonction d'initialisation, le reste n'a plus de sens et il se produira très certainement des erreurs dans la suite de l'exécution.
- De même, après avoir fini d'utiliser l'objet, il est bon de prévoir une fonction membre permettant de détruire l'objet (libération des ressources allouées, ...).
- Les attributs doivent être initialisés dans un constructeur (sauf pour les membres statiques). Le constructeur est une fonction membre spécifique de la classe qui est appelée implicitement à la création de l'objet, assurant ainsi une initialisation correcte. Ce constructeur est une fonction qui porte comme nom, le nom de la classe et qui ne retourne pas de valeur (pas même un void).

6 La notion d'objet C++ (Classe)

6.6 Constructeurs

On appelle constructeur par défaut un constructeur n'ayant pas de paramètre ou ayant des valeurs par défaut pour tous les paramètres.

```
class Nombre {  
    private :  
        int _i;  
    public :  
        Nombre() { _i = 0; } // constructeur  
};
```

Si le concepteur de la classe ne spécifie pas de constructeur, le compilateur générera un constructeur par défaut.

```
class Nombre {  
    private :  
        int _i;  
    public :  
        Nombre(int i) { _i = i; }  
        // ...  
};
```

6 La notion d'objet C++ (Classe)

6.6 Constructeurs

Comme les autres fonctions, les constructeurs peuvent être surchargés.

```
class Nombre {  
private :  
    int _i;  
public :  
    Nombre();          // constructeur par défaut  
    Nombre(int i);     // constructeur à 1 paramètre  
};
```

Le constructeur est appelé à l'instanciation de l'objet. Il n'est donc pas appelé quand on définit un pointeur sur un objet ...

```
Nombre n1;                // correct, appel du constructeur par défaut  
Nombre n2(10);            // correct, appel du constructeur à 1 paramètre  
Nombre n3 = Nombre(10);   // idem que n2  
Nombre *ptr1, *ptr2;      // correct, pas d'appel aux constructeurs  
ptr1 = new Nombre;        // appel au constructeur par défaut  
ptr2 = new Nombre(12);    // appel du constructeur à 1 paramètre  
Nombre tab1[10];          // chaque objet est initialisé par le constructeur par défaut  
Nombre tab2[3] = {        Nombre(10), Nombre(20), Nombre(30) };  
    // initialisation des 3 objets tableau par les nombres 10, 20 et 30
```


6 La notion d'objet C++ (Classe)

6.7 Destructeurs

De la même façon que pour les constructeurs, le destructeur est une fonction membre spécifique de la classe qui est appelée implicitement à la destruction de l'objet. Ce destructeur est une fonction :

- ❖ Qui porte comme nom, le nom de la classe précédé du caractère ~ (tilda)
- ❖ Qui ne retourne pas de valeur (pas même un void)
- ❖ Qui n'accepte aucun paramètre (le destructeur ne peut donc pas être surchargé)

```
class Exemple {  
    private :  
        // ...  
    public :  
        // ...  
        ~Exemple();  
};  
Exemple::~~Exemple() {  
    // ...  
}
```

6 La notion d'objet C++ (Classe)

6.7 Destructeurs

Comme pour le constructeur, le compilateur générera un destructeur par défaut si le concepteur de la classe n'en spécifie pas un.



- ❖ Si des constructeurs sont définis avec des paramètres, le compilateur ne générera pas le constructeur par défaut (le destructeur est unique).
- ❖ Les constructeurs et destructeurs sont les seules fonctions membres non constantes qui peuvent être appelées pour des objets constants.
- ❖ Le nom des constructeurs et du destructeur sont imposés par le langage et doivent être strictement identiques au nom de la classe (ATTENTION à la casse)


Attention aux fautes de frappe, particulièrement dans l'identificateur du constructeur

```
class Essai {  
    private:  
        char *_ptr;  
    public:  
        essai() { _ptr = new char[80]; } // constructeur ?  
        void init(char c) { for(int i=0; i<80; i++) _ptr[i]=c; }  
        ~Essai() { delete[] _ptr; }  
};
```

6 La notion d'objet C++ (Classe)

6.8 Exemple : pile d'entiers

```
#include <iostream>
#include <assert.h>
#include <stdlib.h> // rand()
using namespace std;

class IntStack {
private:
 int _sommet;
    int _taille;
    int *_addr; // adresse de la pile
public:
    IntStack(int taille); // constructeur par défaut
    ~IntStack() { delete[] _addr; } // destructeur

    void push(int n); // empile un entier au sommet de la pile
    int pop(); // retourne l'entier au sommet de la pile
    int vide(); // vrai, si la pile est vide
    int pleine(); // vrai, si la pile est pleine
    int getsize(){ return _taille; }
};
```

6 La notion d'objet C++ (Classe)

6.8 Exemple : pile d'entiers

```
IntStack::IntStack(int taille) {
    _taille = taille ;
    _addr = new int [ taille ] ;
    assert( _addr != 0 );
    _somet = 0;
}
void IntStack::push(int n) {
    if ( ! pleine() ) {
        _addr[ _somet ] = n;
        _somet++ ;
    }
}
int IntStack::pop() {
    if ( ! vide() ) {
        _somet-- ;
        return _addr[ _somet ]
    } else
        return 0;
}
```

```
int IntStack::vide() {
    if (_somet == 0) return 1; else return 0;
}
int IntStack::pleine() {
    if (_somet == _taille) return 1 else
    return 0;
}
void main() {
    IntStack pile1(15);      // pile de 15
entiers
    while ( ! pile1.pleine() ) // remplissage
        pile1.push( rand() % 100 );

    while ( ! pile1.vide() ) // Affichage pile
        cout << pile1.pop() << "  ";
    cout << endl;
}
```

6 La notion d'objet C++ (Classe)

6.9 Constructeur par copie

Présentation du problème :

Reprenons la classe `IntStack` avec un constructeur et un destructeur et écrivons une fonction (`AfficheSommet`) qui affiche la valeur de l'entier au sommet de la pile qui est passée en paramètre.

```
void main() {  
    IntStack pile1(15);      // création d'une pile de 15 entiers  
    // ...  
    AfficheSommet( pile1 );  
    // ...  
}
```

Une version fausse (pour l'instant) de cette fonction pourrait être :

```
void AfficheSommet( IntStack pile ) {  
    cout << "Sommet de la pile : " << pile.pop() << endl;  
}
```

Ici, la pile est passée par valeur, donc il y a création de l'objet temporaire nommé `pile` créé en copiant les valeurs du paramètre réel `pile1`.

6 La notion d'objet C++ (Classe)

6.9 Constructeur par copie

L'affichage de la valeur au sommet de la pile marche bien, mais la fin de cette fonction fait appel au destructeur de l'objet local `pile` qui libère la mémoire allouée par l'objet `pile1` parce que les données membres de l'objet `pile` contiennent les mêmes valeurs que celles de l'objet `pile1`.

<code>pile1</code>	<code>pile</code>
<code>_sommet 15</code>	<code>_sommet 15</code>
<code>_taille 15</code>	<code>_taille 15</code>
<code>_addr 0x400039c0</code>	<code>_addr 0x400039c0</code>

Pour éviter cela, il faut définir la fonction `AfficheSommet` comme :

```
void AfficheSommet( IntStack & pile ) {  
    cout << "Sommet de la pile : " << pile.pop() << endl;  
}
```

Mais une opération comme `pile.pop()` dépile un entier de la pile `pile1` !!!

Il faut faire une copie intelligente : création du constructeur de copie

6 La notion d'objet C++ (Classe)

6.9 Constructeur par recopie

Le constructeur de copie est invoqué à la construction d'un objet à partir d'un objet existant de la même classe.

```
Nombre n1(10);      // appel du constructeur à 1 paramètre
Nombre n2(n1);      // appel du constructeur de copie
Nombre n3=n1;       // appel du constructeur de copie
```

Le constructeur de copie est appelé aussi pour le passage d'arguments par valeur et le retour de valeur

```
Nombre traitement(Nombre n) {
    static Nombre nbre;
    // ...
    return nbre;      // appel du constructeur de copie
}
void main() {
    Nombre n1, n2;
    n2 = traitement( n1 );      // appel du constructeur de copie
}
```

Le compilateur C++ génère par défaut un constructeur de copie "bête"

6 La notion d'objet C++ (Classe)

Constructeur de copie de la classe IntStack

```
class IntStack {
private:
    int _taille; // taille de la pile
    int _sommet; // position de l'entier à empiler
    int *_addr;  // adresse de la pile
public:
    IntStack(int taille);           // constructeur
    IntStack(const IntStack & s);   // constructeur de copie
    // ...
};

// ...

IntStack::IntStack(const IntStack & s) { // constructeur de copie
    _taille = s._taille ;
    _addr = new int [_taille];
    _sommet = s._sommet;
    // recopie des éléments
    for (int i=0; i< _sommet; i++)
        _addr[i] = s._addr[i];
}
```


6 La notion d'objet C++ (Classe)

6.10 Classes imbriquées

Il est possible de créer une classe par une relation d'appartenance « relation a un » ou « est composée de ». Exemple : une voiture a un moteur, a des roues ...

```
class Moteur      { /* ... */ };
class Roue        { /* ... */ };
class Voiture
{
private:
    Moteur _moteur;
    Roue   _roue[4];
    // ....

public:
    // ....
};
```

6 La notion d'objet C++ (Classe)

6.11 Affectation et initialisation

Le langage C++ fait la différence entre l'initialisation et l'affectation.

❖ l'affectation consiste à modifier la valeur d'une variable (et peut avoir lieu plusieurs fois)

❖ l'initialisation est une opération qui n'a lieu qu'une fois immédiatement après que l'espace mémoire ait été alloué. Cette opération consiste à donner une valeur initiale à l'objet ainsi créé.

6 La notion d'objet C++ (Classe)

6.12 Liste d'initialisation d'un constructeur

```
class Y { /* ... */ };
class X {
private:
    const int _x;
    Y _y;
    int _z;
public:
    X(int a, int b, Y y);
    ~X();
    // ....
};

X::X(int a, int b, Y y) {
    _x = a;    // ERREUR: l'affectation à une constante est interdite
    _z = b;    // OK : affectation et comment initialiser l'objet membre _y ???
}
```

Comment initialiser la donnée membre constante `_x` et appeler le constructeur de la classe `Y` ?

Réponse : la liste d'initialisation.

6 La notion d'objet C++ (Classe)

6.12 Liste d'initialisation d'un constructeur

La phase d'initialisation de l'objet utilise une liste d'initialisation qui est spécifiée dans la définition du constructeur.

Syntaxe :

```
nom_classe::nom_constructeur( args ... ) : liste_d_initialisation
{
    // corps du constructeur
}
```

Exemple :

```
IntStack:: IntStack(int a, int b, Y y) : _x( a ) , _y( y ) , _z( b ) {
    // rien d'autre à faire
}
```

L'expression `_x(a)` indique au compilateur d'initialiser la donnée membre `_x` avec la valeur du paramètre `a`.

L'expression `_y(y)` indique au compilateur d'initialiser la donnée membre `_y` par un appel au constructeur (avec l'argument `y`) de la classe `Y`.

6 La notion d'objet C++ (Classe)

6.13 Le pointeur *this*

Toute fonction membre d'une classe X a un paramètre caché : le pointeur *this*.

« *this* » contient l'adresse de l'objet qui l'a appelé, permettant ainsi à la fonction membre d'accéder aux membres de l'objet.

Il est implicitement déclaré comme (pour une variable) :

```
X * const this;
```

et comme (pour un objet constant) :

```
const X * const this;
```

et initialisé avec l'adresse de l'objet sur lequel la fonction membre est appelée.

6 La notion d'objet C++ (Classe)

6.13 Le pointeur this

```
classe X {  
    private:  
        int i;  
    public:  
        int f1() { return this->i; }    // idem que : int f1() { return i; }  
    // ...  
};
```



❖ La valeur de *this* ne peut pas être changée.

❖ *this* ne peut pas être explicitement déclaré.

```
class Voiture;  
class Roue {  
    Voiture *_v;  
public :  
    Roue(Voiture *v) : _v(v) {}  
};
```

La voiture connaît ses roues et
les roues connaissent la voiture.

```
class Voiture {  
    private:  
        Moteur _moteur;  
        Roue* _roue[4];  
    public:  
        Voiture() {  
            for (int i = 0 ; i < 4 ; i++)  
                _roue[i] = new Roue(this)  
        }  
};
```

7 Organisation d'un développement C++

Il est conseillé d'avoir deux fichiers par classe :

- **la définition de la classe (liste des déclarations de ses membres) ainsi que les prototypes de fonctions ordinaires dans un fichier d'en-tête .h,**
- **la définition du corps des fonctions (membres ou ordinaires) dans un fichier .cpp.**



Le nom des fichiers correspondra au nom de la classe.

Cela permet aux clients de la classe de n'inclure, et donc de ne dépendre, que du fichier d'en-tête .h. Le fichier .cpp, quant à lui, peut être compilé une fois pour toutes et placé dans une bibliothèque. Cette séparation de la spécification d'une part, de son implémentation d'autre part, est considérée depuis longtemps comme une bonne pratique de génie logiciel.

7 Organisation d'un développement C++

7.1 Recommandations de style

Il est souvent utile d'adopter des règles d'écriture communes pour faciliter la relecture du code.

Les règles d'écriture ci dessous n'ont rien d'obligatoire et sont données à titre d'exemple :

- ❖ La première lettre du nom de la classe en majuscule.
- ❖ La liste des données membres en premier.
- ❖ Séparer données et fonctions membres.
- ❖ Les noms des fonctions membres avec une majuscule à chaque mot significatif (ex : `afficherCoordonnéeX()`).
- ❖ Le caractère `_` comme premier caractère du nom d'une donnée membre (ex : `_coordonnéeX`)

7 Organisation d'un développement C++

7.2 Le fichier d'en-tête (.h)

Il contient :

- ❖ Les éventuels *include* nécessaires à la déclaration de la classe.
- ❖ La déclaration de la classe.

Pour se protéger contre l'inclusion multiple (source d'erreur à la compilation), utiliser la directive ***#ifdef ... #define ... #endif***.

7 Organisation d'un développement C++

7.3 Le fichier code (.cpp)

Il contient :

- ❖ L'*include* du fichier d'en-tête correspondant.
- ❖ Les éventuels *includes* système nécessaires .
- ❖ La définition des toutes les fonctions membres de la classe.

7

Organisation d'un développement C++

7.4

Exemple

Fichiers définissant une classe Exemple :

- **Exemple.h**
- **Exemple.cpp**

Exemple.h

```
#ifndef __EXEMPLE_H__
#define __EXEMPLE_H__

//----- Includes Systèmes
#include <iostream>

//----- Includes applicatifs
#include "MaClasse.h"
```

7 Organisation d'un développement C++

7.4 Exemple

Exemple.h

```
//----- Déclaration de la classe Exemple
class Exemple {
    // Données membres
private :
    int _donnee1 ;
    int _donnee2 ;
    // Fonctions membres ;
public :
    void fonc1() ;
    int  fonc2() ;
    // Constructeur(s)
    Exemple() ;
    Exemple(int a, int b) ;
    // Destructeur
    ~Exemple() ;
} ;

#endif      // __EXEMPLE_H__
```

7

Organisation d'un développement C++

7.4

Exemple

Exemple.cpp

```
#include <stdlib.h>                //----- Includes Systèmes
#include "Exemple.h"              //----- Includes applicatifs

Exemple::Exemple() {                // Code du constructeur 1
}

Exemple::Exemple(int a, int b) {    // Code du constructeur 2;
}

Exemple::~~Exemple() {             // Code du destructeur;
}

void Exemple :: fonc1() {           // Code de la fonction fonc1;
}
```

7

Organisation d'un développement C++

7.5

Utilisation d'un objet

Comme en C, la fonction `main()` est le point d'entrée du programme. Il est conseillé de coder cette fonction dans un fichier indépendant. Exemple de `main()` utilisant la classe `Exemple` précédemment définie :

Main.cpp

```
//----- Includes Systèmes
#include <iostream>

//----- Includes applicatifs
#include "Exemple.h"

int main() {
    int variable ;
    Exemple mon_obj_ex ;
    // Code...
}
```

- ❖ Ne pas hésiter à décomposer en plusieurs fichiers.
- ❖ Eviter les fichiers interminables (quelques pages écran).
- ❖ Utiliser des noms de fichier représentatifs de leur contenu !
- ❖ Insérer des cartouches en tête des fichiers contenant les informations importantes (contenu, but, auteur, date, etc...).
- ❖ Largement commenter le code.
- ❖ Si le nombre de fichiers devient trop important (gros projet) utiliser des répertoires pour les classer par affinités (ex : couches basses, API, IHM, etc...).

8 Héritage

8.1 L'héritage simple

L'héritage, également appelé dérivation, permet de créer une nouvelle classe à partir d'une classe déjà existante, la classe de base (ou super classe).

« Il est plus facile de modifier que de réinventer »

La nouvelle classe (ou classe dérivée ou sous classe) hérite de tous les membres, qui ne sont pas privés, de la classe de base et peut ainsi réutiliser le code déjà écrit pour la classe de base. On peut aussi lui ajouter de nouveaux membres ou redéfinir des méthodes.

classe de base

hérite de

classe dérivée

```
class A {  
    // membres  
};
```

mode de dérivation

```
class B : public A {  
    // membres ajoutés ou redéfinis  
};
```

La classe B hérite de façon publique de la classe A. Tous les membres publics ou protégés de la classe A font parti de l'interface de la classe B.

8 Héritage

8.2 Mode de dérivation

Seul C++ permet lors de la définition de la classe dérivée de spécifier un mode de dérivation par l'emploi d'un des mots-clés suivants :

- ❖ **public**
- ❖ **protected**
- ❖ **private**

Ce mode de dérivation détermine quels membres de la classe de base sont accessibles dans la classe dérivée. Au cas où aucun mode de dérivation n'est spécifié, le compilateur C++ prend par défaut le mot-clé *private* pour une classe et *public* pour une structure.

Les membres privés de la classe de base ne sont jamais accessibles par les membres des classes dérivées.

C++ supporte l'héritage multiple de classes, et introduit le mécanisme d'héritage virtuel pour éviter la duplication liée à l'héritage (indirect) plusieurs fois de la même classe.

8 Héritage

8.3 Héritage public

Il donne aux membres publics et protégés de la classe de base le même statut dans la classe dérivée. C'est la forme la plus courante d'héritage, car il permet de modéliser les relations «Y est une sorte de X» ou «Y est une spécialisation de la classe de base X».

```
class Vehicule {  
    public:  
        void pub1();  
    protected:  
        void prot1();  
    private:  
        void priv1();  
};
```

```
class Voiture : public Vehicule {  
    public:  
        int pub2() {  
            pub1();        // OK  
            prot1();       // OK  
            priv1();       // ERREUR  
        }  
};
```

```
Voiture safrane;  
safrane.pub1(); // OK  
safrane.pub2(); // OK
```

8 Héritage

8.4 Redéfinition de méthodes dans la classe dérivée

On peut redéfinir une fonction dans une classe dérivée si on lui donne le même nom que dans la classe de base. Il y aura ainsi, comme dans l'exemple ci après, deux fonctions f2(), mais il sera possible de les différencier avec l'opérateur :: de résolution de portée (utilisé aussi dans le cas des espaces de noms, *namespace*).

```
class X {  
    protected:  
        int _xxx;  
    public:  
        void f1();  
        void f2();  
};
```

```
class Y : public X {  
    public:  
        void f2();  
        void f3();  
};
```

```
void Y::f3() {  
    X::f2();        // f2 de la classe X  
    X::_xxx = 12;   // accès au membre xxx de la classe X  
    f1();           // appel de f1 de la classe X  
    f2();           // appel de f2 de la classe Y  
}
```

8 Héritage

8.5 Héritage des constructeurs/destructeurs

Les constructeurs, constructeur par copie, destructeurs et opérateurs d'affectation ne sont jamais hérités.

Les constructeurs par défaut des classes de bases sont automatiquement appelés avant le constructeur de la classe dérivée.

Pour ne pas appeler les constructeurs par défaut, mais des constructeurs avec des paramètres, vous devez employer une liste d'initialisation.

L'appel des destructeurs se fera dans l'ordre inverse des constructeurs.

8 Héritage

8.5 Héritage des constructeurs/destructeurs

```
class Vehicule {
public:
    Vehicule() { cout<< "Vehicule" << endl; }
    ~Vehicule() { cout<< "~Vehicule" << endl; }
};
```

```
class Voiture : public Vehicule {
public:
    Voiture() { cout<< "Voiture" << endl; }
    ~Voiture() { cout<< "~Voiture" << endl; }
};
```

```
void main() {
    Voiture *R21 = new Voiture;
    // ...
    delete R21;
}
```

```
/*-- résultat de l'exécution -----*/
Vehicule
Voiture
~Voiture
~Vehicule
```

8 Héritage

8.5 Héritage des constructeurs/destructeurs

```
class Vehicule {
    public:
        Vehicule(char *nom, int places);
        //...
};

class Voiture : public Vehicule {
    private:
        int _cv;    // puissance fiscale
    public:
        Voiture(char *n, int p, int cv);
        // ...
};

Voiture::Voiture(char *n, int p, int cv): Vehicule(n, p), _cv(cv) {
    /* ... */
}
```

8

Héritage

8.6

Conversion de type dans une hiérarchie de classes

Il est possible de convertir implicitement une instance d'une classe dérivée en une instance de la classe de base si l'héritage est public.

L'inverse est interdit car le compilateur ne saurait pas comment initialiser les membres de la classe dérivée.

```
class Vehicule {  
    public:  
        void f1();  
        // ...  
};
```

```
class Voiture : public Vehicule {  
    public:  
        int f1() {  
            // ...  
        }  
};
```

```
void traitement1(Vehicule v) {  
    // ...  
    v.f1();    // OK  
    // ...  
}  
void main() {  
    Voiture R25;  
    traitement1( R25 );  
}
```

8

Héritage

8.6

Conversion de type dans une hiérarchie de classes

De la même façon on peut utiliser des pointeurs :

Un pointeur (ou une référence) sur un objet d'une classe dérivée peut être implicitement converti en un pointeur (ou une référence) sur un objet de la classe de base.

Cette conversion n'est possible que si l'héritage est public, car la classe de base doit posséder des membres public accessibles (ce n'est pas le cas d'un héritage *protected* ou *private*). C'est le type du pointeur qui détermine laquelle des méthodes `f1()` est appelée.

```
void traitement1(Vehicule *v){
    // ...
    v->f1();    // OK
    // ...
}

void main() {
    Voiture R25;
    traitement1( &R25 );
}
```


L'héritage nous permet de réutiliser le code écrit pour la classe de base dans les autres classes de la hiérarchie des classes de votre application.

Le polymorphisme rendra possible l'utilisation d'une même instruction pour appeler dynamiquement des méthodes différentes dans la hiérarchie des classes.

En C++, le polymorphisme est mis en oeuvre par l'utilisation des fonctions virtuelles.

8

Héritage

8.7

Polymorphisme

```
#include <iostream>
using namespace std;

class Point {
protected :    // pour que _x et _y soient accessible de Pointcol
    int _x, _y ;
public :
    Point(int abs = 0, int ord = 0) { _x = abs ; _y = ord ; }
    virtual void Affiche() { cout << "Point " << _x << " " << _y << "\n"; }
};

class PointCol : public Point {
    short _couleur ;
public :
    PointCol(int abs, int ord, short cl) : Point(abs, ord) { _couleur = cl ;}
    void Affiche(){ cout << "PointCol " << _x <<" " << _y <<" " << _couleur << "\n"; }
};

class PointInutile : public Point {
public :
    PointInutile(int abs, int ord) :Point(abs, ord) { }
};
```

8

Héritage

8.7

Polymorphisme

```
main()
{
    Point p(3,5), *adp = &p ;
    PointCol pc(8,6,2), *adpc = &pc ;
    PointInutile pi(2,2), *adpi = &pi ;

    adp->Affiche() ;
    adpc->Affiche() ;
    adpi->Affiche() ;
    adp = adpc ;                // l'inverse serait rejeté
    adp->Affiche() ;
    adpc->Affiche() ;
}
```

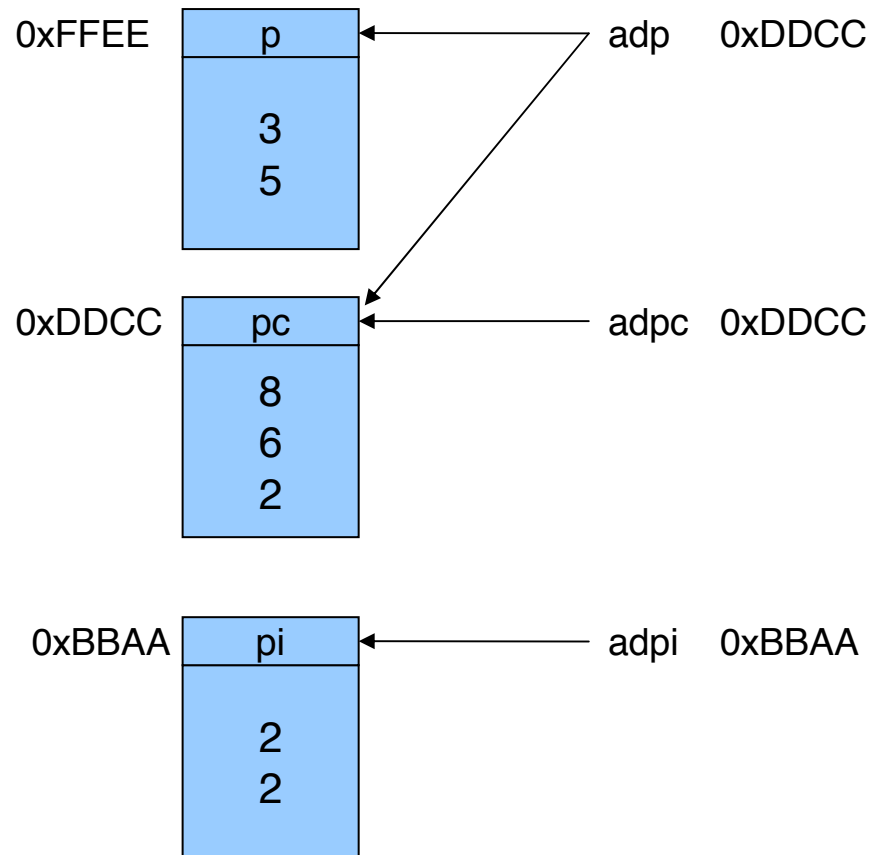
```
/*-- résultat de l'exécution -----*/
Point 3 5
PointCol 8 6 2
Point 2 2
PointCol 8 6 2
PointCol 8 6 2
```

8

Héritage

8.7

Polymorphisme



8

Héritage

8.8

Destructeur virtuel

Il ne faut pas oublier de définir le destructeur comme "virtual" lorsque l'on utilise une méthode virtuelle :

```
class Point {
public :
    virtual ~Point()                { cout << "fin de Point\n"; }
};
class PointCol : public Point {
public :
    virtual ~PointCol()             { cout << "fin de PointCol\n"; }
};

void main() {
    PointCol *adpc = new PointCol ;
    Point      *adp  = adpc ;
    //...
    delete adp ;
}
```

```
/*-- résultat de l'exécution -----*/
fin de PointCol
fin de Point

// si le destructeur n'avait pas été virtuel,
// l'affichage aurait été :
fin de Point
```



Un constructeur, par contre, ne peut pas être déclaré comme virtuel.

8

Héritage

8.9

Classes abstraites

Il arrive souvent que la méthode virtuelle définie dans la classe de base serve de cadre générique pour les méthodes virtuelles des classes dérivées. Ceci permet de garantir une bonne homogénéité de l'architecture de classes.

Une classe est dite abstraite si elle contient au moins une méthode virtuelle pure.

On ne peut pas créer d'instance d'une classe abstraite et une classe abstraite ne peut pas être utilisée comme argument ou type de retour d'une fonction. Par contre, les pointeurs et les références sur une classe abstraite sont parfaitement légitimes et justifiés.

Une méthode virtuelle pure se déclare en ajoutant un = 0 à la fin de sa déclaration.

```
class ObjetGraphique {
    public:
        virtual void Affiche() = 0;
};
class Cercle : public ObjetGraphique {
    public :
        void Affiche() ;
} ;
Void main() {
    ObjetGraphique p(3, 5);    // ERREUR
    Cercle c ;                 // OK
}
```

- ❖ On ne peut utiliser une classe abstraite qu'à partir d'un pointeur ou d'une référence.
- ❖ Contrairement à une méthode virtuelle "normale", une méthode virtuelle pure n'est pas obligée de fournir une définition pour `Point::Affiche()`.
- ❖ Une classe dérivée qui ne redéfinit pas une méthode virtuelle pure est elle aussi abstraite.

Le mécanisme de généricité permet :

- ❖ d'écrire une seule définition d'une classe pour des classes peu différentes, comme vecteur de flottants ou vecteur de complexes (classes paramétrées),
- ❖ d'écrire des fonctions génériques indépendantes de la classe des objets qu'elles traitent comme un algorithme de tri qui doit pouvoir traiter des vecteurs d'entiers comme des vecteurs de complexes.

C++ dispose du mécanisme des patrons ("*template*") pour supporter la généricité.

`template <liste_d_arguments> déclaration`

La déclaration doit être une déclaration de classe ou de fonction. Les arguments sont :

- un type **`<class C>`**
- une valeur **`<int taille>`**
- ou une combinaison des deux **`<class C, int taille>`**

La liste d'arguments ne peut pas être vide.


```
template <class T> class vector {
    T *_v;
    int _sz;
public :
    vector(int size) { _v = new T[_sz = size]; }
    ~vector();
    T& operator[] (int);
};

template <class T> vector<T>::~~vector() { delete [] _v; }

template <class T> T& vector<T>::operator[] (int i) { return(_v[i]); }

int main() {
    vector <int> v1(20);                // vecteur de 20 entiers
    vector <complex> v2(30);            // vecteur de 30 complexes
    v1[7] = 3;
    v2[1] = (1.0, 2.0);
    v1[1] = v2[1];                      // Erreur 1 complexe n'est pas 1 entier
    return(0);
}
```

Le type d'une classe instance d'un patron est :

`nom_class_template <arguments>`

Sur l'exemple précédent, `v2` est de type `vector <complex>`. Ce type est utilisable partout où un type de base est utilisable :

- pour la déclaration d'un pointeur : `const vector<complex> *pvi;`

- pour la dérivation d'une classe :

`class vectorc : public vector<complex> {};`

- pour la définition d'un paramètre de fonction :

`void sort(vector<complex> vc) {};`

Remarques :

- mis en œuvre à la compilation, pas à l'exécution (pas de pertes de performances),
- le compilateur génère des versions du patron adaptées à chaque type de patron utilisé,
- Les patrons sont souvent utilisées pour des classes conteneurs (listes, tableaux, piles, etc.),
- une méthode d'une classe générique est implicitement une fonction générique (voir ex. `vector`) avec comme arguments les arguments du modèle de la classe.

9

LA GENERICITE

9.3

FONCTIONS GENERIQUES

Lors de l'appel de la fonction, il n'est pas nécessaire d'instancier les fonctions génériques, la résolution de surcharge déterminant la fonction qui convient.

```
template <class T> T max(const T& a, const T& b) {return a>b?a:b;}
main() {
    int a, b;
    char c, d;
    int i1 = max(a, b); // Fonction max avec T pris pour int
    char c = max(c, d); // Fonction max avec T pris pour char
    int i2 = max(a, c); // Erreur ???
}
```

Tous les paramètres de généricité doivent être utilisés dans la liste des paramètres de la fonction.

```
template <class T> T* allocation();           // erreur
template <class T> void allocation(T*);       // ok
template <int i> void allocation(int [][]i)); // erreur
template <int i> void allocation(int = i);     // erreur
template <class T, class C> void f(T);         // erreur
```

Deux instances issues d'un même modèle de classe sont du même type si tous les paramètres du modèle ont la même valeur. Exemple :

```
template <class T, int size> class tampon;  
tampon <char, 2*512> x;  
tampon <char, 1024> y;  
tampon <char, 256> z;
```

x et y sont du même type, z d'un autre type. Il est nécessaire dans certains cas de fournir une implémentation explicite pour un type donné d'une fonction membre *template*. L'instance spécialisée est alors appelée à la place de l'instance par défaut. Par exemple :

```
char *c1 = "supérieur";  
char *c2 = "inférieur";  
char *c = max(c1, c2);
```

Cet exemple ne donne pas le résultat escompté car l'opérateur "<" s'applique sur des pointeurs et ne compare pas les chaînes. Solution : fournir une instance spécialisée pour les char*.

```
char * max(char *a, char *b) {  
    if strcmp(a, b) < 0 return(a) else return(b);  
};
```

10 C++ 11 : Standard ANSI/ISO

Après plusieurs années d'efforts des comités de normalisation, un C++ standard est défini en 1998 (avec une version corrigée en 2003) puis une version améliorée du standard est ratifiée en 2011. Il apporte :

- ❖ des clarifications sur le langage lui-même,
- ❖ des extensions au langage,
- ❖ une librairie standard.

Les extensions du langage :

- ❖ templates,
- ❖ exceptions,
- ❖ run time type information (RTTI),
- ❖ namespaces.

10 C++ Standard ANSI/ISO


10.1 Librairie standard

La librairie comporte les parties suivantes :

- ❖ **support à l'exécution du langage : allocation dynamique de la mémoire (new et delete), gestion des exceptions, informations de type à l'exécution.**
- ❖ **exceptions prédéfinies,**
- ❖ **chaînes de caractères (classes char, wchar_t, string, wstring),**
- ❖ **localisation : supports de l'internationalisation (représentation des caractères, des nombres, ponctuation, format des heures et des dates).**
- ❖ **numérique : calcul numérique (complexes, fonctions mathématiques diverses, vecteurs avec algèbre linéaire (BLAS))**
- ❖ **iostream : composants d'entrée/sortie**
- ❖ **librairie C adaptée au C++**

10 C++ Standard ANSI/ISO

10.1 Librairie standard

- ❖ **containers (STL) : collections d'objets (list, vector, stack, queue, set, map, etc.)**
- ❖ **itérateurs (STL) : déplacement dans les containers et les streams.**
- ❖ **algorithmes (STL) : algorithmes sur les containers et d'autres séquences**
 - ❖ **opérations non modificatrices (find, count, search)**
 - ❖ **opérations modificatrices (copy, swap, replace, fill, remove, reverse, rotate)**
 -  ❖ **opération de tri (sort), opération de fusion (merge)**
- ❖ **Multithreading : exécution parallèle de code (thread) avec partage de mémoire et gestion de la concurrence (mutex)**
- ❖ **Expression régulière**