Résumé PRG1

Nicolas Reymond

December 18, 2024

Contents

1 Résumé : 01_Introduction

1.1 Organisation du cours

Le cours PRG1 utilise :

- Supports disponibles sur Moodle.
- Basé sur le livre *Programmer en C++ moderne — de C++11 à C++20*.
- Langage enseigné : **C++20**.
- Compilation et IDE :
  - Compilateur recommandé : `g++`.
  - Environnement optionnel : Visual Studio, CLion, etc.

1.2 Objectifs du cours

Apprendre les bases de la programmation en C++ :

- Comprendre la syntaxe du langage.
- Manipuler des types de données, des instructions, des fonctions et des classes.
- Développer une logique algorithmique.
- Mettre en place des programmes fiables, modulaires et efficaces.

1.3 Un premier programme en C++

Voici un exemple de programme simple en C++ :

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

int main() {
    cout << "Bonjour, PRG1 !" << endl;
    return 0;
}
```

Explications :

- `#include <iostream>` : permet d'utiliser `cout` et `cin`.
- `using namespace std;` : simplifie l'écriture (on évite d'écrire `std::cout`).
- `int main()` : point d'entrée du programme.
- `cout << "Bonjour";` : affiche du texte à l'écran.
- `return 0;` : indique que le programme s'est terminé correctement.

1.4 Étapes de la compilation

La compilation d'un programme C++ se fait en plusieurs étapes :

1. **Préprocesseur** : gère les directives `#include` et `#define`.
2. **Compilation** : transforme le code source en code machine.
3. **Édition de liens** : combine plusieurs fichiers objets pour créer un exécutable.

1.5 Erreurs fréquentes

- **Erreurs de syntaxe** : oublis de `;` ou d'accolades.
- **Erreurs de compilation** : utilisation de types incompatibles.
- **Erreurs de lien** : fonctions manquantes ou bibliothèques non incluses.

Exemple d'erreur classique :

```
int main() {
    cout << "Bonjour" // Manque le point-virgule
    return 0;
}
```

2 Résumé: 02_Bases et Opérateurs

2.1 Identificateurs et Mots-clés

- Les **identificateurs** sont les noms donnés aux variables, fonctions, etc. :
  - Doivent commencer par une lettre ou un `_`.
  - Ne doivent pas contenir d'espaces ou de caractères spéciaux.
  - Exemple : `maVariable`, `_compteur`.
- **Mots-clés** réservés par C++ (ne peuvent pas être utilisés comme identificateurs) :
  - Exemple : `int`, `return`, `while`.

2.2 Notion de Types

En C++, chaque variable a un **type**. Voici les types principaux :

- **Entiers** : `int`, `short`, `long`.
- **Flottants** : `float`, `double`.
- **Caractères** : `char`.
- **Booléens** : `bool` (`true` ou `false`).

Exemple :

```
int age = 25;       // Entier
float taille = 1.8; // Nombre décimal
bool majeur = true; // Booléen
```

2.3 Variables et Constantes

- **Variable** : zone mémoire dont la valeur peut changer.
- **Constante** : valeur fixe, définie avec `const`.

Exemple :

```
const double PI = 3.14; // Constante
int score = 100;        // Variable
```

2.4 Opérateurs et Expressions

Les **opérateurs** sont utilisés pour manipuler les données. Voici les principaux :

- **Arithmétiques** : `+`, `-`, `*`, `/`, `%`.
- **Relationnels** : `<`, `>`, `<=`, `>=`, `==`, `!=`.
- **Logiques** : `&&`, `||`, `!`.
- **Assignation** : `=`, `+=`, `-=`, `*=`, `/=`.

Exemple :

```
int a = 5, b = 2;
int somme = a + b;    // Résultat : 7
bool test = a > b;    // Résultat : true
```

2.5 Priorité des Opérateurs

Les opérateurs ont une priorité. Par exemple :

- `*` et `/` sont évalués avant `+` et `-`.
- Utilisez des **parenthèses** pour clarifier les expressions.

Exemple :

```
int resultat = 5 + 3 * 2;       // Résultat : 11 (3 * 2 évalué en premier)
int resultat2 = (5 + 3) * 2;    // Résultat : 16 (parenthèses prioritaires)
```

2.6 Cast (Conversions de Type)

Pour changer le type d'une variable, utilisez un **cast** :

```
int a = 5;
double b = (double)a; // Convertit a en double
```

2.7 Exemples d'Utilisation

```
// Calcul simple
int a = 10, b = 3;
cout << "Somme : " << (a + b) << endl;  // Affiche 13
cout << "Division : " << (a / b) << endl;  // Affiche 3 (division entière)
cout << "Modulo : " << (a % b) << endl;  // Affiche 1
```

3 Résumé : 03_Instructions de Contrôle

3.1 Structure Séquentielle

Par défaut, les instructions d'un programme sont exécutées dans l'ordre, de haut en bas.

Exemple :

```
int a = 5;
cout << "Valeur de a : " << a << endl;
// Résultat : Valeur de a : 5
```

3.2 Structures Conditionnelles

Les structures conditionnelles permettent de prendre des décisions dans le programme.

3.2.1 Instruction `if-else`

- Exécute un bloc d'instructions si une condition est vraie.

Syntaxe :

```
if (condition) {
    // Instructions si condition est vraie
} else {
    // Instructions si condition est fausse
}
```

Exemple :

```
int age = 20;
if (age >= 18) {
    cout << "Vous êtes majeur." << endl;
} else {
    cout << "Vous êtes mineur." << endl;
}
```

3.2.2 Instruction `switch`

- Permet de choisir parmi plusieurs options.
- Utilise des `case` et un `default`.

Syntaxe :

```
switch (variable) {
    case valeur1:
        // Instructions
        break;
    case valeur2:
        // Instructions
        break;
    default:
        // Instructions par défaut
}
```

Exemple :

```
int jour = 3;
switch (jour) {
    case 1:
        cout << "Lundi" << endl;
        break;
    case 2:
        cout << "Mardi" << endl;
        break;
    default:
        cout << "Autre jour" << endl;
}
```

3.3 Structures Itératives

Les boucles permettent de répéter un bloc d'instructions plusieurs fois.

3.3.1 Boucle `while`

- Exécute tant qu'une condition est vraie.

Syntaxe :

```
while (condition) {
    // Instructions
}
```

Exemple :

```
int i = 0;
while (i < 5) {
    cout << i << endl;
    i++;
}
```

3.3.2 Boucle `do-while`

- Exécute au moins une fois, puis tant que la condition est vraie.

Syntaxe :

```
do {
    // Instructions
} while (condition);
```

Exemple :

```
int i = 0;
do {
    cout << i << endl;
    i++;
} while (i < 5);
```

3.3.3 Boucle `for`

- Boucle compacte avec initialisation, condition et incrémentation.

Syntaxe :

```
for (initialisation; condition; incrément) {
    // Instructions
}
```

Exemple :

```
for (int i = 0; i < 5; i++) {
    cout << i << endl;
}
```

3.4 Instructions de Saut

- `break` : Sort de la boucle ou du `switch`.
- `continue` : Passe directement à l'itération suivante.
- `goto` : Saute à une étiquette (peu recommandé).

Exemple d'utilisation de `break` :

```
for (int i = 0; i < 10; i++) {
    if (i == 5) break; // Sort de la boucle si i vaut 5
    cout << i << endl;
}
```

Exemple d'utilisation de `continue` :

```cpp
for (int i = 0; i < 5; i++) {
    if (i == 2) continue; // Passe directement à i = 3
    cout << i << endl;
}
```

3.5 Résumé visuel des instructions de contrôle

4 Résumé : 04_Fonctions

4.1 Rôle des Fonctions

Les fonctions permettent :

- De structurer le code pour le rendre plus lisible et réutilisable.
- De réaliser des opérations spécifiques (calculs, affichage, etc.).
- D'éviter la duplication de code.

Exemple :

```cpp
double addition(double a, double b) {
    return a + b;
}

int main() {
    double resultat = addition(3.5, 2.2);
    cout << "Résultat : " << resultat << endl;
    return 0;
}
```

4.2 Déclaration et Définition

- Une **déclaration** informe le compilateur de l'existence de la fonction.
- Une **définition** contient le code de la fonction.

Exemple :

```cpp
// Déclaration
double addition(double a, double b);

// Définition
double addition(double a, double b) {
    return a + b;
}
```

4.3 Arguments et Paramètres

- Les **paramètres** sont des variables locales définies dans la fonction.
- Les **arguments** sont les valeurs passées à la fonction.

Exemple :

```cpp
void afficherMessage(string message) { // Paramètre : message
    cout << message << endl;
}

int main() {
    afficherMessage("Bonjour, PRG1 !"); // Argument : "Bonjour, PRG1 !"
    return 0;
}
```

4.4 Transmission par Valeur ou Référence

- **Par valeur** : une copie de l'argument est faite (ne modifie pas l'original).
- **Par référence** : permet de modifier la variable originale.

Exemple :

```cpp
// Par valeur
void incrementerValeur(int n) {
    n++;
}

// Par référence
void incrementerReference(int &n) {
    n++;
}
```

Différence à l'exécution :

```cpp
int main() {
    int a = 5;
    incrementerValeur(a);   // a reste 5
    incrementerReference(a); // a devient 6
}
```

4.5 Valeurs de Retour

- Les fonctions peuvent retourner une valeur avec `return`.
- Si aucune valeur n'est retournée, le type `void` est utilisé.

Exemple :

```cpp
int carre(int x) {
    return x * x;
}

void afficherMessage() {
    cout << "Message sans retour." << endl;
}
```

4.6 Arguments par Défaut

Les arguments peuvent avoir des valeurs par défaut dans leur déclaration.

Exemple :

```cpp
void saluer(string nom = "Utilisateur") {
    cout << "Bonjour, " << nom << " !" << endl;
}

int main() {
    saluer();              // Affiche : Bonjour, Utilisateur !
```

```
    saluer("Alice");       // Affiche : Bonjour, Alice !
}
```

4.7 Fonction Récursive

Une fonction peut s'appeler elle-même (récursivité).

Exemple : Calcul Factoriel

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

int main() {
    cout << "Factoriel de 5 : " << factoriel(5) << endl;
    return 0;
}
```

4.8 Résumé des Concepts

5 Résumé : 05_Type String

5.1 Introduction au Type `string`

Le type `string` en C++ permet de manipuler facilement des chaînes de caractères.

- Défini dans la bibliothèque standard `<string>`.
- Remplace les chaînes de style C (`char[]`).

Exemple :

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

string nom = "Alice";
cout << "Bonjour, " << nom << " !" << endl; // Affiche : Bonjour, Alice !
```

—

5.2 Déclaration et Initialisation

- Chaîne vide : `string ch1; // ch1 est vide`.
- Initialisation par une chaîne littérale : `string ch2 = "Bonjour";`.
- Initialisation par répétition : `string ch3(10, 'x'); // xxxxxxxxxx`.

Exemple :

```
string vide;
string chaine = "Hello";
string repetition(5, '*'); // ***** (5 étoiles)
```

—

5.3 Méthodes Utiles pour les Chaînes

- `size()` ou `length()` : donne la taille.
- `empty()` : vérifie si la chaîne est vide.
- `at(pos)` ou `operator[]` : accès au caractère à la position `pos`.
- `substr(pos, len)` : extrait une sous-chaîne.
- `append()` ou `+=` : ajoute à la fin.
- `erase(pos, len)` : supprime une portion.

Exemple :

```
string texte = "Programmation";
cout << "Taille : " << texte.size() << endl;  // Taille : 13
cout << texte.at(0) << texte[1] << endl;      // Pr
cout << texte.substr(0, 7) << endl;           // Program
texte.append(" 1");                           // Ajoute " 1" à la fin
texte.erase(0, 4);                            // Supprime "Prog"
```

—

5.4 Concaténation et Comparaison

- **Concaténation :** Utilisez `+` ou `+=`.
- **Comparaison :** Utilisez `==`, `!=`, `<`, `>`, etc.

Exemple :

```
string a = "Bonjour", b = "Monde";
string c = a + " " + b;  // Bonjour Monde
cout << (a == "Bonjour"); // true
cout << (b > "Monde");    // false
```

—

5.5 Lecture et Écriture

- **Lecture ligne complète :** `getline(cin, chaine)`.
- **Lecture mot à mot :** `cin >> chaine`.

Exemple :

```
string nom, phrase;
cout << "Quel est votre nom ? ";
cin >> nom; // Lit un mot
cin.ignore(); // Ignorer le saut de ligne
cout << "Entrez une phrase complète : ";
getline(cin, phrase);
```

—

5.6 Conversion de Numériques en Chaînes et Inversement

- **Chaîne vers numérique :** Utilisez `stoi()`, `stod()`, etc.
- **Numérique vers chaîne :** Utilisez `to_string()`.

Exemple :

```
string chaine = "123";
int nombre = stoi(chaine);       // Convertit en entier
double reel = stod("12.34");     // Convertit en réel
string texte = to_string(45.67); // "45.67"
```

—

5.7 Exemples Pratiques

Manipulation de texte :

```
string phrase = "Programmation C++";
phrase.insert(13, " en");   // Programmation en C++
phrase.replace(0, 12, "Cours"); // Cours en C++
phrase.pop_back();          // Supprime le dernier caractère
cout << phrase;             // Cours en C+
```

5.8 Résumé visuel des fonctionnalités

6 Résumé : 06_Les Flux

6.1 Introduction aux Flux

Un flux est un canal permettant d'envoyer ou de recevoir des données.

- Flux de sortie : envoi des données vers un périphérique (`cout`).
- Flux d'entrée : réception des données depuis un périphérique (`cin`).

Exemple :

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

int main() {
    int nombre;
    cout << "Entrez un nombre : ";
    cin >> nombre;
    cout << "Vous avez entré : " << nombre << endl;
    return 0;
}
```

—

6.2 Les Flux Prédéfinis

- `cout` : sortie standard (écran).
- `cin` : entrée standard (clavier).
- `cerr` : sortie standard sans tampon (affichage immédiat des erreurs).
- `clog` : sortie standard avec tampon (messages de log).

—

6.3 Manipulateurs de Flux

Les manipulateurs permettent de formater l'affichage dans les flux.

6.3.1 3.1 Manipulateurs pour les Entiers

- `dec` : base décimale (par défaut).
- `hex` : base hexadécimale.
- `oct` : base octale.
- `showpos` : affiche le signe `+` pour les nombres positifs.

Exemple :

```
int nombre = 255;
cout << dec << nombre << endl;  // 255
cout << hex << nombre << endl;  // ff
cout << oct << nombre << endl;  // 377
```

6.3.2 3.2 Manipulateurs pour les Nombres Réels

- `fixed` : notation fixe (6 chiffres après la virgule par défaut).
- `scientific` : notation scientifique.
- `setprecision(n)` : spécifie le nombre de chiffres significatifs.
- `showpoint` : force l'affichage des chiffres après la virgule.

Exemple :

```
#include <iomanip>
double pi = 3.14159;
cout << fixed << setprecision(2) << pi << endl;  // 3.14
cout << scientific << pi << endl;                // 3.14e+00
```

—

6.4 Lecture au Clavier (`cin`)

- `cin >> variable` : lit une entrée (ignore les espaces).
- `getline(cin, chaine)` : lit une ligne complète.

Exemple :

```
#include <string>
string nom;
cout << "Entrez votre nom : ";
getline(cin, nom);
cout << "Bonjour, " << nom << " !" << endl;
```

—

6.5 Flux Fichiers

Pour lire ou écrire des fichiers, on utilise les classes suivantes :

- `ofstream` : flux de sortie vers un fichier.
- `ifstream` : flux d'entrée depuis un fichier.
- `fstream` : flux d'entrée et de sortie.

6.5.1 5.1 Écriture dans un Fichier

```
#include <fstream>
ofstream fichier("sortie.txt");
fichier << "Hello, fichier !" << endl;
fichier.close();
```

6.5.2 5.2 Lecture depuis un Fichier

```
#include <fstream>
ifstream fichier("entree.txt");
string ligne;
while (getline(fichier, ligne)) {
    cout << ligne << endl;
}
fichier.close();
```

—

6.6 Manipulation Avancée des Fichiers

- `seekg(pos)` : déplace le pointeur de lecture.
- `seekp(pos)` : déplace le pointeur d'écriture.
- `tellg()` : donne la position actuelle du pointeur de lecture.
- `tellp()` : donne la position actuelle du pointeur d'écriture.

Exemple : Lecture Inversée d'un Fichier :

```
#include <fstream>
ifstream fichier("texte.txt", ios::ate); // Ouvert en mode "fin de fichier"
streampos taille = fichier.tellg();     // Taille du fichier
for (int i = taille - 1; i >= 0; --i) {
    fichier.seekg(i);
    char c;
    fichier.get(c);
    cout << c;
}
fichier.close();
```

—

6.7 Résumé Visuel des Flux

7 Résumé: 07_Types Numériques

7.1 Types Numériques en C++

En C++, les types numériques se divisent en deux grandes catégories :

- **Types entiers :** pour les nombres sans décimales.
- **Types flottants :** pour les nombres avec décimales.

Exemples :

```
int entier = 42;           // Type entier
double flottant = 3.14;    // Type flottant
```

—

7.2 Tableau des Types Numériques

Type	Taille (bits)	Plage de Valeurs	Exemple
`char`	8	-128 à 127	`char c = 'A';`
`short`	16	-32,768 à 32,767	`short s = 100;`
`int`	32	-2,147,483,648 à 2,147,483,647	`int i = 123;`
`long`	64	Très grand	`long l = 1234567890;`
`float`	32	$\pm 10^{-38}$ à $\pm 10^{38}$	`float f = 3.14;`
`double`	64	$\pm 10^{-308}$ à $\pm 10^{308}$	`double d = 3.14159;`

—

7.3 Spécificateurs de Type

Les spécificateurs permettent d'ajuster la taille ou la plage des types numériques.

- `unsigned` : supprime les nombres négatifs.
- `long` : augmente la capacité.

Exemples :

```
unsigned int positif = 42;  // Seulement des nombres positifs
long long tresGrand = 123456789012345;
```

—

7.4 Casting (Conversion de Type)

- **Cast implicite :** automatique quand la conversion est sûre.
- **Cast explicite :** forcé par le programmeur.

Exemples :

```
int entier = 42;
double flottant = entier;          // Implicite
double pi = 3.14;
int arrondi = (int)pi;             // Explicite
```

—

7.5 Opérations sur les Types Numériques

Les opérateurs mathématiques s'appliquent directement :

- **Arithmétiques :** `+`, `-`, `*`, `/`, `%`.
- **Relationnels :** `==`, `!=`, `<`, `>`.

Exemples :

```
int a = 10, b = 3;
cout << a + b;  // Affiche 13
cout << a / b;  // Affiche 3 (division entière)
cout << a % b;  // Affiche 1 (modulo)
```

—

7.6 Limites des Types Numériques

La bibliothèque `<limits>` permet de connaître les limites d'un type.

Exemple :

```
#include <limits>
cout << "Max int : " << numeric_limits<int>::max() << endl;
cout << "Min double : " << numeric_limits<double>::min() << endl;
```

—

7.7 Exemples Pratiques

Calcul de Moyenne :

```
int a = 5, b = 8, c = 12;
double moyenne = (a + b + c) / 3.0;  // Division réelle
cout << "Moyenne : " << moyenne;
```

Vérification de Dépassement :

```
int a = numeric_limits<int>::max();
cout << "a : " << a << endl;
a = a + 1; // Dépassement : résultat indéfini
cout << "a après dépassement : " << a << endl;
```

—

7.8 Résumé Visuel des Types Numériques

8 Résumé : 08_Type Vector

8.1 Introduction au `std::vector`

Le type `std::vector`, défini dans `<vector>`, est une structure de données dynamique qui permet de stocker une collection d'éléments.

- Similaire à un tableau (`array`), mais de taille dynamique.
- Permet des opérations avancées comme l'ajout ou la suppression d'éléments.

Exemple :

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

vector<int> nombres = {1, 2, 3, 4, 5};
cout << "Premier élément : " << nombres[0] << endl;
```

—

8.2 Création et Initialisation

- Déclaration vide : `vector<int> vec;`.
- Initialisation avec des valeurs : `vector<int> vec = {1, 2, 3};`.
- Taille et valeur initiale : `vector<int> vec(5, 0); // 5 zéros`.

Exemples :

```
vector<int> vide;               // Vecteur vide
vector<int> valeurs = {10, 20}; // Initialisé avec des valeurs
vector<int> zeros(5, 0);        // 5 zéros
```

—

8.3 Accès aux Éléments

- `operator[]` : accès direct.
- `at()` : accès sécurisé (vérifie les limites).
- `front()` : premier élément.
- `back()` : dernier élément.

Exemple :

```
vector<int> nombres = {10, 20, 30};
cout << nombres[1];           // 20
cout << nombres.at(2);        // 30
cout << nombres.front();      // 10
cout << nombres.back();       // 30
```

—

8.4 Opérations de Base

- `push_back(val)` : ajoute un élément à la fin.
- `pop_back()` : retire le dernier élément.
- `size()` : retourne la taille.
- `clear()` : supprime tous les éléments.
- `empty()` : vérifie si le vecteur est vide.

Exemple :

```
vector<int> vec = {1, 2, 3};
vec.push_back(4);  // {1, 2, 3, 4}
vec.pop_back();    // {1, 2, 3}
cout << vec.size(); // 3
vec.clear();        // Vide le vecteur
```

—

8.5 Parcours d'un `vector`

- Avec une boucle `for`.
- Avec une boucle `range-based for`.
- Avec des itérateurs.

Exemples :

```
vector<int> nombres = {10, 20, 30};

// Boucle for classique
for (int i = 0; i < nombres.size(); i++) {
    cout << nombres[i] << " ";
}

// Boucle for basée sur la plage
for (int n : nombres) {
    cout << n << " ";
}

// Avec des itérateurs
for (auto it = nombres.begin(); it != nombres.end(); ++it) {
    cout << *it << " ";
}
```

—

8.6 Fonctions Avancées

- `insert(it, val)` : insère un élément à une position donnée.
- `erase(it)` : supprime un élément à une position donnée.
- `resize(n)` : redimensionne le vecteur.

Exemples :

```
vector<int> vec = {10, 20, 30};
vec.insert(vec.begin() + 1, 15); // {10, 15, 20, 30}
vec.erase(vec.begin() + 2);      // {10, 15, 30}
vec.resize(5, 0);                // {10, 15, 30, 0, 0}
```

—

8.7 Comparaison avec les Tableaux

Vecteur (`vector`)	**Tableau** (`array`)
Taille dynamique	Taille fixe
Mémoire allouée dynamiquement	Mémoire allouée statiquement
Méthodes intégrées (`push_back`, etc.)	Pas de méthodes supplémentaires
Plus lent (gestion dynamique)	Plus rapide (gestion fixe)

—

8.8 Exemples Pratiques

Tri d'un vecteur :

```
#include <algorithm>
vector<int> vec = {4, 2, 8, 6};
sort(vec.begin(), vec.end()); // {2, 4, 6, 8}
```

Rechercher un élément :

```
#include <algorithm>
vector<int> vec = {10, 20, 30};
if (find(vec.begin(), vec.end(), 20) != vec.end()) {
    cout << "20 trouvé !";
}
```

—

8.9 Résumé Visuel des Méthodes

9 Résumé : 09_Surcharge et Fonctions Génériques

9.1 Qu'est-ce que la Surcharge de Fonctions ?

La surcharge de fonctions permet de définir plusieurs fonctions ayant le même nom, mais des paramètres différents.

- Cela améliore la lisibilité et la flexibilité du code.
- Le compilateur choisit la bonne version de la fonction en fonction des arguments fournis.

Exemple :

```cpp
void afficher(int x) {
    cout << "Entier : " << x << endl;
}

void afficher(double x) {
    cout << "Flottant : " << x << endl;
}

void afficher(string x) {
    cout << "Texte : " << x << endl;
}

int main() {
    afficher(42);         // Appelle afficher(int)
    afficher(3.14);       // Appelle afficher(double)
    afficher("Bonjour"); // Appelle afficher(string)
}
```

—

9.2 Règles pour la Surcharge de Fonctions

- Les fonctions doivent avoir des signatures différentes (nombre, type ou ordre des paramètres).
- Le retour de la fonction n'est pas pris en compte dans la surcharge.

Exemple Incorrect :

```cpp
int addition(int a, int b);
double addition(int a, int b); // Erreur : signature identique
```

—

9.3 Surcharge d'Opérateurs

En C++, il est possible de redéfinir les opérateurs pour des types définis par l'utilisateur (par exemple, des classes).

- Utilisé pour rendre les objets de classes manipulables comme des types natifs.
- Les opérateurs sont redéfinis en utilisant la fonction membre `operator`.

Exemple :

```cpp
class Point {
public:
    int x, y;

    Point(int a, int b) : x(a), y(b) {}

    // Surcharge de l'operateur +=
    Point operator+=(const Point &p) {
        return Point(x + p.x, y + p.y);
    }

    // Surcharge de l'operateur +
    Point operator+(const Point_droit &p, const Point_gauche &q) {
        return Point(p.x + q.x, p.y + q.y);
    }

    void afficher() {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};
```

```
21
22 int main() {
23     Point p1(2, 3), p2(4, 5);
24     Point p2 += p1;      // Utilise operator+=
25     Point p3 = p1 + p2; // Utilise operator+
26     p2.afficher();       // Affiche (6, 8)
27 }
```

9.4 Surcharge de l'opérateur de flux

Il est possible de surcharger l'opérateur de flux `<<` pour permettre l'affichage des objets d'une classe. Cette surcharge personnalise le comportement de `cout` (ou tout autre flux de sortie) pour afficher une représentation spécifique d'un objet.

9.4.1 1. Pourquoi surcharger << ?

Par défaut, `cout` ne sait pas comment afficher les objets définis par l'utilisateur. La surcharge de `<<` permet de définir une représentation adaptée.

Exemple (sans surcharge) :

```
1 Point p(2, 3);
2 cout << p; // Erreur : pas de définition pour afficher un Point
```

En surchargeant `<<`, nous pouvons afficher un objet comme suit :

```
1 Point p(2, 3);
2 cout << p; // Affiche : (2, 3)
```

9.4.2 2. Syntaxe de la surcharge

Pour surcharger `<<`, on définit une fonction globale ou amie de la classe :

```
1 ostream& operator<<(ostream &out, const Point &p);
```

- `ostream&` : La fonction retourne une référence au flux pour permettre le chaînage.
- `ostream &out` : Flux de sortie (comme `cout`).
- `const Point &p` : Objet à afficher, passé par référence constante.

9.4.3 3. Exemple complet

```
1  #include <iostream>
2  using namespace std;
3
4  // Classe Point
5  class Point {
6  private:
7      int x, y;
8
9  public:
10     // Constructeur
11     Point(int a, int b) : x(a), y(b) {}
12
13     // Surcharge de l'opérateur <<
14     friend ostream& operator<<(ostream &out, const Point &p) {
15         out << "(" << p.x << ", " << p.y << ")";
16         return out; // Permet le chaînage
17     }
18 };
19
```

```cpp
int main() {
    Point p1(2, 3), p2(5, 8);
    cout << p1 << endl; // Affiche : (2, 3)
    cout << p2 << endl; // Affiche : (5, 8)
}
```

9.4.4 4. Explication détaillée

- La fonction `operator<<` est définie comme amie (`friend`) pour accéder aux membres privés de `Point`.
- La fonction utilise `out <<` pour ajouter les données au flux.
- La fonction retourne `out` pour permettre le chaînage :

```cpp
            cout << p1 << " et " << p2;
            // Affiche : (2, 3) et (5, 8)
```

9.4.5 5. Autres opérateurs

De manière similaire, vous pouvez surcharger l'opérateur `>>` pour permettre la saisie d'objets.

Exemple :

```cpp
istream& operator>>(istream &in, Point &p) {
    in >> p.x >> p.y;
    return in;
}

int main() {
    Point p;
    cin >> p; // Saisir deux entiers pour x et y
    cout << p; // Affiche le point saisi
}
```

9.4.6 6. Résumé

La surcharge de l'opérateur `<<` permet de personnaliser l'affichage des objets d'une classe. Elle doit :

- Être déclarée comme fonction globale ou amie.
- Prendre en paramètre un flux de sortie (`ostream &`) et une référence constante à l'objet.
- Retourner le flux pour permettre le chaînage.

—

9.5 Fonctions Génériques avec `templates`

Les **templates** permettent de créer des fonctions génériques qui fonctionnent avec différents types sans redéfinir la fonction.

Syntaxe :

```cpp
template<typename T>
T addition(T a, T b) {
    return a + b;
}
```

Exemple :

```cpp
template<typename T>
T addition(T a, T b) {
    return a + b;
}

int main() {
    cout << addition(5, 3) << endl;        // Utilise int
    cout << addition(2.5, 1.5) << endl;  // Utilise double
    return 0;
}
```

—

9.6 Surcharge de Fonctions Génériques

Les `templates` peuvent être combinés avec des fonctions spécifiques pour des cas particuliers.

Exemple :

```cpp
template<typename T>
void afficher(T valeur) {
    cout << "Valeur generique : " << valeur << endl;
}

// Surcharge pour string
void afficher(string valeur) {
    cout << "Texte : " << valeur << endl;
}

int main() {
    afficher(42);           // Appelle la fonction generique
    afficher("Bonjour");    // Appelle la surcharge pour string
}
```

—

9.7 Templates de Classe

Les `templates` peuvent également être utilisés pour créer des classes génériques.

Exemple :

```cpp
template<typename T>
class Boite {
private:
    T valeur;
public:
    Boite(T v) : valeur(v) {}
    void afficher() {
        cout << "Valeur : " << valeur << endl;
    }
};

int main() {
    Boite<int> boiteEntiere(42);
    Boite<string> boiteTexte("Bonjour");

    boiteEntiere.afficher(); // Valeur : 42
    boiteTexte.afficher();   // Valeur : Bonjour
}
```

—

9.8 Exemples Pratiques

Combinaison de Templates et Surcharge :

```
template<typename T>
T carre(T x) {
    return x * x;
}

// Specialisation pour string
string carre(string x) {
    return x + x; // Repete la chaine
}

int main() {
    cout << carre(5) << endl;         // 25
    cout << carre(3.14) << endl;     // 9.8596
    cout << carre("Hello") << endl; // HelloHello
}
```

10 Résumé: 10_Classes et Objets

10.1 1. Qu'est-ce qu'une Classe ?

Une classe est un modèle (ou blueprint) pour créer des objets. Elle regroupe des données (**attributs**) et des fonctions (**méthodes**) liées.

Exemple :

```
class Rectangle {
private:
    double largeur, hauteur; // Attributs privés

public:
    // Méthodes publiques
    void setDimensions(double l, double h) {
        largeur = l;
        hauteur = h;
    }

    double calculerAire() {
        return largeur * hauteur;
    }
};
```

—

10.2 2. Qu'est-ce qu'un Objet ?

Un objet est une instance d'une classe. Chaque objet possède ses propres valeurs pour les attributs définis par la classe.

Exemple :

```
int main() {
    Rectangle rect; // Création d'un objet
    rect.setDimensions(5.0, 3.0); // Utilisation des méthodes
    cout << "Aire : " << rect.calculerAire(); // Affiche 15.0
    return 0;
}
```

—

10.3 3. Visibilité des Membres

Les membres d'une classe peuvent avoir différents niveaux d'accès :

- **public** : accessible de n'importe où.
- **private** : accessible uniquement dans la classe.
- **protected** : accessible dans la classe et ses dérivées.

Exemple :

```
class Exemple {
public:
    int publicAttr; // Accessible partout
private:
    int privateAttr; // Accessible uniquement dans la classe
protected:
    int protectedAttr; // Accessible dans les classes dérivées
};
```

—

10.4 4. Constructeurs et Destructeurs

- **Constructeur :** Une méthode spéciale appelée lors de la création d'un objet. Il initialise les attributs.
- **Destructeur :** Une méthode spéciale appelée lors de la destruction d'un objet. Elle libère les ressources si nécessaire.

Exemple :

```
class Point {
private:
    int x, y;

public:
    // Constructeur
    Point(int a, int b) : x(a), y(b) {
        cout << "Point créé !" << endl;
    }

    // Destructeur
    ~Point() {
        cout << "Point détruit !" << endl;
    }
};
```

—

10.5 5. Encapsulation

L'encapsulation consiste à restreindre l'accès direct aux attributs d'une classe, en les rendant privés et en fournissant des méthodes pour y accéder (**getters et setters**).

Exemple :

```
class CompteBancaire {
private:
    double solde;

public:
    void deposer(double montant) {
        solde += montant;
    }
```

```
    double obtenirSolde() {
        return solde;
    }
};
```

—

10.6 6. Méthodes Constantes

Une méthode constante (`const`) garantit qu'elle ne modifiera pas les attributs de l'objet.

Exemple :

```
class Point {
private:
    int x, y;

public:
    int obtenirX() const { return x; } // Méthode constante
};
```

—

10.7 7. Attributs et Méthodes Statistiques

Les membres statiques appartiennent à la classe et non à ses instances :

- Les **attributs statiques** sont partagés entre toutes les instances.
- Les **méthodes statiques** ne peuvent accéder qu'aux attributs statiques.

Exemple :

```
class Compteur {
private:
    static int total; // Attribut statique

public:
    static int obtenirTotal() { return total; } // Méthode statique
};

int Compteur::total = 0; // Initialisation
```

11 Résumé: 11 Patrons de Classes

11.1 1. Introduction aux Patrons de Classes

Un patron de classe (**template**) permet de créer des classes génériques. Il facilite la réutilisation du code en permettant à une classe de fonctionner avec différents types de données.

Syntaxe Générale :

```
template<typename T>
class Classe {
    // Définition de la classe avec le type générique T
};
```

—

11.2 2. Exemple Simple de Patron de Classe

Voici un exemple de classe générique `Boite` qui peut contenir n'importe quel type de données.

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

template<typename T>
class Boite {
private:
    T valeur;

public:
    Boite(T v) : valeur(v) {}

    T obtenirValeur() {
        return valeur;
    }
};

int main() {
    Boite<int> boiteEntiere(42);        // Contient un entier
    Boite<string> boiteTexte("Bonjour"); // Contient une chaîne

    cout << "Entier : " << boiteEntiere.obtenirValeur() << endl;
    cout << "Texte : " << boiteTexte.obtenirValeur() << endl;
    return 0;
}
```

—

11.3 3. Patrons avec plusieurs Types

Les patrons peuvent accepter plusieurs paramètres de type.

Exemple :

```cpp
template<typename T, typename U>
class Paire {
private:
    T premier;
    U second;

public:
    Paire(T a, U b) : premier(a), second(b) {}

    void afficher() {
        cout << "Paire : (" << premier << ", " << second << ")" << endl;
    }
};

int main() {
    Paire<int, string> paire(1, "Un");
    paire.afficher(); // Affiche : Paire : (1, Un)
}
```

—

11.4 4. Fonctions Membres Hors de la Classe

Les définitions de fonctions membres pour les classes génériques doivent aussi utiliser le mot-clé `template`.

Exemple :

```cpp
template<typename T>
class Boite {
```

```
private:
    T valeur;

public:
    Boite(T v);
    T obtenirValeur();
};

template<typename T>
Boite<T>::Boite(T v) : valeur(v) {}

template<typename T>
T Boite<T>::obtenirValeur() {
    return valeur;
}
```

—

11.5 5. Spécialisation des Patrons de Classes

La spécialisation permet de définir un comportement spécifique pour un type particulier.

Exemple :

```
template<typename T>
class Afficheur {
public:
    void afficher(T valeur) {
        cout << "Valeur générique : " << valeur << endl;
    }
};

// Spécialisation pour string
template<>
class Afficheur<string> {
public:
    void afficher(string valeur) {
        cout << "Chaîne : " << valeur << endl;
    }
};

int main() {
    Afficheur<int> afficheurInt;
    Afficheur<string> afficheurString;

    afficheurInt.afficher(42);           // Affiche : Valeur générique : 42
    afficheurString.afficher("Bonjour"); // Affiche : Chaîne : Bonjour
}
```

—

11.6 6. Patrons avec des Méthodes Statistiques

Les patrons peuvent aussi contenir des membres statiques. Ces membres sont partagés entre toutes les instances d'un type donné.

Exemple :

```
template<typename T>
class Compteur {
private:
    static int total;

public:
    Compteur() {
        total++;
```

```
    }

    static int obtenirTotal() {
        return total;
    }
};

template<typename T>
int Compteur<T>::total = 0;

int main() {
    Compteur<int> c1, c2;
    Compteur<string> c3;

    cout << "Total pour int : " << Compteur<int>::obtenirTotal() << endl; // 2
    cout << "Total pour string : " << Compteur<string>::obtenirTotal() << endl;
        // 1
}
```

—

11.7 7. Limitations et Avantages des Patrons

Avantages :

- Réduction de la duplication de code.
- Flexibilité pour différents types.
- Compatible avec les types primitifs et personnalisés.

Limitations :

- La taille du binaire peut augmenter en raison de la génération de code pour chaque type utilisé.
- La gestion des erreurs de compilation peut être complexe.