

Résumé PRG1

Nicolas Reymond

January 2, 2025

Contents

Contents	2
1 Résumé : 01_Introduction	1
1.1 Organisation du cours	1
1.2 Objectifs du cours	1
1.3 Un premier programme en C++	1
1.4 Étapes de la compilation	1
1.5 Erreurs fréquentes	2
2 Résumé: 02_Bases et Opérateurs	2
2.1 Identificateurs et Mots-clés	2
2.2 Notion de Types	2
2.3 Variables et Constantes	2
2.4 Opérateurs et Expressions	3
2.5 Priorité des Opérateurs	3
2.6 Cast (Conversions de Type)	3
2.7 Exemples d'Utilisation	3
3 Résumé : 03_Instructions de Contrôle	4
3.1 Structure Séquentielle	4
3.2 Structures Conditionnelles	4
3.2.1 Instruction <code>if-else</code>	4
3.2.2 Instruction <code>switch</code>	4
3.3 Structures Itératives	5
3.3.1 Boucle <code>while</code>	5
3.3.2 Boucle <code>do-while</code>	5
3.3.3 Boucle <code>for</code>	6
3.4 Instructions de Saut	6
3.5 Résumé visuel des instructions de contrôle	7
4 Résumé : 04_Fonctions	7
4.1 Rôle des Fonctions	7
4.2 Déclaration et Définition	7
4.3 Arguments et Paramètres	8
4.4 Transmission par Valeur ou Référence	8
4.5 Valeurs de Retour	8
4.6 Arguments par Défaut	9
4.7 Fonction Récursive	9
4.8 Résumé des Concepts	9
5 Résumé : 05_Type String	9
5.1 Introduction au Type <code>string</code>	9
5.2 Déclaration et Initialisation	10
5.3 Méthodes Utiles pour les Chaînes	10
5.4 Concaténation et Comparaison	11
5.5 Lecture et Écriture	11
5.6 Conversion de Numériques en Chaînes et Inversement	11
5.7 Exemples Pratiques	11
5.8 Résumé visuel des fonctionnalités	12
6 Résumé : 06_Les Flux	12
6.1 Introduction aux Flux	12
6.2 Les Flux Prédéfinis	12
6.3 Manipulateurs de Flux	12
6.3.1 Manipulateurs pour les Entiers	13
6.3.2 Manipulateurs pour les Nombres Réels	13

6.4	Lecture au Clavier (<code>cin</code>)	13
6.5	Flux Fichiers	13
6.5.1	Écriture dans un Fichier	14
6.5.2	Lecture depuis un Fichier	14
6.6	Manipulation Avancée des Fichiers	14
6.7	Résumé Visuel des Flux	14
7	Résumé: 07_Types Numériques	14
7.1	Types Numériques en C++	14
7.2	Tableau des Types Numériques	15
7.3	Spécificateurs de Type	15
7.4	Casting (Conversion de Type)	15
7.5	Opérations sur les Types Numériques	16
7.6	Limites des Types Numériques	16
7.7	Exemples Pratiques	16
8	Résumé : 08_Type Vector	16
8.1	Introduction au <code>std::vector</code>	16
8.2	Création et Initialisation	17
8.3	Accès aux Éléments	17
8.4	Opérations de Base	17
8.5	Parcours d'un <code>vector</code>	18
8.6	Fonctions Avancées	18
8.7	Comparaison avec les Tableaux	19
8.8	Exemples Pratiques	19
8.9	Résumé Visuel des Méthodes	19
9	Résumé : 09_Surcharge et Fonctions Génériques	19
9.1	Qu'est-ce que la Surcharge de Fonctions ?	19
9.2	Règles pour la Surcharge de Fonctions	20
9.3	Surcharge d'Opérateurs	20
9.4	Surcharge de l'opérateur de flux	21
9.4.1	Pourquoi surcharger <code><<</code> ?	21
9.4.2	Syntaxe de la surcharge	21
9.4.3	Exemple complet	22
9.4.4	Explication détaillée	22
9.4.5	Autres opérateurs	22
9.4.6	Résumé	23
9.5	Fonctions Génériques avec <code>templates</code>	23
9.6	Surcharge de Fonctions Génériques	23
9.7	Templates de Classe	24
9.8	Exemples Pratiques	24
10	Résumé: Chapitre 10 - Classes et Objets	25
10.1	Notion de classe	25
10.2	Structures	25
10.3	Affectation d'objets	26
10.4	Constructeur et destructeur	26
10.5	Objets transmis en argument d'une fonction	27
10.6	Membres données statiques	27
10.7	Exploitation d'une classe	27
10.8	Surdéfinition de fonctions membres	27
10.9	Arguments par défaut	27
10.10	Fonctions membres en ligne	28
10.11	Constructeurs délégués	28
10.12	Objets transmis à une fonction membre	29
10.13	Mode de transmission des objets	29
10.14	Fonction renvoyant un objet	29
10.15	Autoréférence : <code>this</code>	29

10.16	Fonctions membres statiques	29
10.17	Fonctions membres et objets constants	30
10.18	Objets automatiques et statiques	30
11	Résumé: 11 Patrons de Classes	30
11.1	Introduction aux Patrons de Classes	30
11.2	Exemple Simple de Patron de Classe	31
11.3	Patrons avec plusieurs Types	32
11.4	Fonctions Membres Hors de la Classe	35
11.5	Spécialisation des Patrons de Classes	36
11.6	Patrons avec des Méthodes Statistiques	38
11.7	Limitations et Avantages des Patrons	38
12	Résumé : Bibliothèque <algorithm> et Itérateurs	39
12.1	Introduction aux Itérateurs	39
12.2	Prédicats	39
12.3	Algorithmes de Manipulation de Séquences	40
12.4	Transformation des Données	41
12.5	Algorithmes Statistiques	41
12.6	Expressions Lambda	42
12.7	Algorithmes de Recherche	42
12.8	Algorithmes de Suppression et Transformation	43
12.9	Mélanges et Réorganisation	44
12.10	Expressions Lambda	44
12.11	Concepts Clés	45
13	Tri : Algorithmes et Utilisation	45
13.1	Introduction au Tri	45
13.2	Tri à Bulles (Bubble Sort)	45
13.3	Tri par Insertion (Insertion Sort)	45
13.4	Utilisation de <code>std::sort</code>	46
13.5	Tri par Fusion (Merge Sort)	46
13.6	Comparaison des Algorithmes de Tri	47

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++ :

Exemple de Premier Programme

```
#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 :

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 :

Exemple de Types

```
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 :

Exemple de Variables et Constantes

```
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 :

Exemple d'Opérateurs et Expressions

```
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 :

Exemple de Priorité des Opérateurs

```
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** :

Exemple de Cast

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

2.7 Exemples d'Utilisation

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

// Boucle for basée sur une plage
vector<int> nombres = {1, 2, 3, 4, 5};
for (int n : nombres) {
    cout << n << " ";
} // <-- Add this closing brace
```

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 :

Exemple de Structure Séquentielle

```
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 :

Syntaxe de l'Instruction if-else

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

Exemple :

Exemple de l'Instruction if-else

```
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 :

Syntaxe de l'Instruction switch

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

Exemple :

Exemple de l'Instruction switch

```
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 :

Syntaxe de la Boucle while

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

Exemple :

Exemple de la Boucle while

```
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 :

Syntaxe de la Boucle do-while

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

Exemple :

Exemple de la Boucle do-while

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

3.3.3 Boucle for

- Boucle compacte avec initialisation, condition et incrémentation.

Syntaxe :

Syntaxe de la Boucle for

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

Exemple :

Exemple de la Boucle for

```
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** :

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** :

Exemple d'Utilisation de continue

```
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 :

Exemple de Fonction

```
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 :

Exemple de Déclaration et Définition

```
// 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 :

Exemple d'Arguments et Paramètres

```
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 :

Exemple de Transmission par Valeur ou Référence

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

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

Différence à l'exécution :

Exemple de Différence à l'Exécution

```
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 :

Exemple de Valeurs de Retour

```
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 :

Exemple d'Arguments par Défaut

```
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

Exemple de Fonction Récursive

```
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 :

Exemple de Type string

```
#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 :

Exemple de Déclaration et Initialisation

```
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 :

Exemple de Méthodes Utiles

```
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 :

Exemple de Concaténation et Comparaison

```
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 :

Exemple de Lecture et Écriture

```
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 :

Exemple de Conversion

```
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 :

Exemple de 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 :

Exemple de Flux

```
#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 Manipulateurs pour les Entiers

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

Exemple :

Exemple de Manipulateurs pour les Entiers

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

6.3.2 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 :

Exemple de Manipulateurs pour les Nombres Réels

```
#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 :

Exemple de Lecture au Clavier

```
#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 Écriture dans un Fichier

Exemple d'Écriture dans un Fichier

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

6.5.2 Lecture depuis un Fichier

Exemple de 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 :

Exemple de 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 :

Exemples de Types Numériques

```
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 :

Exemples de Spécificateurs de Type

```
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 :

Exemples de Casting

```
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 :

Exemples d'Opérations sur les Types Numériques

```
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 :

Exemple de Limites des Types Numériques

```
#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 :

Exemple de 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 :

Exemple de 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;
```

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 :

Exemple de `std::vector`

```
#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 :

Exemple de Création et Initialisation

```
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 :

Exemple d'Accès aux Éléments

```
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 :

Exemple d'Opérations de Base

```
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 :

Exemple de Parcours d'un vector

```
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 :

Exemple de Fonctions Avancées

```
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 :

Exemple de 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 :

Exemple de Recherche d'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 :

Exemple de Surcharge de Fonctions

```
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 :

Exemple Incorrect de Surcharge

```
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 :

Exemple de Surcharge d'Opérateurs

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

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

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

    // Surcharge de l'opérateur +
    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;
    }
};

int main() {
    Point p1(2, 3), p2(4, 5);
    Point p2 += p1;      // Utilise operator+=
    Point p3 = p1 + p2;  // Utilise operator+
    p2.afficher();       // Affiche (6, 8)
}
```

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 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) :

Exemple sans Surcharge de Flux

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

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

Exemple avec Surcharge de Flux

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

9.4.2 Syntaxe de la surcharge

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

Syntaxe de la Surcharge de Flux

```
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 Exemple complet

Exemple Complet de Surcharge de Flux

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

// Classe Point
class Point {
private:
    int x, y;

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

    // Surcharge de l'opérateur <<
    friend ostream& operator<<(ostream &out, const Point &p) {
        out << "(" << p.x << ", " << p.y << ")";
        return out; // Permet le chaînage
    }
};

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

9.4.4 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 :

Exemple de Chaînage de Flux

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

9.4.5 Autres opérateurs

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

Exemple :

Exemple de Surcharge de l'Opérateur `>>`

```
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 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 :

Syntaxe des Templates

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

Exemple :

Exemple de Template

```
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 :

Exemple de Surcharge de Template

```
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 :

Exemple de Template de Classe

```
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 :

Exemple de 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: Chapitre 10 - Classes et Objets

10.1 Notion de classe

Une classe regroupe des attributs (**données**) et des méthodes (**fonctions**) pour définir le comportement des objets. Une classe peut représenter une entité du monde réel ou abstraite.

Exemple 1 : Classe simple Exemple 2 : Classe avec constructeur —

Exemple d'Attributs et Méthodes Statistiques

```
class Rectangle {
private:
    double largeur, hauteur;

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

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

Rectangle r;
r.setDimensions(5.0, 3.0);
std::cout << "Aire : " << r.calculerAire(); // Affiche : Aire : 15
```

10.2 Structures

Une structure (**struct**) est utilisée pour regrouper des données simples. Contrairement à une classe, ses membres sont publics par défaut.

Exemple 1 : Structure simple Exemple 2 : Structure avec méthode —

Exemple d'Attributs et Méthodes Statistiques

```
class Rectangle {
private:
    double largeur, hauteur;

public:
    Rectangle(double l, double h) : largeur(l), hauteur(h) {}

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

Rectangle r(5.0, 3.0);
std::cout << "Aire : " << r.calculerAire(); // Affiche : Aire : 15
```

Exemple d'Attributs et Méthodes Statistiques

```
struct Point {
    int x, y;
};

Point p = {2, 3};
std::cout << "Coordonnées : " << p.x << ", " << p.y; // Affiche : 2, 3
```

10.3 Affectation d'objets

L'affectation d'un objet copie ses valeurs dans un autre objet. Cela peut être effectué avec le constructeur de copie ou l'opérateur d'affectation.

Exemple 1 : Copie par défaut Exemple 2 : Copie avec constructeur de copie personnalisé —

10.4 Constructeur et destructeur

Le constructeur initialise les attributs de l'objet, tandis que le destructeur libère les ressources lorsque l'objet est détruit.

Exemple 1 : Constructeur par défaut et destructeur Exemple 2 : Constructeur avec paramètres —

Exemple d'Attributs et Méthodes Statistiques

```
struct Point {
    int x, y;

    void afficher() {
        std::cout << "Point : " << x << ", " << y << "\n";
    }
};

Point p = {4, 5};
p.afficher(); // Affiche : Point : 4, 5
```

Exemple d'Attributs et Méthodes Statistiques

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

Point p1 = {2, 3};
Point p2 = p1; // Copie les valeurs de p1 dans p2
std::cout << "p2 : " << p2.x << ", " << p2.y; // Affiche : 2, 3
```

Exemple d'Attributs et Méthodes Statistiques

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

    // Constructeur de recopie
    Point(const Point& autre) {
        x = autre.x;
        y = autre.y;
        std::cout << "Copie effectuée\n";
    }
};

Point p1 = {4, 5};
Point p2 = p1; // Affiche : Copie effectuée
```

10.5 Objets transmis en argument d'une fonction

Les objets peuvent être passés à une fonction de différentes manières : par valeur, par référence, ou par référence constante.

Exemple 1 : Transmission par valeur Exemple 2 : Transmission par référence constante (efficace) —

10.6 Membres données statiques

Les membres statiques appartiennent à la classe et sont partagés entre toutes ses instances.

Exemple 1 : Attribut statique pour compter les objets Exemple 2 : Utilisation d'un membre statique partagé —

10.7 Exploitation d'une classe

Utiliser une classe consiste à appeler ses méthodes et manipuler ses attributs pour effectuer des tâches.

Exemple 1 : Calcul simple avec une classe —

10.8 Surdéfinition de fonctions membres

La surdéfinition permet de définir plusieurs versions d'une méthode dans une classe en utilisant des paramètres différents.

Exemple 1 : Surcharge de méthodes Exemple 2 : Surcharge avec des types différents —

10.9 Arguments par défaut

Les arguments par défaut permettent de simplifier les appels de fonctions en fournissant une valeur par défaut à certains paramètres.

Exemple 1 : Fonction avec un argument par défaut Exemple 2 : Plusieurs arguments

Exemple d'Attributs et Méthodes Statistiques

```
class Exemple {
public:
    Exemple() {
        std::cout << "Constructeur appelé\n";
    }
    ~Exemple() {
        std::cout << "Destructeur appelé\n";
    }
};

int main() {
    Exemple e; // Constructeur appelé
} // Destructeur appelé
```

Exemple d'Attributs et Méthodes Statistiques

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

    Point(int a, int b) : x(a), y(b) {
        std::cout << "Point initialisé : " << x << ", " << y << "\n";
    }
};

Point p(2, 3); // Affiche : Point initialisé : 2, 3
```

par défaut —

10.10 Fonctions membres en ligne

Les fonctions membres en ligne (`inline`) permettent d'éviter le surcoût des appels en insérant directement leur code.

Exemple 1 : Fonction inline dans une classe Exemple 2 : Fonction inline définie hors de la classe —

10.11 Constructeurs délégués

Les constructeurs délégués permettent d'appeler un constructeur depuis un autre pour éviter la duplication de code.

Exemple 1 : Constructeurs délégués Exemple 2 : Délégation pour des initialisations complexes —

Exemple d'Attributs et Méthodes Statistiques

```
void afficherPoint(Point p) {
    std::cout << "Coordonnées : " << p.x << ", " << p.y << "\n";
}

Point p = {2, 3};
afficherPoint(p); // Copie l'objet
```


Exemple d'Attributs et Méthodes Statistiques

```
void afficherPoint(const Point& p) {  
    std::cout << "Coordonnées : " << p.x << ", " << p.y << "\n";  
}  
  
Point p = {4, 5};  
afficherPoint(p); // Ne copie pas l'objet
```

Exemple d'Attributs et Méthodes Statistiques

```
class Compteur {  
private:  
    static int total;  
  
public:  
    Compteur() { total++; }  
    static int getTotal() { return total; }  
};  
  
int Compteur::total = 0; // Initialisation  
  
int main() {  
    Compteur c1, c2, c3;  
    std::cout << "Total : " << Compteur::getTotal(); // Affiche : Total : 3  
}
```

10.12 Objets transmis à une fonction membre

Lorsqu'une fonction membre utilise un autre objet, elle peut le manipuler par valeur, par référence, ou par référence constante.

Exemple 1 : Transmission par référence Exemple 2 : Transmission par valeur —

10.13 Mode de transmission des objets

Les objets peuvent être transmis de différentes manières. Passer par référence est souvent plus efficace pour éviter les copies inutiles.

Exemple : Transmission par référence constante —

10.14 Fonction renvoyant un objet

Une fonction membre peut retourner un nouvel objet ou une copie d'un objet existant.

Exemple : —

10.15 Autoréférence : this

Le mot-clé `this` est un pointeur implicite vers l'objet courant. Il est utile pour retourner l'objet lui-même ou différencier les membres des paramètres.

Exemple : —

10.16 Fonctions membres statiques

Les fonctions statiques n'opèrent que sur des données statiques de la classe et ne nécessitent pas d'instance pour être appelées.

Exemple : —

Exemple d'Attributs et Méthodes Statistiques

```
class Config {
public:
    static std::string version;

    static void afficherVersion() {
        std::cout << "Version : " << version << "\n";
    }
};

std::string Config::version = "1.0.0"; // Initialisation

int main() {
    Config::afficherVersion(); // Affiche : Version : 1.0.0
}
```

Exemple d'Attributs et Méthodes Statistiques

```
class Rectangle {
private:
    double largeur, hauteur;

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

    double calculerPerimetre() {
        return 2 * (largeur + hauteur);
    }
};

Rectangle rect;
rect.setDimensions(4, 5);
std::cout << "Périmètre : " << rect.calculerPerimetre(); // Affiche : Périmètre : 18
```

10.17 Fonctions membres et objets constants

Une fonction constante (`const`) garantit qu'elle ne modifie pas l'objet courant.

Exemple : —

10.18 Objets automatiques et statiques

Les objets automatiques sont détruits en fin de portée, tandis que les objets statiques persistent pendant toute l'exécution du programme.

Exemple :

11 Résumé: 11 Patrons de Classes

11.1 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 :

Exemple d'Attributs et Méthodes Statistiques

```
class Point {
public:
    void setCoord(int a) { x = a; }
    void setCoord(int a, int b) { x = a; y = b; }

private:
    int x, y;
};

Point p;
p.setCoord(5);          // Appelle la méthode avec un paramètre
p.setCoord(2, 3);       // Appelle la méthode avec deux paramètres
```

Exemple d'Attributs et Méthodes Statistiques

```
class Exemple {
public:
    void afficher(int n) { std::cout << "Entier : " << n << "\n"; }
    void afficher(double d) { std::cout << "Double : " << d << "\n"; }
};

Exemple e;
e.afficher(5);           // Affiche : Entier : 5
e.afficher(3.14);        // Affiche : Double : 3.14
```

Syntaxe Générale des Templates

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

11.2 Exemple Simple de Patron de Classe

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

Exemple d'Attributs et Méthodes Statistiques

```
void afficher(int x, int y = 0) {
    std::cout << x << ", " << y << "\n";
}

afficher(5);           // Affiche : 5, 0
afficher(5, 7);        // Affiche : 5, 7
```

Exemple d'Attributs et Méthodes Statistiques

```
void afficherRectangle(int largeur = 1, int hauteur = 1) {
    std::cout << "Rectangle : " << largeur << " x " << hauteur << "\n";
}

afficherRectangle();           // Affiche : Rectangle : 1 x 1
afficherRectangle(5);         // Affiche : Rectangle : 5 x 1
afficherRectangle(4, 6);      // Affiche : Rectangle : 4 x 6
```

Exemple d'Attributs et Méthodes Statistiques

```
class Point {
public:
    inline int getX() const { return x; }
    inline int getY() const { return y; }

private:
    int x = 5, y = 10;
};

Point p;
std::cout << p.getX() << ", " << p.getY(); // Affiche : 5, 10
```

Exemple de Patron de Classe

```
#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 Patrons avec plusieurs Types

Les patrons peuvent accepter plusieurs paramètres de type.

Exemple d'Attributs et Méthodes Statistiques

```
class Rectangle {
private:
    double largeur, hauteur;

public:
    Rectangle(double l, double h) : largeur(l), hauteur(h) {}

    inline double calculerAire() const;
};

// Définition inline hors de la classe
inline double Rectangle::calculerAire() const {
    return largeur * hauteur;
}

Rectangle rect(5, 3);
std::cout << "Aire : " << rect.calculerAire(); // Affiche : Aire : 15
```

Exemple d'Attributs et Méthodes Statistiques

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

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

    // Constructeur délégué
    Point() : Point(0, 0) {}
};

Point p1;          // Appelle le constructeur délégué (x = 0, y = 0)
Point p2(3, 4);    // Appelle le constructeur principal (x = 3, y = 4)
```

Exemple :

Exemple d'Attributs et Méthodes Statistiques

```
class Rectangle {
private:
    double largeur, hauteur;

public:
    Rectangle(double l, double h) : largeur(l), hauteur(h) {}

    Rectangle() : Rectangle(1.0, 1.0) {} // Initialise à un rectangle par défaut
};

Rectangle r1;           // Rectangle par défaut : 1 x 1
Rectangle r2(4.0, 5.0); // Rectangle : 4 x 5
```

Exemple d'Attributs et Méthodes Statistiques

```
class Rectangle {
public:
    double largeur, hauteur;

    void copier(const Rectangle& autre) {
        largeur = autre.largeur;
        hauteur = autre.hauteur;
    }
};

Rectangle r1{5, 3}, r2;
r2.copier(r1); // Copie les dimensions de r1 dans r2
std::cout << r2.largeur << " x " << r2.hauteur; // Affiche : 5 x 3
```

Exemple de Patron avec Plusieurs Types

```
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)
}
```

Exemple d'Attributs et Méthodes Statistiques

```
class Rectangle {
public:
    double largeur, hauteur;

    Rectangle doubler(Rectangle autre) {
        autre.largeur *= 2;
        autre.hauteur *= 2;
        return autre;
    }
};

Rectangle r1{4, 5};
Rectangle r2 = r1.doubler(r1); // r1 n'est pas modifié
std::cout << r2.largeur << " x " << r2.hauteur; // Affiche : 8 x 10
```

Exemple d'Attributs et Méthodes Statistiques

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

    void afficher(const Point& autre) const {
        std::cout << autre.x << ", " << autre.y;
    }
};

Point p1{3, 7}, p2{4, 8};
p1.afficher(p2); // Affiche : 4, 8
```

11.4 Fonctions Membres Hors de la Classe

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

Exemple :

Exemple de Fonctions Membres Hors de la Classe

```
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;
}
```

Exemple d'Attributs et Méthodes Statistiques

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

    Point ajouter(const Point& autre) const {
        return Point{x + autre.x, y + autre.y};
    }
};

Point p1{2, 3}, p2{4, 5};
Point p3 = p1.ajouter(p2); // Retourne un nouveau point (6, 8)
std::cout << p3.x << ", " << p3.y; // Affiche : 6, 8
```

Exemple d'Attributs et Méthodes Statistiques

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

    Point& setX(int val) {
        x = val;
        return *this; // Retourne l'objet courant
    }

    Point& setY(int val) {
        y = val;
        return *this;
    }
};

Point p;
p.setX(5).setY(10); // Chaîne les appels
std::cout << p.x << ", " << p.y; // Affiche : 5, 10
```

11.5 Spécialisation des Patrons de Classes

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

Exemple :

Exemple d'Attributs et Méthodes Statistiques

```
class Compteur {
private:
    static int total;

public:
    static void incrementer() { total++; }
    static int getTotal() { return total; }
};

int Compteur::total = 0;

Compteur::incrementer();
std::cout << "Total : " << Compteur::getTotal(); // Affiche : Total : 1
```

Exemple d'Attributs et Méthodes Statistiques

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

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

    int getX() const { return x; } // Méthode constante
    int getY() const { return y; }
};

const Point p{2, 3};
std::cout << p.getX(); // Valide, car getX est const
```

Exemple de Spécialisation de Patron

```
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
}
```

Exemple d'Attributs et Méthodes Statistiques

```
void exemple() {
    static int compteur = 0; // Persiste entre les appels
    compteur++;
    std::cout << "Compteur : " << compteur << "\n";
}

int main() {
    exemple(); // Compteur : 1
    exemple(); // Compteur : 2
}
```

11.6 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 :

Exemple de Méthodes Statistiques

```
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 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.

12 Résumé : Bibliothèque <algorithm> et Itérateurs

12.1 Introduction aux Itérateurs

Catégories d'Itérateurs :

- **In_it** : Pour la lecture (*it).
- **Out_it** : Pour l'écriture (*it = value).
- **Unidir_it** : Unidirectionnels (++it).
- **Bidir_it** : Bidirectionnels (++it, --it).
- **Direct_it** : Aléatoires (it[n]).

Exemples d'Itérateurs :

Exemples d'Itérateurs

```
// Exemple d'itérateur unidirectionnel
std::vector<int> vec = {1, 2, 3, 4, 5};
for (auto it = vec.begin(); it != vec.end(); ++it) {
    std::cout << *it << " ";
}

// Exemple d'itérateur inversé
for (auto it = vec.rbegin(); it != vec.rend(); ++it) {
    std::cout << *it << " ";
}
```

Fonctions associées :

- **begin(), end()** : Renvoient les itérateurs pour parcourir une séquence.
- **rbegin(), rend()** : Itérateurs pour un parcours inversé.
- **advance(it, n)** : Déplace un itérateur de **n** pas.

Exemple : Avancer un Itérateur :

Exemple d'Avancer un Itérateur

```
auto it = vec.begin();
std::advance(it, 2); // Avance l'itérateur de 2 positions
std::cout << *it;    // Affiche 3
```

12.2 Prédicats

Un **prédicat** est une fonction qui retourne un booléen (**true** ou **false**). Les prédicats permettent de filtrer ou transformer des données.

Types de Prédicats :

- **Prédicat unaire** : Prend un seul argument.
- **Prédicat binaire** : Prend deux arguments.

Exemples de Prédicats :

Exemples de Prédicats

```
// Prédicat unaire
bool estPair(int x) {
    return x % 2 == 0;
}

// Prédicat binaire
bool comparerDescendant(int a, int b) {
    return a > b;
}

// Utilisation dans un algorithme
std::sort(vec.begin(), vec.end(), comparerDescendant); // Tri en ordre décroissant
```

12.3 Algorithmes de Manipulation de Séquences

Remplir une Séquence :

- `fill(first, last, val)` : Remplit une plage avec une valeur donnée.
- `fill_n(first, n, val)` : Remplit un nombre donné d'éléments.

Exemples :

Exemples de Remplissage

```
std::vector<int> vec(5);
std::fill(vec.begin(), vec.end(), 42); // Tous les éléments valent 42
std::fill_n(vec.begin(), 3, 10);      // Les 3 premiers éléments valent 10
```

Copier une Séquence :

- `copy(first, last, result)` : Copie une plage.
- `copy_if(first, last, result, pred)` : Copie les éléments satisfaisant un prédicat.
- `copy_backward(first, last, result)` : Copie dans l'ordre inverse.

Exemples :

Exemples de Copie

```
std::vector<int> source = {1, 2, 3, 4, 5};
std::vector<int> destination(5);

// Copier tous les éléments
std::copy(source.begin(), source.end(), destination.begin());

// Copier uniquement les éléments pairs
std::vector<int> pairs;
std::copy_if(source.begin(), source.end(), std::back_inserter(pairs), [](int x) { return x % 2 == 0; });
```

12.4 Transformation des Données

Opérations de Transformation :

- `transform(first, last, result, op)` : Applique une opération à chaque élément.
- `replace(first, last, old_val, new_val)` : Remplace les éléments égaux à `old_val`.
- `replace_if(first, last, pred, new_val)` : Remplace les éléments satisfaisant un prédicat.

Exemples :

Exemples de Transformation

```
std::vector<int> vec = {1, 2, 3, 4, 5};

// Transformer chaque élément en son double
std::transform(vec.begin(), vec.end(), vec.begin(), [](int x) { return x * 2; });

// Remplacer 4 par 99
std::replace(vec.begin(), vec.end(), 4, 99);

// Remplacer les éléments pairs par 0
std::replace_if(vec.begin(), vec.end(), [](int x) { return x % 2 == 0; }, 0);
```

12.5 Algorithmes Statistiques

Maximum et Minimum :

- `max_element(first, last)` : Trouve le plus grand élément.
- `min_element(first, last)` : Trouve le plus petit élément.

Exemples :

Exemples de Maximum et Minimum

```
std::vector<int> vec = {5, 1, 8, 3, 2};
auto max_it = std::max_element(vec.begin(), vec.end()); // max_it pointe sur 8
auto min_it = std::min_element(vec.begin(), vec.end()); // min_it pointe sur 1
```

Tests de Prédicats :

- `all_of(first, last, pred)` : Vérifie si **tous** les éléments satisfont un prédicat.
- `any_of(first, last, pred)` : Vérifie si **au moins un** élément satisfait un prédicat.
- `none_of(first, last, pred)` : Vérifie si **aucun** élément ne satisfait un prédicat.

Exemples :

Exemples de Tests de Prédicats

```
std::vector<int> vec = {2, 4, 6, 8};

// Vérifie si tous les éléments sont pairs
bool tousPairs = std::all_of(vec.begin(), vec.end(), [](int x) { return x % 2 == 0; });

// Vérifie s'il existe un élément supérieur à 5
bool auMoinsUnSup5 = std::any_of(vec.begin(), vec.end(), [](int x) { return x > 5; });

// Vérifie si aucun élément n'est impair
bool aucunImpair = std::none_of(vec.begin(), vec.end(), [](int x) { return x % 2 != 0; });
```

12.6 Expressions Lambda

Syntaxe Générale :

Syntaxe Générale des Lambdas

```
[(paramètres) -> type_retour { corps }
```

Exemples de Lambdas :

Exemples de Lambdas

```
// Lambda pour vérifier si un nombre est pair
auto estPair = [](int x) { return x % 2 == 0; };

// Lambda capturant une variable
int seuil = 5;
auto plusGrandQueSeuil = [seuil](int x) { return x > seuil; };

// Appliquer une lambda dans un algorithme
std::vector<int> vec = {1, 2, 3, 4, 5, 6};
std::copy_if(vec.begin(), vec.end(), std::back_inserter(result), estPair);
```

12.7 Algorithmes de Recherche

Trouver un Élément :

- `find(first, last, val)` : Cherche un élément donné.
- `find_if(first, last, pred)` : Cherche le premier élément satisfaisant un prédicat.
- `find_if_not(first, last, pred)` : Cherche le premier élément qui ne satisfait pas un prédicat.

Exemples :

Exemples de Recherche d'Élément

```
std::vector<int> vec = {1, 3, 5, 7, 9};

// Trouve la première occurrence de 5
auto it = std::find(vec.begin(), vec.end(), 5);
if (it != vec.end()) {
    std::cout << "5 trouvé à l'indice : " << std::distance(vec.begin(), it);
}

// Trouve le premier élément pair
auto pair_it = std::find_if(vec.begin(), vec.end(), [](int x) { return x % 2 == 0; });
if (pair_it != vec.end()) {
    std::cout << "Premier élément pair : " << *pair_it;
}
```

Autres Recherches :

- `search(first1, last1, first2, last2)` : Trouve une sous-séquence dans une plage.
- `search_n(first, last, n, val)` : Trouve une sous-séquence de `n` éléments consécutifs égaux.

Exemples :

Exemples de Recherche Avancée

```
// Recherche d'une sous-séquence {3, 5}
std::vector<int> sub = {3, 5};
auto sub_it = std::search(vec.begin(), vec.end(), sub.begin(), sub.end());
if (sub_it != vec.end()) {
    std::cout << "Sous-séquence trouvée à l'indice : " << std::distance(vec.begin(), sub_it);
}

// Recherche de 3 occurrences consécutives de 7
auto n_it = std::search_n(vec.begin(), vec.end(), 3, 7);
if (n_it != vec.end()) {
    std::cout << "Trois occurrences consécutives trouvées à l'indice : " << std::distance(vec.begin(), n_it);
}
```

12.8 Algorithmes de Suppression et Transformation

Supprimer des Éléments :

- `remove(first, last, val)` : Supprime les occurrences d'une valeur.
- `remove_if(first, last, pred)` : Supprime les éléments satisfaisant un prédicat.

Exemples :

Exemples de Suppression

```
std::vector<int> vec = {1, 2, 3, 4, 5, 6};

// Supprimer tous les éléments égaux à 3
vec.erase(std::remove(vec.begin(), vec.end(), 3), vec.end());

// Supprimer tous les éléments pairs
vec.erase(std::remove_if(vec.begin(), vec.end(), [](int x) { return x % 2 == 0; }), vec.end());
```

Transformer une Séquence :

- `transform(first, last, result, op)` : Applique une opération à chaque élément.
- `replace(first, last, old_val, new_val)` : Remplace les occurrences d'une valeur.
- `replace_if(first, last, pred, new_val)` : Remplace les éléments satisfaisant un prédicat.

Exemples :

Exemples de Transformation

```
// Multiplier chaque élément par 2
std::transform(vec.begin(), vec.end(), vec.begin(), [](int x) { return x * 2; });

// Remplacer 4 par 42
std::replace(vec.begin(), vec.end(), 4, 42);

// Remplacer les éléments pairs par 0
std::replace_if(vec.begin(), vec.end(), [](int x) { return x % 2 == 0; }, 0);
```

12.9 Mélanges et Réorganisation

Mélanger les Éléments :

- `shuffle(first, last, rng)` : Mélange aléatoirement les éléments (C++11 et ultérieur).
- `random_shuffle(first, last)` : Mélange aléatoire (obsolète depuis C++17).

Exemple :

Exemple de Mélange

```
std::random_device rd;
std::mt19937 rng(rd());
std::shuffle(vec.begin(), vec.end(), rng);
```

Réorganiser les Éléments :

- `reverse(first, last)` : Inverse une séquence.
- `rotate(first, middle, last)` : Fait pivoter une séquence autour d'un pivot.

Exemple :

Exemple de Réorganisation

```
// Inverser l'ordre des éléments
std::reverse(vec.begin(), vec.end());

// Faire pivoter pour que le 3 devienne le premier élément
std::rotate(vec.begin(), vec.begin() + 2, vec.end());
```

12.10 Expressions Lambda

Syntaxe Générale :

Syntaxe Générale des Lambdas

```
[](paramètres) -> type_retour { corps }
```

Exemples de Lambdas :

Exemples de Lambdas

```
// Lambda pour vérifier si un nombre est pair
auto estPair = [](int x) { return x % 2 == 0; };

// Lambda capturant une variable
int seuil = 5;
auto plusGrandQueSeuil = [seuil](int x) { return x > seuil; };

// Appliquer une lambda dans un algorithme
std::vector<int> vec = {1, 2, 3, 4, 5, 6};
std::copy_if(vec.begin(), vec.end(), std::back_inserter(result), estPair);
```


12.11 Concepts Clés

- Comprendre les différentes catégories d'itérateurs.
- Savoir utiliser les prédicats dans les algorithmes (`find_if`, `copy_if`, `remove_if`).
- Maîtriser les transformations de séquences (`transform`, `replace`).
- Utiliser les expressions lambda pour simplifier le code.
- Connaître les algorithmes de recherche (`find`, `search`, `search_n`).

13 Tri : Algorithmes et Utilisation

13.1 Introduction au Tri

Le tri consiste à organiser les éléments d'un tableau ou d'une liste dans un ordre spécifique (croissant ou décroissant). Voici quelques algorithmes de tri couramment utilisés et leurs caractéristiques.

13.2 Tri à Bulles (Bubble Sort)

Le tri à bulles compare chaque paire d'éléments adjacents et les échange s'ils sont dans le mauvais ordre. Cet algorithme est simple mais inefficace pour les grands tableaux.

Exemple :

Exemple d'Attributs et Méthodes Statistiques

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Échange
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Complexité :

- Meilleur cas : $O(n)$ (tableau déjà trié).
- Pire cas : $O(n^2)$ (tableau trié dans l'ordre inverse).
- Moyenne : $O(n^2)$.

13.3 Tri par Insertion (Insertion Sort)

Le tri par insertion insère chaque élément à sa position correcte dans un sous-tableau déjà trié.

Exemple :

Exemple d'Attributs et Méthodes Statistiques

```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        // Déplace les éléments plus grands que key
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

Complexité :

- Meilleur cas : $O(n)$.
 - Pire cas : $O(n^2)$.
 - Moyenne : $O(n^2)$.
-

13.4 Utilisation de `std::sort`

La bibliothèque standard C++ propose l'algorithme `std::sort`, qui est beaucoup plus rapide et optimisé.

Exemple :

Exemple d'Attributs et Méthodes Statistiques

```
#include <algorithm>
#include <vector>

int main() {
    std::vector<int> vec = {5, 2, 9, 1, 5, 6};
    std::sort(vec.begin(), vec.end());

    for (int v : vec) {
        std::cout << v << " ";
    }
    return 0;
}
```

Complexité :

- En moyenne : $O(n \log n)$.
-

13.5 Tri par Fusion (Merge Sort)

Le tri par fusion divise le tableau en sous-tableaux, les trie récursivement, puis les fusionne.

Exemple :

Exemple d'Attributs et Méthodes Statistiques

```
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++) L[i] = arr[l + i];
    for (int i = 0; i < n2; i++) R[i] = arr[m + 1 + i];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```

Complexité :

- Pire cas : $O(n \log n)$.

—

13.6 Comparaison des Algorithmes de Tri

Algorithme	Meilleur Cas	Pire Cas	Moyenne
Tri à Bulles	$O(n)$	$O(n^2)$	$O(n^2)$
Tri par Insertion	$O(n)$	$O(n^2)$	$O(n^2)$
<code>std::sort</code>	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Tri par Fusion	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$