

Résumé PRG1

Nicolas Reymond

December 19, 2024

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	3
3.1 Structure Séquentielle	3
3.2 Structures Conditionnelles	3
3.2.1 Instruction <code>if-else</code>	4
3.2.2 Instruction <code>switch</code>	4
3.3 Structures Itératives	4
3.3.1 Boucle <code>while</code>	4
3.3.2 Boucle <code>do-while</code>	5
3.3.3 Boucle <code>for</code>	5
3.4 Instructions de Saut	5
3.5 Résumé visuel des instructions de contrôle	6
4 Résumé : 04_Fonctions	6
4.1 Rôle des Fonctions	6
4.2 Déclaration et Définition	6
4.3 Arguments et Paramètres	6
4.4 Transmission par Valeur ou Référence	7
4.5 Valeurs de Retour	7
4.6 Arguments par Défaut	7
4.7 Fonction Récursive	8
4.8 Résumé des Concepts	8
5 Résumé : 05_Type String	8
5.1 Introduction au Type <code>string</code>	8
5.2 Déclaration et Initialisation	8
5.3 Méthodes Utiles pour les Chaînes	9
5.4 Concaténation et Comparaison	9
5.5 Lecture et Écriture	9
5.6 Conversion de Numériques en Chaînes et Inversement	9
5.7 Exemples Pratiques	10
5.8 Résumé visuel des fonctionnalités	10
6 Résumé : 06_Les Flux	10
6.1 Introduction aux Flux	10
6.2 Les Flux Prédéfinis	10
6.3 Manipulateurs de Flux	10
6.3.1 Manipulateurs pour les Entiers	10
6.3.2 Manipulateurs pour les Nombres Réels	11

6.4	Lecture au Clavier (<code>cin</code>)	11
6.5	Flux Fichiers	11
6.5.1	Écriture dans un Fichier	11
6.5.2	Lecture depuis un Fichier	12
6.6	Manipulation Avancée des Fichiers	12
6.7	Résumé Visuel des Flux	12
7	Résumé: 07_Types Numériques	12
7.1	Types Numériques en C++	12
7.2	Tableau des Types Numériques	13
7.3	Spécificateurs de Type	13
7.4	Casting (Conversion de Type)	13
7.5	Opérations sur les Types Numériques	13
7.6	Limites des Types Numériques	13
7.7	Exemples Pratiques	14
7.8	Résumé Visuel des Types Numériques	14
8	Résumé : 08_Type Vector	14
8.1	Introduction au <code>std::vector</code>	14
8.2	Création et Initialisation	14
8.3	Accès aux Éléments	14
8.4	Opérations de Base	15
8.5	Parcours d'un <code>vector</code>	15
8.6	Fonctions Avancées	16
8.7	Comparaison avec les Tableaux	16
8.8	Exemples Pratiques	16
8.9	Résumé Visuel des Méthodes	16
9	Résumé : 09_Surcharge et Fonctions Génériques	16
9.1	Qu'est-ce que la Surcharge de Fonctions ?	16
9.2	Règles pour la Surcharge de Fonctions	17
9.3	Surcharge d'Opérateurs	17
9.4	Surcharge de l'opérateur de flux	18
9.4.1	Pourquoi surcharger <code><<</code> ?	18
9.4.2	Syntaxe de la surcharge	18
9.4.3	Exemple complet	18
9.4.4	Explication détaillée	19
9.4.5	Autres opérateurs	19
9.4.6	Résumé	19
9.5	Fonctions Génériques avec <code>templates</code>	19
9.6	Surcharge de Fonctions Génériques	20
9.7	Templates de Classe	20
9.8	Exemples Pratiques	21
10	Résumé: 10_Classes et Objets	21
10.1	Qu'est-ce qu'une Classe ?	21
10.2	Qu'est-ce qu'un Objet ?	21
10.3	Visibilité des Membres	22
10.4	Constructeurs et Destructeurs	22
10.5	Encapsulation	22
10.6	Méthodes Constantes	23
10.7	Attributs et Méthodes Statistiques	23
11	Résumé: 11 Patrons de Classes	23
11.1	Introduction aux Patrons de Classes	23
11.2	Exemple Simple de Patron de Classe	24
11.3	Patrons avec plusieurs Types	24
11.4	Fonctions Membres Hors de la Classe	24
11.5	Spécialisation des Patrons de Classes	25

11.6	Patrons avec des Méthodes Statistiques	25
11.7	Limitations et Avantages des Patrons	26
12	Résumé : Bibliothèque <algorithm> et Itérateurs	26
12.1	Introduction aux Itérateurs	26
12.2	Prédicats	27
12.3	Algorithmes de Manipulation de Séquences	27
12.4	Transformation des Données	28
12.5	Algorithmes Statistiques	28
12.6	Expressions Lambda	29
12.7	Algorithmes de Recherche	29
12.8	Algorithmes de Suppression et Transformation	30
12.9	Mélanges et Réorganisation	31
12.10	Expressions Lambda	31
12.11	Concepts Clés	31
13	Tri : Algorithmes et Utilisation	32
13.1	Introduction au Tri	32
13.2	Tri à Bulles (Bubble Sort)	32
13.3	Tri par Insertion (Insertion Sort)	32
13.4	Utilisation de <code>std::sort</code>	33
13.5	Tri par Fusion (Merge Sort)	33
13.6	Comparaison des Algorithmes de Tri	34

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++ :

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     cout << "Bonjour, PRG1 !" << endl;
6     return 0;
7 }
```

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 :

```
1 int main() {  
2     cout << "Bonjour" // Manque le point-virgule  
3     return 0;  
4 }
```

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 :

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

2.3 Variables et Constantes

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

Exemple :

```
1 const double PI = 3.14; // Constante  
2 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 :

```
1 int a = 5, b = 2;
2 int somme = a + b;    // Résultat : 7
3 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 :

```
1 int resultat = 5 + 3 * 2;    // Résultat : 11 (3 * 2 évalué en premier)
2 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** :

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

2.7 Exemples d'Utilisation

```
1 // Calcul simple
2 int a = 10, b = 3;
3 cout << "Somme : " << (a + b) << endl; // Affiche 13
4 cout << "Division : " << (a / b) << endl; // Affiche 3 (division entière)
5 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 :

```
1 int a = 5;
2 cout << "Valeur de a : " << a << endl;
3 // 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 :

```
1 if (condition) {
2     // Instructions si condition est vraie
3 } else {
4     // Instructions si condition est fausse
5 }
```

Exemple :

```
1 int age = 20;
2 if (age >= 18) {
3     cout << "Vous êtes majeur." << endl;
4 } else {
5     cout << "Vous êtes mineur." << endl;
6 }
```

3.2.2 Instruction switch

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

Syntaxe :

```
1 switch (variable) {
2     case valeur1:
3         // Instructions
4         break;
5     case valeur2:
6         // Instructions
7         break;
8     default:
9         // Instructions par défaut
10 }
```

Exemple :

```
1 int jour = 3;
2 switch (jour) {
3     case 1:
4         cout << "Lundi" << endl;
5         break;
6     case 2:
7         cout << "Mardi" << endl;
8         break;
9     default:
10         cout << "Autre jour" << endl;
11 }
```

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 :


```

1 while (condition) {
2     // Instructions
3 }

```

Exemple :

```

1 int i = 0;
2 while (i < 5) {
3     cout << i << endl;
4     i++;
5 }

```

3.3.2 Boucle do-while

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

Syntaxe :

```

1 do {
2     // Instructions
3 } while (condition);

```

Exemple :

```

1 int i = 0;
2 do {
3     cout << i << endl;
4     i++;
5 } while (i < 5);

```

3.3.3 Boucle for

- Boucle compacte avec initialisation, condition et incrémentation.

Syntaxe :

```

1 for (initialisation; condition; incrément) {
2     // Instructions
3 }

```

Exemple :

```

1 for (int i = 0; i < 5; i++) {
2     cout << i << endl;
3 }

```

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 :

```

1 for (int i = 0; i < 10; i++) {
2     if (i == 5) break; // Sort de la boucle si i vaut 5
3     cout << i << endl;
4 }

```

Exemple d'utilisation de continue :

```

1 for (int i = 0; i < 5; i++) {
2     if (i == 2) continue; // Passe directement à i = 3
3     cout << i << endl;
4 }

```

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 :

```

1 double addition(double a, double b) {
2     return a + b;
3 }
4
5 int main() {
6     double resultat = addition(3.5, 2.2);
7     cout << "Résultat : " << resultat << endl;
8     return 0;
9 }

```

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 :

```

1 // Déclaration
2 double addition(double a, double b);
3
4 // Définition
5 double addition(double a, double b) {
6     return a + b;
7 }

```

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 :

```

1 void afficherMessage(string message) { // Paramètre : message
2     cout << message << endl;
3 }
4
5 int main() {
6     afficherMessage("Bonjour, PRG1 !"); // Argument : "Bonjour, PRG1 !"
7     return 0;
8 }

```

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 :

```

1 // Par valeur
2 void incrementerValeur(int n) {
3     n++;
4 }
5
6 // Par référence
7 void incrementerReference(int &n) {
8     n++;
9 }

```

Différence à l'exécution :

```

1 int main() {
2     int a = 5;
3     incrementerValeur(a); // a reste 5
4     incrementerReference(a); // a devient 6
5 }

```

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 :

```

1 int carre(int x) {
2     return x * x;
3 }
4
5 void afficherMessage() {
6     cout << "Message sans retour." << endl;
7 }

```

4.6 Arguments par Défaut

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

Exemple :

```

1 void saluer(string nom = "Utilisateur") {
2     cout << "Bonjour, " << nom << " !" << endl;
3 }
4
5 int main() {
6     saluer(); // Affiche : Bonjour, Utilisateur !

```

```

7     saluer("Alice");           // Affiche : Bonjour, Alice !
8 }

```

4.7 Fonction Récursive

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

Exemple : Calcul Factoriel

```

1 int factoriel(int n) {
2     if (n <= 1)
3         return 1;
4     else
5         return n * factoriel(n - 1);
6 }
7
8 int main() {
9     cout << "Factoriel de 5 : " << factoriel(5) << endl;
10    return 0;
11 }

```

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 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 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 Écriture dans un Fichier

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

6.5.2 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 :

```
1 int entier = 42;           // Type entier
2 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 :

```
1 unsigned int positif = 42; // Seulement des nombres positifs
2 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 :

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

7.5 Opérations sur les Types Numériques

Les opérateurs mathématiques s'appliquent directement :

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

Exemples :

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

7.6 Limites des Types Numériques

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

Exemple :

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

7.7 Exemples Pratiques

Calcul de Moyenne :

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

Vérification de Dépassement :

```
1 int a = numeric_limits<int>::max();
2 cout << "a : " << a << endl;
3 a = a + 1; // Dépassement : résultat indéfini
4 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 (<code>push_back</code> , 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 :

```

1 void afficher(int x) {
2     cout << "Entier : " << x << endl;
3 }
4
5 void afficher(double x) {
6     cout << "Flottant : " << x << endl;
7 }
8
9 void afficher(string x) {
10    cout << "Texte : " << x << endl;
11 }
12
13 int main() {
14     afficher(42);           // Appelle afficher(int)
15     afficher(3.14);        // Appelle afficher(double)
16     afficher("Bonjour");   // Appelle afficher(string)
17 }

```

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 :

```

1 int addition(int a, int b);
2 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 :

```

1 class Point {
2 public:
3     int x, y;
4
5     Point(int a, int b) : x(a), y(b) {}
6
7     // Surcharge de l'opérateur +=
8     Point operator+=(const Point &p) {
9         return Point(x + p.x, y + p.y);
10    }
11
12    // Surcharge de l'opérateur +
13    Point operator+(const Point_droit &p, const Point_gauche &q) {
14        return Point(p.x + q.x, p.y + q.y);
15    }
16
17    void afficher() {
18        cout << "(" << x << ", " << y << ")" << endl;
19    }
20 };

```

```

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 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 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 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

```

```

20 int main() {
21     Point p1(2, 3), p2(5, 8);
22     cout << p1 << endl; // Affiche : (2, 3)
23     cout << p2 << endl; // Affiche : (5, 8)
24 }

```

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 :

```

1     cout << p1 << " et " << p2;
2     // 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 :

```

1 istream& operator>>(istream &in, Point &p) {
2     in >> p.x >> p.y;
3     return in;
4 }
5
6 int main() {
7     Point p;
8     cin >> p; // Saisir deux entiers pour x et y
9     cout << p; // Affiche le point saisi
10 }

```

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 :

```

1 template<typename T>
2 T addition(T a, T b) {
3     return a + b;
4 }

```

Exemple :

```

1 template<typename T>
2 T addition(T a, T b) {
3     return a + b;
4 }
5
6 int main() {
7     cout << addition(5, 3) << endl;    // Utilise int
8     cout << addition(2.5, 1.5) << endl; // Utilise double
9     return 0;
10 }

```

9.6 Surcharge de Fonctions Génériques

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

Exemple :

```

1 template<typename T>
2 void afficher(T valeur) {
3     cout << "Valeur generique : " << valeur << endl;
4 }
5
6 // Surcharge pour string
7 void afficher(string valeur) {
8     cout << "Texte : " << valeur << endl;
9 }
10
11 int main() {
12     afficher(42);           // Appelle la fonction generique
13     afficher("Bonjour");   // Appelle la surcharge pour string
14 }

```

9.7 Templates de Classe

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

Exemple :

```

1 template<typename T>
2 class Boite {
3 private:
4     T valeur;
5 public:
6     Boite(T v) : valeur(v) {}
7     void afficher() {
8         cout << "Valeur : " << valeur << endl;
9     }
10 };
11
12 int main() {
13     Boite<int> boiteEntiere(42);
14     Boite<string> boiteTexte("Bonjour");
15
16     boiteEntiere.afficher(); // Valeur : 42
17     boiteTexte.afficher();   // Valeur : Bonjour
18 }

```


9.8 Exemples Pratiques

Combinaison de Templates et Surcharge :

```
1 template<typename T>
2 T carre(T x) {
3     return x * x;
4 }
5
6 // Specialisation pour string
7 string carre(string x) {
8     return x + x; // Repete la chaine
9 }
10
11 int main() {
12     cout << carre(5) << endl;           // 25
13     cout << carre(3.14) << endl;       // 9.8596
14     cout << carre("Hello") << endl;    // HelloHello
15 }
```

10 Résumé: 10_Classes et Objets

10.1 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 :

```
1 class Rectangle {
2 private:
3     double largeur, hauteur; // Attributs privés
4
5 public:
6     // Méthodes publiques
7     void setDimensions(double l, double h) {
8         largeur = l;
9         hauteur = h;
10    }
11
12    double calculerAire() {
13        return largeur * hauteur;
14    }
15 };
```

10.2 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 :

```
1 int main() {
2     Rectangle rect; // Création d'un objet
3     rect.setDimensions(5.0, 3.0); // Utilisation des méthodes
4     cout << "Aire : " << rect.calculerAire(); // Affiche 15.0
5     return 0;
6 }
```

10.3 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 :

```
1 class Exemple {
2 public:
3     int publicAttr; // Accessible partout
4 private:
5     int privateAttr; // Accessible uniquement dans la classe
6 protected:
7     int protectedAttr; // Accessible dans les classes dérivées
8 };
```

10.4 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 :

```
1 class Point {
2 private:
3     int x, y;
4
5 public:
6     // Constructeur
7     Point(int a, int b) : x(a), y(b) {
8         cout << "Point créé !" << endl;
9     }
10
11    // Destructeur
12    ~Point() {
13        cout << "Point détruit !" << endl;
14    }
15 };
```

10.5 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 :

```
1 class CompteBancaire {
2 private:
3     double solde;
4
5 public:
6     void deposer(double montant) {
7         solde += montant;
8     }
9 };
```

```

9
10     double obtenirSolde() {
11         return solde;
12     }
13 };

```

10.6 Méthodes Constantes

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

Exemple :

```

1 class Point {
2 private:
3     int x, y;
4
5 public:
6     int obtenirX() const { return x; } // Méthode constante
7 };

```

10.7 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 :

```

1 class Compteur {
2 private:
3     static int total; // Attribut statique
4
5 public:
6     static int obtenirTotal() { return total; } // Méthode statique
7 };
8
9 int Compteur::total = 0; // Initialisation

```

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 :

```

1 template<typename T>
2 class Classe {
3     // Définition de la classe avec le type générique T
4 };

```

11.2 Exemple Simple de Patron de Classe

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

```
1 #include <iostream>
2 using namespace std;
3
4 template<typename T>
5 class Boite {
6 private:
7     T valeur;
8
9 public:
10    Boite(T v) : valeur(v) {}
11
12    T obtenirValeur() {
13        return valeur;
14    }
15 };
16
17 int main() {
18     Boite<int> boiteEntiere(42);           // Contient un entier
19     Boite<string> boiteTexte("Bonjour"); // Contient une chaîne
20
21     cout << "Entier : " << boiteEntiere.obtenirValeur() << endl;
22     cout << "Texte : " << boiteTexte.obtenirValeur() << endl;
23     return 0;
24 }
```

11.3 Patrons avec plusieurs Types

Les patrons peuvent accepter plusieurs paramètres de type.

Exemple :

```
1 template<typename T, typename U>
2 class Paire {
3 private:
4     T premier;
5     U second;
6
7 public:
8     Paire(T a, U b) : premier(a), second(b) {}
9
10    void afficher() {
11        cout << "Paire : (" << premier << ", " << second << ")" << endl;
12    }
13 };
14
15 int main() {
16     Paire<int, string> paire(1, "Un");
17     paire.afficher(); // Affiche : Paire : (1, Un)
18 }
```

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 :

```
1 template<typename T>
2 class Boite {
```

```

3 private:
4     T valeur;
5
6 public:
7     Boite(T v);
8     T obtenirValeur();
9 };
10
11 template<typename T>
12 Boite<T>::Boite(T v) : valeur(v) {}
13
14 template<typename T>
15 T Boite<T>::obtenirValeur() {
16     return valeur;
17 }

```

11.5 Spécialisation des Patrons de Classes

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

Exemple :

```

1 template<typename T>
2 class Afficheur {
3 public:
4     void afficher(T valeur) {
5         cout << "Valeur générique : " << valeur << endl;
6     }
7 };
8
9 // Spécialisation pour string
10 template<>
11 class Afficheur<string> {
12 public:
13     void afficher(string valeur) {
14         cout << "Chaîne : " << valeur << endl;
15     }
16 };
17
18 int main() {
19     Afficheur<int> afficheurInt;
20     Afficheur<string> afficheurString;
21
22     afficheurInt.afficher(42); // Affiche : Valeur générique : 42
23     afficheurString.afficher("Bonjour"); // Affiche : Chaîne : Bonjour
24 }

```

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 :

```

1 template<typename T>
2 class Compteur {
3 private:
4     static int total;
5
6 public:
7     Compteur() {
8         total++;

```

```

9      }
10
11      static int obtenirTotal() {
12          return total;
13      }
14  };
15
16  template<typename T>
17  int Compteur<T>::total = 0;
18
19  int main() {
20      Compteur<int> c1, c2;
21      Compteur<string> c3;
22
23      cout << "Total pour int : " << Compteur<int>::obtenirTotal() << endl; // 2
24      cout << "Total pour string : " << Compteur<string>::obtenirTotal() << endl;
25          // 1
26  }

```

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 :

```

1  // Exemple d'itérateur unidirectionnel
2  std::vector<int> vec = {1, 2, 3, 4, 5};
3  for (auto it = vec.begin(); it != vec.end(); ++it) {
4      std::cout << *it << " ";
5  }
6
7  // Exemple d'itérateur inversé
8  for (auto it = vec.rbegin(); it != vec.rend(); ++it) {
9      std::cout << *it << " ";
10 }

```

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 :

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

```
1 // Prédicat unaire
2 bool estPair(int x) {
3     return x % 2 == 0;
4 }
5
6 // Prédicat binaire
7 bool comparerDescendant(int a, int b) {
8     return a > b;
9 }
10
11 // Utilisation dans un algorithme
12 std::sort(vec.begin(), vec.end(), comparerDescendant); // Tri en ordre dé
    croissant
```

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 :

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

```
1 std::vector<int> source = {1, 2, 3, 4, 5};
2 std::vector<int> destination(5);
3
4 // Copier tous les éléments
5 std::copy(source.begin(), source.end(), destination.begin());
6
7 // Copier uniquement les éléments pairs
8 std::vector<int> pairs;
9 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 :

```
1 std::vector<int> vec = {1, 2, 3, 4, 5};
2
3 // Transformer chaque élément en son double
4 std::transform(vec.begin(), vec.end(), vec.begin(), [](int x) { return x * 2;
    });
5
6 // Remplacer 4 par 99
7 std::replace(vec.begin(), vec.end(), 4, 99);
8
9 // Remplacer les éléments pairs par 0
10 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 :

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

```
1 std::vector<int> vec = {2, 4, 6, 8};
2
3 // Vérifie si tous les éléments sont pairs
4 bool tousPairs = std::all_of(vec.begin(), vec.end(), [](int x) { return x % 2
  == 0; });
5
6 // Vérifie s'il existe un élément supérieur à 5
7 bool auMoinsUnSup5 = std::any_of(vec.begin(), vec.end(), [](int x) { return x >
  5; });
8
9 // Vérifie si aucun élément n'est impair
10 bool aucunImpair = std::none_of(vec.begin(), vec.end(), [](int x) { return x %
  2 != 0; });
```

12.6 Expressions Lambda

Syntaxe Générale :

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

Exemples :

```
1 auto lambda = [](int x) { return x % 2 == 0; }; // Retourne vrai si x est pair
2 std::copy_if(vec.begin(), vec.end(), std::back_inserter(result), lambda);
```

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 :

```
1 std::vector<int> vec = {1, 3, 5, 7, 9};
2
3 // Trouve la première occurrence de 5
4 auto it = std::find(vec.begin(), vec.end(), 5);
5 if (it != vec.end()) {
6     std::cout << "5 trouvé à l'indice : " << std::distance(vec.begin(), it);
7 }
8
9 // Trouve le premier élément pair
10 auto pair_it = std::find_if(vec.begin(), vec.end(), [](int x) { return x % 2 ==
  0; });
11 if (pair_it != vec.end()) {
12     std::cout << "Premier élément pair : " << *pair_it;
13 }
```

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 :

```
1 // Recherche d'une sous-séquence {3, 5}
2 std::vector<int> sub = {3, 5};
3 auto sub_it = std::search(vec.begin(), vec.end(), sub.begin(), sub.end());
4 if (sub_it != vec.end()) {
5     std::cout << "Sous-séquence trouvée à l'indice : " << std::distance(vec.
6         begin(), sub_it);
7 }
8 // Recherche de 3 occurrences consécutives de 7
9 auto n_it = std::search_n(vec.begin(), vec.end(), 3, 7);
10 if (n_it != vec.end()) {
11     std::cout << "Trois occurrences consécutives trouvées à l'indice : " << std
12         ::distance(vec.begin(), n_it);
13 }
```

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 :

```
1 std::vector<int> vec = {1, 2, 3, 4, 5, 6};
2
3 // Supprimer tous les éléments égaux à 3
4 vec.erase(std::remove(vec.begin(), vec.end(), 3), vec.end());
5
6 // Supprimer tous les éléments pairs
7 vec.erase(std::remove_if(vec.begin(), vec.end(), [](int x) { return x % 2 == 0;
8     }), 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 :

```
1 // Multiplier chaque élément par 2
2 std::transform(vec.begin(), vec.end(), vec.begin(), [](int x) { return x * 2;
3     });
4
5 // Remplacer 4 par 42
6 std::replace(vec.begin(), vec.end(), 4, 42);
7
8 // Remplacer les éléments pairs par 0
9 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 :

```
1 std::random_device rd;
2 std::mt19937 rng(rd());
3 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 :

```
1 // Inverser l'ordre des éléments
2 std::reverse(vec.begin(), vec.end());
3
4 // Faire pivoter pour que le 3 devienne le premier élément
5 std::rotate(vec.begin(), vec.begin() + 2, vec.end());
```

12.10 Expressions Lambda

Syntaxe Générale :

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

Exemples de Lambdas :

```
1 // Lambda pour vérifier si un nombre est pair
2 auto estPair = [](int x) { return x % 2 == 0; };
3
4 // Lambda capturant une variable
5 int seuil = 5;
6 auto plusGrandQueSeuil = [seuil](int x) { return x > seuil; };
7
8 // Appliquer une lambda dans un algorithme
9 std::vector<int> vec = {1, 2, 3, 4, 5, 6};
10 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 :

```
1 void bubbleSort(int arr[], int n) {
2     for (int i = 0; i < n - 1; i++) {
3         for (int j = 0; j < n - i - 1; j++) {
4             if (arr[j] > arr[j + 1]) {
5                 // Échange
6                 int temp = arr[j];
7                 arr[j] = arr[j + 1];
8                 arr[j + 1] = temp;
9             }
10        }
11    }
12 }
```

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 :

```
1 void insertionSort(int arr[], int n) {
2     for (int i = 1; i < n; i++) {
3         int key = arr[i];
4         int j = i - 1;
5
6         // Déplace les éléments plus grands que key
7         while (j >= 0 && arr[j] > key) {
8             arr[j + 1] = arr[j];
9             j--;
10        }
11        arr[j + 1] = key;
12    }
13 }
```

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 :

```
1 #include <algorithm>
2 #include <vector>
3
4 int main() {
5     std::vector<int> vec = {5, 2, 9, 1, 5, 6};
6     std::sort(vec.begin(), vec.end());
7
8     for (int v : vec) {
9         std::cout << v << " ";
10    }
11    return 0;
12 }
```

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 :

```
1 void merge(int arr[], int l, int m, int r) {
2     int n1 = m - l + 1;
3     int n2 = r - m;
4
5     int L[n1], R[n2];
6
7     for (int i = 0; i < n1; i++) L[i] = arr[l + i];
8     for (int i = 0; i < n2; i++) R[i] = arr[m + 1 + i];
9
10    int i = 0, j = 0, k = l;
11    while (i < n1 && j < n2) {
12        if (L[i] <= R[j]) arr[k++] = L[i++];
13        else arr[k++] = R[j++];
14    }
15
16    while (i < n1) arr[k++] = L[i++];
17    while (j < n2) arr[k++] = R[j++];
18 }
19
20 void mergeSort(int arr[], int l, int r) {
21     if (l < r) {
22         int m = l + (r - l) / 2;
23         mergeSort(arr, l, m);
24         mergeSort(arr, m + 1, r);
25         merge(arr, l, m, r);
26     }
27 }
```

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	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$