

Résumé PRG1

Nicolas Reymond

December 17, 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
6.4 Lecture au Clavier (<code>cin</code>)	11
6.5 Flux Fichiers	11

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	Fonctions Génériques avec <code>templates</code>	18
9.5	Surcharge de Fonctions Génériques	18
9.6	Templates de Classe	18
9.7	Exemples Pratiques	19

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.

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
```

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.

5.1 Écriture dans un Fichier

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

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    void afficher() {
13        cout << "(" << x << ", " << y << ")" << endl;
14    }
15 };
16
17 int main() {
18     Point p1(2, 3), p2(4, 5);
19     Point p3 = p1 + p2; // Utilise operator+
20     p3.afficher();      // Affiche (6, 8)
21 }

```

9.4 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 }
```

9.5 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 }
```

9.6 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 }
```

```

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 }

```

9.7 Exemples Pratiques

Combinaison de Templates et Surchage :

```

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 }

```