C++

Bases

Fonction `main`

La fonction `main` est nécessaire à l'exécution du programme, c'est la première fonction qui est exécutée.

```
int main()
{
        // Code
        return 0;
}
```

Commentaires

Les commentaires sont des lignes de code qui ne sont pas exécutées par le programme, ils permettent de commenter le code pour le rendre plus compréhensible.

```
// Ceci est un commentaire sur une ligne

/*
        Ceci est un commentaire
        sur plusieurs lignes
*/
```

Variables

Les variables sont des espaces mémoires qui permettent de stocker des valeurs.

```
bool a = true; // Variable de type booléen
int a = 5; // Variable de type entier
float b = 5.5f; // Variable de type décimal
```

```
double c = 5.5; // Variable de type décimal avec une plus grande précision
char c = 'a'; // Variable de type caractère
std::string d = "Hello World"; // Variable de type chaîne de caractères
```

Constantes

Les constantes sont des variables dont la valeur ne peut pas être modifiée.

```
const bool a = true; // Constante de type booléen
const int a = 5; // Constante de type entier
const float b = 5.5f; // Constante de type décimal
const double c = 5.5; // Constante de type décimal avec une plus grande précision
const char c = 'a'; // Constante de type caractère
const std::string d = "Hello World"; // Constante de type chaîne de caractères
```

Opérateurs

Les opérateurs sont des symboles qui permettent d'effectuer des opérations sur des variables.

```
int a = 5 + 5; // Addition
int b = 5 - 5; // Soustraction
int c = 5 * 5; // Multiplication
int d = 5 / 5; // Division
int e = 5 % 5; // Modulo
```

Conditions

Les conditions permettent d'effectuer des actions en fonction de la valeur d'une variable.

```
if (a == 5) // Si a est égal à 5
{
        // Code
}
else if (a > 5) // Si a est supérieur à 5
{
        // Code
}
else if (a < 5) // Si a est inférieur à 5
{
        // Code
}
else if (a ≥ 5) // Si a est supérieur ou égal à 5
{
        // Code
}
else if (a ≤ 5) // Si a est inférieur ou égal à 5
{
        // Code
}
else if (a ≠ 5) // Si a est différent de 5
{
        // Code
}
else // Sinon
{
        // Code
}
```

Boucles

Les boucles permettent d'effectuer des actions plusieurs fois.

```
for (int i = 0; i < 10; i++) // Boucle for
{
        // Code
}

while (a < 10) // Boucle while
{
```

```
        // Code
        break; // Sort de la boucle
        continue; // Passe à l'itération suivante
}

do // Boucle do while
{
        // Code
} while (a < 10);
```

Fonctions

Les fonctions sont des blocs de code qui peuvent être appelés plusieurs fois.

```
void fonction() // Fonction sans paramètres
{
        // Code
}

void fonction(int a) // Fonction avec un paramètre
{
        // Code
}

void fonction(int a, int b) // Fonction avec plusieurs
paramètres
{
        // Code
}

int fonction() // Fonction qui retourne un entier
{
        // Code
        return 0;
}

int fonction(int a) // Fonction avec un paramètre qui
retourne un entier
{
```

```
        // Code
        return 0;
}

int fonction(int a, int b) // Fonction avec plusieurs
paramètres qui retourne un entier
{
        // Code
        return 0;
}
```

Tableaux

Les tableaux sont des variables qui peuvent contenir plusieurs valeurs.

```
int a[5]; // Tableau d'entiers de taille 5

int a[] = { 1, 2, 3, 4, 5 }; // Tableau d'entiers de taille
5

int a[5] = { 1, 2, 3, 4, 5 }; // Tableau d'entiers de taille
5

int a[5] = { 1, 2, 3 }; // Tableau d'entiers de taille 5
avec les 3 premières valeurs initialisées

int a[5] = { }; // Tableau d'entiers de taille 5 avec toutes
les valeurs initialisées à 0

int a[5] = { 1 }; // Tableau d'entiers de taille 5 avec la
première valeur initialisée à 1 et les autres à 0

int a[5] = { 1, 2, 3, 4, 5 }; // Tableau d'entiers de taille
5

float b[5]; // Tableau de décimaux de taille 5

double c[5]; // Tableau de décimaux avec une plus grande
précision de taille 5
```

```
char d[5]; // Tableau de caractères de taille 5

std::string e[5]; // Tableau de chaînes de caractères de
taille 5
```

Pointeurs

Les pointeurs sont des variables qui contiennent l'adresse d'une autre variable.

```
int a = 5; // Variable de type entier

int *b = &a; // Pointeur sur la variable a

std::cout << b << std::endl; // Affiche l'adresse de la
variable a

std::cout << *b << std::endl; // Affiche la valeur de la
variable a
```

Structures

Les structures sont des variables qui peuvent contenir plusieurs variables.

```
struct Personne // Structure Personne
{
        std::string nom;
        std::string prenom;
        int age;
};

Personne personne; // Variable de type Personne

personne.nom = "Doe";
```

```
personne.prenom = "John";
personne.age = 42;
```

Classes

Les classes sont des structures qui peuvent contenir des fonctions.

```
class Personne // Classe Personne
{
        public:
                std::string nom;
                std::string prenom;
                int age;

                void afficher()
                {
                        std::cout << "Nom : " << nom <<
std::endl;
                        std::cout << "Prenom : " << prenom
<< std::endl;
                        std::cout << "Age : " << age <<
std::endl;
                }
};

Personne personne; // Variable de type Personne

personne.nom = "Doe";
personne.prenom = "John";
personne.age = 42;

personne.afficher();
```

Héritage

L'héritage permet de créer une classe à partir d'une autre classe.

```
class Personne // Classe Personne
{
        public:
                std::string nom;
                std::string prenom;
                int age;

                void afficher()
                {
                        std::cout << "Nom : " << nom <<
std::endl;
                        std::cout << "Prenom : " << prenom
<< std::endl;
                        std::cout << "Age : " << age <<
std::endl;
                }
};

class Etudiant : public Personne // Classe Etudiant qui
hérite de la classe Personne
{
        public:
                std::string ecole;
                std::string classe;

                void afficher()
                {
                        std::cout << "Nom : " << nom <<
std::endl;
                        std::cout << "Prenom : " << prenom
<< std::endl;
                        std::cout << "Age : " << age <<
std::endl;
                        std::cout << "Ecole : " << ecole <<
std::endl;
                        std::cout << "Classe : " << classe
<< std::endl;
                }
};
```

```
Etudiant etudiant; // Variable de type Etudiant

etudiant.nom = "Doe";
etudiant.prenom = "John";
etudiant.age = 42;
etudiant.ecole = "Ecole";
etudiant.classe = "Classe";

etudiant.afficher();
```

Polymorphisme

Le polymorphisme permet de créer une fonction qui peut avoir plusieurs comportements.

```
class Personne // Classe Personne
{
        public:
                std::string nom;
                std::string prenom;
                int age;

                virtual void afficher() // Fonction
virtuelle
                {
                        std::cout << "Nom : " << nom <<
std::endl;
                        std::cout << "Prenom : " << prenom
<< std::endl;
                        std::cout << "Age : " << age <<
std::endl;
                }
};

class Etudiant : public Personne // Classe Etudiant qui
hérite de la classe Personne
{
        public:
                std::string ecole;
```

```
                std::string classe;

                void afficher() // Fonction qui remplace la
fonction de la classe Personne
                {
                        std::cout << "Nom : " << nom <<
std::endl;
                        std::cout << "Prenom : " << prenom
<< std::endl;
                        std::cout << "Age : " << age <<
std::endl;
                        std::cout << "Ecole : " << ecole <<
std::endl;
                        std::cout << "Classe : " << classe
<< std::endl;
                }
};

Personne *personne = new Personne(); // Variable de type
Personne

personne->nom = "Doe";
personne->prenom = "John";
personne->age = 42;

personne->afficher();

personne = new Etudiant(); // Variable de type Etudiant

personne->nom = "Doe";
personne->prenom = "John";
personne->age = 42;
personne->ecole = "Ecole";
personne->classe = "Classe";

personne->afficher();
```

Encapsulation

L'encapsulation permet de cacher des variables et des fonctions.

```
class Personne // Classe Personne
{
        private:
                std::string nom;
                std::string prenom;
                int age;

        public:
                void setNom(std::string nom)
                {
                        this->nom = nom;
                }

                std::string getNom()
                {
                        return nom;
                }

                void setPrenom(std::string prenom)
                {
                        this->prenom = prenom;
                }

                std::string getPrenom()
                {
                        return prenom;
                }

                void setAge(int age)
                {
                        this->age = age;
                }

                int getAge()
                {
                        return age;
                }
};

Personne personne; // Variable de type Personne
```

```
personne.setNom("Doe");
personne.setPrenom("John");
personne.setAge(42);

std::cout << "Nom : " << personne.getNom() << std::endl;
std::cout << "Prenom : " << personne.getPrenom() <<
std::endl;
std::cout << "Age : " << personne.getAge() << std::endl;
```

Surcharge

La surcharge permet de créer plusieurs fonctions avec le même nom mais avec des paramètres différents.

```
class Personne // Classe Personne
{
        public:
                std::string nom;
                std::string prenom;
                int age;

                void afficher()
                {
                        std::cout << "Nom : " << nom <<
std::endl;
                        std::cout << "Prenom : " << prenom
<< std::endl;
                        std::cout << "Age : " << age <<
std::endl;
                }

                void afficher(std::string ecole, std::string
classe)
                {
                        std::cout << "Nom : " << nom <<
std::endl;
                        std::cout << "Prenom : " << prenom
<< std::endl;
```

```
                        std::cout << "Age : " << age <<
std::endl;
                        std::cout << "Ecole : " << ecole <<
std::endl;
                        std::cout << "Classe : " << classe
<< std::endl;
                }
};

Personne personne; // Variable de type Personne

personne.nom = "Doe";
personne.prenom = "John";
personne.age = 42;

personne.afficher();

personne.afficher("Ecole", "Classe");
```

Exceptions

Les exceptions permettent de gérer les erreurs.

```
try
{
        // Code
}
catch (std::exception const& e)
{
        std::cerr << "Erreur : " << e.what() << std::endl;
}
```

Entrées / Sorties

Les entrées / sorties permettent d'interagir avec l'utilisateur.

```
std::cout << "Hello World" << std::endl; // Affiche Hello
```

```
World

std::cin >> a; // Demande à l'utilisateur d'entrer une
valeur
```

Fichiers

Les fichiers permettent de lire et d'écrire dans des fichiers.

```
std::ofstream fichier("fichier.txt", std::ios::out |
std::ios::trunc); // Ouvre le fichier en écriture

if (fichier)
{
        fichier << "Hello World" << std::endl; // Ecrit
Hello World dans le fichier
        fichier.close(); // Ferme le fichier
}

std::ifstream fichier("fichier.txt", std::ios::in); // Ouvre
le fichier en lecture

if (fichier)
{
        std::string ligne;

        while (std::getline(fichier, ligne)) // Lit le
fichier ligne par ligne
        {
                std::cout << ligne << std::endl; // Affiche
la ligne
        }

        fichier.close(); // Ferme le fichier
}
```

Pointeurs intelligents

Les pointeurs intelligents permettent de gérer automatiquement la mémoire.

```
std::unique_ptr<int> a = std::make_unique<int>(5); //
Pointeur intelligent unique

std::shared_ptr<int> b = std::make_shared<int>(5); //
Pointeur intelligent partagé

std::weak_ptr<int> c = b; // Pointeur intelligent faible
```

Lambda

Les lambdas sont des fonctions anonymes.

```
auto a = []() // Lambda sans paramètres
{
        // Code
};

auto b = [](int a) // Lambda avec un paramètre
{
        // Code
};

auto c = [](int a, int b) // Lambda avec plusieurs
paramètres
{
        // Code
};

auto d = []() -> int // Lambda qui retourne un entier
{
        // Code
        return 0;
};
```

```
auto e = [](int a) -> int // Lambda avec un paramètre qui
retourne un entier
{
        // Code
        return 0;
};

auto f = [](int a, int b) -> int // Lambda avec plusieurs
paramètres qui retourne un entier
{
        // Code
        return 0;
};
```

Fonctions lambda

Les fonctions lambda permettent d'effectuer des actions sur des conteneurs.

```
std::vector<int> a = { 1, 2, 3, 4, 5 }; // Conteneur de type
vecteur

std::for_each(a.begin(), a.end(), [](int a) // Parcours le
conteneur
{
        std::cout << a << std::endl; // Affiche la valeur
});

std::vector<int> b = { 1, 2, 3, 4, 5 }; // Conteneur de type
vecteur

std::transform(b.begin(), b.end(), b.begin(), [](int a) //
Parcours le conteneur
{
        return a * 2; // Multiplie la valeur par 2
});

std::vector<int> c = { 1, 2, 3, 4, 5 }; // Conteneur de type
vecteur
```

```
std::vector<int> d; // Conteneur de type vecteur

std::copy_if(c.begin(), c.end(), std::back_inserter(d), []
(int a) // Parcours le conteneur
{
        return a % 2 == 0; // Ajoute la valeur si elle est
paire
});

std::vector<int> e = { 1, 2, 3, 4, 5 }; // Conteneur de type
vecteur

std::vector<int> f; // Conteneur de type vecteur

std::remove_copy_if(e.begin(), e.end(),
std::back_inserter(f), [](int a) // Parcours le conteneur
{
        return a % 2 == 0; // Ajoute la valeur si elle est
impaire
});
```

Gesion de plusieurs fichiers

La gestion de plusieurs fichiers permet de séparer le code en plusieurs fichiers.

```
// main.cpp

#include "fonctions.h" // Importe le fichier fonctions.h

int main()
{
        fonction(); // Appelle la fonction fonction
        return 0;
}
```

```
// fonctions.h

#pragma once // Empêche les inclusions multiples

void fonction(); // Déclaration de la fonction fonction
```

```
// fonctions.cpp

#include "fonctions.h" // Importe le fichier fonctions.h

void fonction() // Définition de la fonction fonction
{
        // Code
}
```

Bibliothèques

Affichage

La bibliothèque `iostream` permet d'afficher du texte.

```

#include <iostream> // Importe la bibliothèque iostream

std::cout << "Hello World" << std::endl; // Affiche Hello
World
```

Chaînes de caractères

La bibliothèque `string` permet de manipuler des chaînes de caractères.

```

#include <string> // Importe la bibliothèque string

std::string a = "Hello World"; // Chaîne de caractères

std::cout << a << std::endl; // Affiche Hello World
```

```
std::string b = a + " !"; // Concaténation de chaînes de
caractères

std::cout << b << std::endl; // Affiche Hello World !

std::string c = b.substr(0, 5); // Extrait une partie de la
chaîne de caractères

std::cout << c << std::endl; // Affiche Hello

std::string d = b.substr(6, 5); // Extrait une partie de la
chaîne de caractères

std::cout << d << std::endl; // Affiche World

std::string e = b.substr(12, 1); // Extrait une partie de la
chaîne de caractères

std::cout << e << std::endl; // Affiche !

std::string f = b.substr(0, 5) + b.substr(6, 5) +
b.substr(12, 1); // Extrait une partie de la chaîne de
caractères

std::cout << f << std::endl; // Affiche HelloWorld!
```

Tableaux

La bibliothèque `array` permet de manipuler des tableaux.

```
#include <array> // Importe la bibliothèque array

std::array<int, 5> a; // Tableau d'entiers de taille 5

std::array<int, 5> b = { 1, 2, 3, 4, 5 }; // Tableau
d'entiers de taille 5

std::array<int, 5> c = { 1, 2, 3 }; // Tableau d'entiers de
```

```
taille 5 avec les 3 premières valeurs initialisées

std::array<int, 5> d = { }; // Tableau d'entiers de taille 5
avec toutes les valeurs initialisées à 0

std::array<int, 5> e = { 1 }; // Tableau d'entiers de taille
5 avec la première valeur initialisée à 1 et les autres à 0

std::array<int, 5> f = { 1, 2, 3, 4, 5 }; // Tableau
d'entiers de taille 5

std::array<float, 5> g; // Tableau de décimaux de taille 5

std::array<double, 5> h; // Tableau de décimaux avec une
plus grande précision de taille 5

std::array<char, 5> i; // Tableau de caractères de taille 5

std::array<std::string, 5> j; // Tableau de chaînes de
caractères de taille 5
```

Vecteurs

La bibliothèque `vector` permet de manipuler des vecteurs.

```
#include <vector> // Importe la bibliothèque vector

std::vector<int> a; // Vecteur d'entiers

std::vector<int> b = { 1, 2, 3, 4, 5 }; // Vecteur d'entiers

std::vector<int> c = { 1, 2, 3 }; // Vecteur d'entiers avec
les 3 premières valeurs initialisées

std::vector<int> d(5); // Vecteur d'entiers de taille 5

std::vector<int> e(5, 1); // Vecteur d'entiers de taille 5
avec toutes les valeurs initialisées à 1
```

```
std::vector<int> f(5, 1); // Vecteur d'entiers de taille 5 avec la première valeur initialisée à 1 et les autres à 0

std::vector<int> g = { 1, 2, 3, 4, 5 }; // Vecteur d'entiers

std::vector<float> h; // Vecteur de décimaux

std::vector<double> i; // Vecteur de décimaux avec une plus grande précision

std::vector<char> j; // Vecteur de caractères

std::vector<std::string> k; // Vecteur de chaînes de caractères
```

Listes

La bibliothèque `list` permet de manipuler des listes.

```
#include <list> // Importe la bibliothèque list

std::list<int> a; // Liste d'entiers

std::list<int> b = { 1, 2, 3, 4, 5 }; // Liste d'entiers

std::list<int> c = { 1, 2, 3 }; // Liste d'entiers avec les 3 premières valeurs initialisées

std::list<int> d(5); // Liste d'entiers de taille 5

std::list<int> e(5, 1); // Liste d'entiers de taille 5 avec toutes les valeurs initialisées à 1

std::list<int> f(5, 1); // Liste d'entiers de taille 5 avec la première valeur initialisée à 1 et les autres à 0

std::list<int> g = { 1, 2, 3, 4, 5 }; // Liste d'entiers

std::list<float> h; // Liste de décimaux
```

```
std::list<double> i; // Liste de décimaux avec une plus grande précision

std::list<char> j; // Liste de caractères

std::list<std::string> k; // Liste de chaînes de caractères
```

Files

La bibliothèque `queue` permet de manipuler des files.

```
#include <queue> // Importe la bibliothèque queue

std::queue<int> a; // File d'entiers

std::queue<int> b = { 1, 2, 3, 4, 5 }; // File d'entiers

std::queue<int> c = { 1, 2, 3 }; // File d'entiers avec les 3 premières valeurs initialisées

std::queue<int> d; // File d'entiers

d.push(1); // Ajoute la valeur 1 à la file

d.push(2); // Ajoute la valeur 2 à la file

d.push(3); // Ajoute la valeur 3 à la file

d.push(4); // Ajoute la valeur 4 à la file

d.push(5); // Ajoute la valeur 5 à la file

std::cout << d.front() << std::endl; // Affiche la valeur 1

d.pop(); // Supprime la valeur 1 de la file

std::cout << d.front() << std::endl; // Affiche la valeur 2
```

```
d.pop(); // Supprime la valeur 2 de la file

std::cout << d.front() << std::endl; // Affiche la valeur 3

d.pop(); // Supprime la valeur 3 de la file

std::cout << d.front() << std::endl; // Affiche la valeur 4

d.pop(); // Supprime la valeur 4 de la file

std::cout << d.front() << std::endl; // Affiche la valeur 5

d.pop(); // Supprime la valeur 5 de la file

std::queue<int> e; // File d'entiers

e.push(1); // Ajoute la valeur 1 à la file

e.push(2); // Ajoute la valeur 2 à la file

e.push(3); // Ajoute la valeur 3 à la file

e.push(4); // Ajoute la valeur 4 à la file

e.push(5); // Ajoute la valeur 5 à la file

std::cout << e.back() << std::endl; // Affiche la valeur 5

e.pop(); // Supprime la valeur 5 de la file

std::cout << e.back() << std::endl; // Affiche la valeur 4

e.pop(); // Supprime la valeur 4 de la file

std::cout << e.back() << std::endl; // Affiche la valeur 3

e.pop(); // Supprime la valeur 3 de la file

std::cout << e.back() << std::endl; // Affiche la valeur 2

e.pop(); // Supprime la valeur 2 de la file
```

```
std::cout << e.back() << std::endl; // Affiche la valeur 1

e.pop(); // Supprime la valeur 1 de la file
```

Piles

La bibliothèque `stack` permet de manipuler des piles.

```
#include <stack> // Importe la bibliothèque stack

std::stack<int> a; // Pile d'entiers

std::stack<int> b = { 1, 2, 3, 4, 5 }; // Pile d'entiers

std::stack<int> c = { 1, 2, 3 }; // Pile d'entiers avec les
3 premières valeurs initialisées

std::stack<int> d; // Pile d'entiers

d.push(1); // Ajoute la valeur 1 à la pile

d.push(2); // Ajoute la valeur 2 à la pile

d.push(3); // Ajoute la valeur 3 à la pile

d.push(4); // Ajoute la valeur 4 à la pile

d.push(5); // Ajoute la valeur 5 à la pile

std::cout << d.top() << std::endl; // Affiche la valeur 5

d.pop(); // Supprime la valeur 5 de la pile

std::cout << d.top() << std::endl; // Affiche la valeur 4

d.pop(); // Supprime la valeur 4 de la pile

std::cout << d.top() << std::endl; // Affiche la valeur 3
```

```
d.pop(); // Supprime la valeur 3 de la pile

std::cout << d.top() << std::endl; // Affiche la valeur 2

d.pop(); // Supprime la valeur 2 de la pile

std::cout << d.top() << std::endl; // Affiche la valeur 1

d.pop(); // Supprime la valeur 1 de la pile
```

Ensembles

La bibliothèque `set` permet de manipuler des ensembles.

```
#include <set> // Importe la bibliothèque set

std::set<int> a; // Ensemble d'entiers

std::set<int> b = { 1, 2, 3, 4, 5 }; // Ensemble d'entiers

std::set<int> c = { 1, 2, 3 }; // Ensemble d'entiers avec
les 3 premières valeurs initialisées

std::set<int> d; // Ensemble d'entiers

d.insert(1); // Ajoute la valeur 1 à l'ensemble

d.insert(2); // Ajoute la valeur 2 à l'ensemble

d.insert(3); // Ajoute la valeur 3 à l'ensemble

d.insert(4); // Ajoute la valeur 4 à l'ensemble

d.insert(5); // Ajoute la valeur 5 à l'ensemble

std::cout << d.count(1) << std::endl; // Affiche 1 si la
valeur 1 est présente dans l'ensemble sinon 0

std::cout << d.count(2) << std::endl; // Affiche 1 si la
```

```
valeur 2 est présente dans l'ensemble sinon 0

std::cout << d.count(3) << std::endl; // Affiche 1 si la
valeur 3 est présente dans l'ensemble sinon 0

std::cout << d.count(4) << std::endl; // Affiche 1 si la
valeur 4 est présente dans l'ensemble sinon 0

std::cout << d.count(5) << std::endl; // Affiche 1 si la
valeur 5 est présente dans l'ensemble sinon 0

std::set<int> e; // Ensemble d'entiers

e.insert(1); // Ajoute la valeur 1 à l'ensemble

e.insert(2); // Ajoute la valeur 2 à l'ensemble

e.insert(3); // Ajoute la valeur 3 à l'ensemble

e.insert(4); // Ajoute la valeur 4 à l'ensemble

e.insert(5); // Ajoute la valeur 5 à l'ensemble

std::cout << e.size() << std::endl; // Affiche la taille de
l'ensemble

e.erase(1); // Supprime la valeur 1 de l'ensemble

e.erase(2); // Supprime la valeur 2 de l'ensemble

e.erase(3); // Supprime la valeur 3 de l'ensemble

e.erase(4); // Supprime la valeur 4 de l'ensemble

e.erase(5); // Supprime la valeur 5 de l'ensemble

std::cout << e.size() << std::endl; // Affiche la taille de
l'ensemble
```

Dictionnaires

La bibliothèque `map` permet de manipuler des dictionnaires.

```
#include <map> // Importe la bibliothèque map

std::map<std::string, int> a; // Dictionnaire de chaînes de
caractères et d'entiers

std::map<std::string, int> b = { { "a", 1 }, { "b", 2 }, {
"c", 3 }, { "d", 4 }, { "e", 5 } }; // Dictionnaire de
chaînes de caractères et d'entiers

std::map<std::string, int> c = { { "a", 1 }, { "b", 2 }, {
"c", 3 } }; // Dictionnaire de chaînes de caractères et
d'entiers avec les 3 premières valeurs initialisées

std::map<std::string, int> d; // Dictionnaire de chaînes de
caractères et d'entiers

d["a"] = 1; // Ajoute la valeur 1 à la clé a

d["b"] = 2; // Ajoute la valeur 2 à la clé b

d["c"] = 3; // Ajoute la valeur 3 à la clé c

d["d"] = 4; // Ajoute la valeur 4 à la clé d

d["e"] = 5; // Ajoute la valeur 5 à la clé e

std::cout << d["a"] << std::endl; // Affiche la valeur 1 de
la clé a

std::cout << d["b"] << std::endl; // Affiche la valeur 2 de
la clé b

std::cout << d["c"] << std::endl; // Affiche la valeur 3 de
la clé c

std::cout << d["d"] << std::endl; // Affiche la valeur 4 de
la clé d
```

```
std::cout << d["e"] << std::endl; // Affiche la valeur 5 de
la clé e

std::map<std::string, int> e; // Dictionnaire de chaînes de
caractères et d'entiers

e["a"] = 1; // Ajoute la valeur 1 à la clé a

e["b"] = 2; // Ajoute la valeur 2 à la clé b

e["c"] = 3; // Ajoute la valeur 3 à la clé c

e["d"] = 4; // Ajoute la valeur 4 à la clé d

e["e"] = 5; // Ajoute la valeur 5 à la clé e

std::cout << e.size() << std::endl; // Affiche la taille du
dictionnaire

e.erase("a"); // Supprime la clé a du dictionnaire

e.erase("b"); // Supprime la clé b du dictionnaire

e.erase("c"); // Supprime la clé c du dictionnaire

e.erase("d"); // Supprime la clé d du dictionnaire

e.erase("e"); // Supprime la clé e du dictionnaire

std::cout << e.size() << std::endl; // Affiche la taille du
dictionnaire
```

Algorithmes

La bibliothèque `algorithm` permet de manipuler des algorithmes.

```
#include <algorithm> // Importe la bibliothèque algorithm

std::vector<int> a = { 1, 2, 3, 4, 5 }; // Conteneur de type
```

```
vecteur

std::for_each(a.begin(), a.end(), [](int a) // Parcours le
conteneur
{
        std::cout << a << std::endl; // Affiche la valeur
});

std::vector<int> b = { 1, 2, 3, 4, 5 }; // Conteneur de type
vecteur

std::transform(b.begin(), b.end(), b.begin(), [](int a) //
Parcours le conteneur
{
        return a * 2; // Multiplie la valeur par 2
});

std::vector<int> c = { 1, 2, 3, 4, 5 }; // Conteneur de type
vecteur

std::vector<int> d; // Conteneur de type vecteur

std::copy_if(c.begin(), c.end(), std::back_inserter(d), []
(int a) // Parcours le conteneur
{
        return a % 2 == 0; // Ajoute la valeur si elle est
paire
});

std::vector<int> e = { 1, 2, 3, 4, 5 }; // Conteneur de type
vecteur

std::vector<int> f; // Conteneur de type vecteur

std::remove_copy_if(e.begin(), e.end(),
std::back_inserter(f), [](int a) // Parcours le conteneur
{
        return a % 2 == 0; // Ajoute la valeur si elle est
impaire
});
```

Fonctions lambda

La bibliothèque `functional` permet de manipuler des fonctions lambda.

```
#include <functional> // Importe la bibliothèque functional

std::vector<int> a = { 1, 2, 3, 4, 5 }; // Conteneur de type
vecteur

std::for_each(a.begin(), a.end(), [](int a) // Parcours le
conteneur
{
        std::cout << a << std::endl; // Affiche la valeur
});

std::vector<int> b = { 1, 2, 3, 4, 5 }; // Conteneur de type
vecteur

std::transform(b.begin(), b.end(), b.begin(), [](int a) //
Parcours le conteneur
{
        return a * 2; // Multiplie la valeur par 2
});

std::vector<int> c = { 1, 2, 3, 4, 5 }; // Conteneur de type
vecteur

std::vector<int> d; // Conteneur de type vecteur

std::copy_if(c.begin(), c.end(), std::back_inserter(d), []
(int a) // Parcours le conteneur
{
        return a % 2 == 0; // Ajoute la valeur si elle est
paire
});

std::vector<int> e = { 1, 2, 3, 4, 5 }; // Conteneur de type
vecteur

std::vector<int> f; // Conteneur de type vecteur
```

```
std::remove_copy_if(e.begin(), e.end(),
std::back_inserter(f), [](int a) // Parcours le conteneur
{
        return a % 2 == 0; // Ajoute la valeur si elle est
impaire
});
```

Entrées / Sorties

La bibliothèque `fstream` permet de manipuler des entrées / sorties.

```
#include <fstream> // Importe la bibliothèque fstream

std::ofstream fichier("fichier.txt", std::ios::out |
std::ios::trunc); // Ouvre le fichier en écriture

if (fichier)
{
        fichier << "Hello World" << std::endl; // Ecrit
Hello World dans le fichier
        fichier.close(); // Ferme le fichier
}

std::ifstream fichier("fichier.txt", std::ios::in); // Ouvre
le fichier en lecture

if (fichier)
{
        std::string ligne;

        while (std::getline(fichier, ligne)) // Lit le
fichier ligne par ligne
        {
                std::cout << ligne << std::endl; // Affiche
la ligne
        }
```

```
        fichier.close(); // Ferme le fichier
}
```