Cours 07 - Programmation Générique

Templates en C++

Pourquoi la programmation générique ?

Problème : code répétitif

```
int min(int a, int b) {
    return (a < b) ? a : b;
}

double min(double a, double b) {
    return (a < b) ? a : b;
}

string min(string a, string b) {
    return (a < b) ? a : b;
}

// ... pour chaque type !
```

Inconvénients

- Duplication de code
- Maintenance difficile
- Erreurs lors des modifications

Solution : les templates

```
template <typename T>
T min(T a, T b) {
    return (a < b) ? a : b;
}

// Fonctionne pour tous les types
// supportant l'opérateur <
```

Avantages

- Code unique
- Vérification à la compilation
- Aucun coût à l'exécution
- Type-safe (contrairement aux macros)

Philosophie

- Réfléchir plus pour travailler moins
- Abstraction sur les types

Mise en place des templates

Syntaxe générale

```
template <typename T1, typename T2, ...>
// Déclaration de fonction ou classe
```

Mots-clés équivalents

```
template <typename T>  // Moderne
template <class T>    // Ancien (équivalent)
```

Les deux sont équivalents, `typename` est préféré

Paramètres possibles

Types

```
template <typename T>
```

Valeurs (entiers, pointeurs)

```
template <int N>
template <typename T, int Size>
```

Mixte

```
template <typename T, int N, typename U>
```

Templates de Fonctions

Généricité au niveau des fonctions

Alternatives sans templates

1. Surcharge manuelle

```
inline int min(int a, int b) {
    return (a < b) ? a : b;
}

inline double min(double a, double b) {
    return (a < b) ? a : b;
}

// Répétitif et fastidieux
```

Inconvénients

- Beaucoup de code
- Oublis possibles
- Maintenance complexe

2. Macro préprocesseur

```
#define min(a, b) ((a < b) ? (a) : (b))
```

Inconvénients

- Pas de vérification de type
- Évaluations multiples dangereuses
- Pas de débogage
- Pollution namespace

```
int x = 5;
min(x++, 10); // x incrémenté 2 fois !
```

Les templates sont la solution moderne

Comparaison : Macro vs Surcharge vs Template (1/2)

Critère	Macro `#define`	Surcharge	Template
Type-safe	❌ Non	✅ Oui	✅ Oui
Vérification	Préprocesseur	Compilation	Compilation
Message d'erreur	Cryptique	Clair	Moyennement clair
Débogage	❌ Difficile	✅ Facile	✅ Facile
Évaluation args	❌ Multiple	✅ Une seule	✅ Une seule
Code généré	Expansion textuelle	Une par type	Une par type instancié
Maintenance	❌ Difficile	Répétitif	✅ Un seul code
Performance	✅ Inline	✅ Inline	✅ Inline + optimisations
Portée	Globale (⚠️)	Namespace	Namespace
Spécialisation	❌ Impossible	Manuelle	✅ Automatique

Comparaison : Macro vs Surcharge vs Template (2/2)

Macro : À éviter

- Aucune sécurité
- Effets de bord
- Difficile à déboguer

Surcharge : Acceptable

- Si peu de types
- Code simple
- Besoin explicite

Template : Préféré

- Code générique
- Type-safe
- Performant

Template de fonction - Syntaxe

Définition

```
template <typename T>
T min(const T& a, const T& b) {
    return (a < b) ? a : b;
}
```

Règle importante

- Les types abstraits (`T`) doivent intervenir dans les paramètres d'entrée
- Le compilateur déduit les types à partir des arguments
- Déduction automatique basée sur la signature

Utilisation

```
int main() {
    int i = min(3, 5);         // T = int
    double d = min(3.2, 5.1); // T = double
    string s = min("abc", "def"); // T = string

    // Déduction automatique !
}
```

Contraintes implicites

- Le type `T` doit supporter `operator<`
- Le type `T` doit être copiable
- Vérification à la compilation

Déduction de type et ambiguïtés

Problème : types différents

```cpp
template <typename T>
T min(const T& a, const T& b) {
    return (a < b) ? a : b;
}

int main() {
    int i = 2;
    double d = 1.9;

    min(i, d); // ERREUR !
    // Quel type pour T ? int ou double ?
}
```

Erreur du compilateur

```
conflicting types for parameter T
cannot deduce T
```

Solutions

1. Spécification explicite

```cpp
min<double>(i, d); // Force T = double
// i converti en double
```

2. Conversion manuelle

```cpp
min(double(i), d); // Les deux sont double
min(i, int(d));    // Les deux sont int
```

3. Template avec 2 types

```cpp
template <typename T1, typename T2>
T1 min(const T1& a, const T2& b) {
    return (a < b) ? a : T1(b);
}

// Déconseillé : comportement surprenant
min(2.5, 3)  // → 2.5 (T1=double)
min(3, 2.5)  // → 2   (T1=int, troncature!)
```

Type de retour et spécification explicite

Problème : type de retour inconnu

```
template <typename T>
T convert(int& i) {
    return T(i);
}

int main() {
    double x = convert(2); // ERREUR !
}
```

Solution : spécification explicite

```
template <typename T>
T convert(int& i) {
    return T(i);
}

int main() {
    double x = convert<double>(2); // OK
}
```

Pourquoi ?

1. Compilateur cherche `convert(int)`
2. Trouve `template<typename T> convert(int&)`
3. Ne peut pas déduire `T` (pas dans paramètres)
4. Erreur : fonction non trouvée

Généralisation

```
template <typename T1, typename T2>
T1 convert(const T2& x) {
    return T1(x);
}

// Utilisation
double d = convert<double, int>(42);
double e = convert<double>(42); // T2 déduit
```

Surcharge de templates

Templates peuvent être surchargés

```cpp
// Version 2 paramètres
template <typename T>
T min(const T& a, const T& b) {
    return (a < b) ? a : b;
}

// Version 3 paramètres
template <typename T>
T min(const T& a, const T& b, const T& c) {
    return min(min(a, b), c);
}

// Version tableau
template <typename T, int N>
T min(const T (&arr)[N]) {
    T result = arr[0];
    for (int i = 1; i < N; ++i) {
        if (arr[i] < result)
            result = arr[i];
    }
    return result;
}
```

Utilisation

```cpp
int a = min(3, 5);           // 2 params
int b = min(3, 5, 7);        // 3 params
int arr[] = {3, 1, 4, 1, 5};
int c = min(arr);            // tableau
```

Compatibilité

- Fonctionne pour tous les types supportant :
  - `operator<`
  - Constructeur par copie `T(const T&)`

Attention : pas pour char*

```cpp
const char* s1 = "abc";
const char* s2 = "def";
min(s1, s2); // Compare les pointeurs !
```

Spécialisation de templates

Cas particulier : char*

```
// Template général
template <typename T>
T min(const T& a, const T& b) {
    return (a < b) ? a : b;
}

// Spécialisation pour char*
template <>
const char* min(const char*& a,
                const char*& b) {
    return (strcmp(a, b) < 0) ? a : b;
}

// Ou sans template (encore mieux)
const char* min(const char* a,
                const char* b) {
    return (strcmp(a, b) < 0) ? a : b;
}
```

Règle de résolution

En cas de conflit, priorité :

1. **Fonction non-template exacte**
2. **Spécialisation template**
3. **Template général**

Exemple

```
const char* s1 = "abc";
const char* s2 = "def";

min(s1, s2); // Appelle version char*
             // pas le template général

int i = min(3, 5); // Appelle template général
```

Conseil

- Spécialiser quand nécessaire
- Préférer fonction non-template si possible

Templates de Classes

Généricité au niveau des classes

Classe template - Principe

Syntaxe générale

```
template <typename T1, typename T2, ...>
class NomClasse {
    // Utilisation de T1, T2, ...
};
```

Généralisation

- Concept de template étendu aux classes
- Types abstraits utilisables comme types standards
- Création de conteneurs génériques

Exemple : vecteur générique

```
template <typename T>
class Vecteur {
private:
    T* data;
    int taille;

public:
    Vecteur(int n) : taille(n) {
        data = new T[n];
    }

    ~Vecteur() {
        delete[] data;
    }

    T& operator[](int i) {
        return data[i];
    }

    int size() const {
        return taille;
    }
};
```

Instanciation de classe template

Déclaration et utilisation

```
Vecteur<int> v1(10);      // Vecteur d'entiers
Vecteur<double> v2(5);    // Vecteur de doubles
Vecteur<string> v3(3);    // Vecteur de strings

v1[0] = 42;
v2[0] = 3.14;
v3[0] = "Hello";
```

Types composés

```
// Vecteur de vecteurs (matrice)
Vecteur<Vecteur<double>> matrice(10);

// Attention C++98/03 : espace requis
Vecteur<Vecteur<double> > matrice(10);
// C++11+ : OK sans espace

// Initialisation
for (int i = 0; i < matrice.size(); ++i) {
    matrice[i] = Vecteur<double>(20);
}
```

Processus de compilation

1. Compilateur cherche type `Vecteur<double>`
2. Trouve `template<typename T> class Vecteur`
3. Génère code en remplaçant `T` par `double`
4. Compile le code généré
5. Cache le type pour réutilisation
6. Instancie constructeur `Vecteur<double>(int)`

Chaque type = code distinct

- `Vecteur<int>` ≠ `Vecteur<double>`
- Code généré pour chaque type utilisé
- Optimisé par le compilateur

Types complexes

```
Vecteur<pair<int, string>> v;
Vecteur<map<string, int>> v2;
Vecteur<shared_ptr<Document>> v3;
```

Opérations sur classe template

Opérateurs membres

```
template <typename T>
class Vecteur {
    T* data;
    int taille;

public:
    Vecteur& operator+=(const Vecteur& v) {
        for (int i = 0; i < taille; ++i) {
            data[i] += v.data[i];
        }
        return *this;
    }

    Vecteur operator+(const Vecteur& v) const {
        Vecteur result(taille);
        for (int i = 0; i < taille; ++i) {
            result.data[i] = data[i] + v.data[i];
        }
        return result;
    }
};
```

Contraintes implicites

Pour `operator+=` :

- Type `T` doit supporter `operator+=`

Pour `operator+` :

- Type `T` doit supporter `operator+`
- Type `T` doit être copiable

Conséquence

```
Vecteur<int> v1, v2;
v1 += v2; // OK

Vecteur<string> s1, s2;
s1 += s2; // OK (string a +=)

// Mais :
struct NoOp {};
Vecteur<NoOp> n1, n2;
n1 += n2; // ERREUR à la compilation NoOp n'a pas +=
```

Exemple STL : std::vector (1/2)

std::vector = template de classe

```cpp
#include <vector>

template <typename T>
class vector {
    T* data;
    size_t sz, cap;

public:
    void push_back(const T& value);
    T& operator[](size_t i);
    size_t size() const;
    // ... beaucoup d'autres méthodes
};
```

Utilisation

```cpp
vector<int> v;
v.push_back(10);
v.push_back(20);

vector<string> names;
names.push_back("Alice");
names.push_back("Bob");

// Même code, types différents !
```

Exemple STL : std::vector (2/2)

Algorithmes STL = templates de fonctions

```cpp
#include <algorithm>

template <typename Iterator, typename T>
Iterator find(Iterator first, Iterator last,
              const T& value);

template <typename Iterator>
void sort(Iterator first, Iterator last);

template <typename Iterator, typename Func>
void for_each(Iterator first, Iterator last,
              Func f);
```

Utilisation

```cpp
vector<int> v = {3, 1, 4, 1, 5};

// find : template de fonction
auto it = find(v.begin(), v.end(), 4);

// sort : template de fonction
sort(v.begin(), v.end());

// for_each : template de fonction
for_each(v.begin(), v.end(),
    [](int x) { cout << x << " "; });
```

Membre template dans classe template (1/2)

Fonction membre générique

```
template <typename T>
class Vecteur {
    T* data;
    int taille;

public:
    Vecteur(int n);

    template <typename U>
    Vecteur(const Vecteur<U>& other) {
        taille = other.size();
        data = new T[taille];
        for (int i = 0; i < taille; ++i) {
            data[i] = T(other[i]);
        } }

    template <typename S>
    Vecteur& operator*=(const S& scalaire) {
        for (int i = 0; i < taille; ++i) {
            data[i] *= scalaire;
        }
        return *this;
    } };
```

Utilisation

```
Vecteur<int> vi(3);
Vecteur<double> vd = vi;  // Conversion

Vecteur<double> v(10);
v *= 2;       // S = int
v *= 3.14;    // S = double
```

Membre template dans classe template (2/2)

Bibliothèque : Catalogue

```
template <typename DocType>
class Catalogue {
    vector<shared_ptr<DocType>> docs;

public:
    template <typename OtherDoc>
    void copierDepuis(
        const Catalogue<OtherDoc>& autre) {

        for (const auto& doc : autre.docs) {
            auto converted =
                dynamic_pointer_cast<DocType>(doc);
            if (converted) {
                docs.push_back(converted);
            }
        }
    }
};
```

Erreurs courantes avec les templates (1/3)

Erreur 1 : Déduction impossible

```cpp
template <typename T>
T convert(int x) { return T(x); }

double d = convert(42); // ERREUR
```

Message compilateur

```
error: no matching function for call to 'convert(int)'
note: template argument deduction/substitution failed
note: couldn't deduce template parameter 'T'
```

Solution

```cpp
double d = convert<double>(42); // OK
```

Erreur 2 : Types incompatibles

```cpp
template <typename T>
T min(const T& a, const T& b) {
    return (a < b) ? a : b;
}

int i = 5;
double d = 3.14;
auto result = min(i, d); // ERREUR
```

Message compilateur

```
error: no matching function for call to 'min(int&, double
note: deduced conflicting types for parameter 'T'
  ('int' and 'double')
```

Solutions

```cpp
min<double>(i, d);     // Spécification explicite
min(double(i), d);     // Conversion
```

Erreurs courantes avec les templates (2/3)

Erreur 3 : Contrainte implicite

```
template <typename T>
class Vecteur {
public:
    Vecteur& operator+=(const Vecteur& v) {
        for (int i = 0; i < size; ++i)
            data[i] += v.data[i]; // Requiert +=
        return *this;
    }
};

struct Point { int x, y; };
Vecteur<Point> v; // ERREUR lors de +=
```

Message compilateur

```
error: no match for 'operator+='
  (operand types are 'Point' and 'const Point')
in instantiation of 'Vecteur<T>::operator+='
```

Solution

```
struct Point {
    int x, y;
    Point& operator+=(const Point& p) {
        x += p.x; y += p.y;
        return *this;
    }
};
```

Erreurs courantes avec les templates (3/3)

Erreur 4 : Oubli de typename

```
template <typename T>
class MyClass {
    T::value_type x; // ERREUR
};
```

Message

```
error: need 'typename' before 'T::value_type'
```

Solution

```
typename T::value_type x; // OK
```

Erreur 5 : Espace >> (C++03)

```
vector<vector<int>> m; // ERREUR C++03
```

Message

```
error: '>>' should be '> >'
```

Solution

```
vector<vector<int> > m; // C++03
vector<vector<int>> m;  // C++11+ OK
```

Quel niveau d'abstraction choisir ? (1/2)

Exemple

```
template <typename T>
class Vecteur {
public:
    Vecteur& operator+=(const Vecteur& v) {
        for (int i = 0; i < taille; ++i) {
            data[i] += v.data[i]; // Requiert +=
        }
        return *this;
    }
};
```

Conséquence

- `Vecteur<char*>` ne compile plus
- `char*` n'a pas d'`operator+=`

Quel niveau d'abstraction choisir ? (2/2)

Option 1 : Restreindre

- Template pour types numériques seulement
- Documentation claire des contraintes
- Erreurs de compilation explicites

Option 2 : Concepts (C++20)

```cpp
template <typename T>
concept Addable = requires(T a, T b) {
    { a += b } -> std::same_as<T&>;
};

template <Addable T>
class Vecteur {
    // ...
};
```

La notion de **concept** repose sur des expressions ou des traits de type permettant de vérifier certaines caractéristiques des types. Dans l'exemple, le type doit supporter l'opération **+=**.

Option 3 : SFINAE / enable_if

- Techniques avancées
- Désactivation conditionnelle

```cpp
template <typename, typename = void>
struct is_addable : std::false_type {};

template <typename T>
struct is_addable<T, std::void_t<
    decltype(std::declval<T&>() += std::declval<T>())
>> : std::true_type {};

template <typename T>
std::enable_if_t<is_addable<T>::value, T>
add(T a, T b) {
    a += b;
    return a;
}
```

Implémentation séparée des templates

Fichier header (.h ou .hpp)

```
// vecteur.h
template <typename T>
class Vecteur {
    T* data;
    int taille;

public:
    Vecteur(int n);
    ~Vecteur();
    T& operator[](int i);
    Vecteur& operator+=(const Vecteur& v);
};
```

Problème classique

- Templates doivent être disponibles à la compilation
- Éditeur de liens ne suffit pas

Fichier implémentation (.txx)

```
// vecteur.txx
template <typename T>
Vecteur<T>::Vecteur(int n) : taille(n) {
    data = new T[n];
}

template <typename T>
Vecteur<T>::~Vecteur() {
    delete[] data;
}

template <typename T>
T& Vecteur<T>::operator[](int i) {
    return data[i];
}

template <typename T>
Vecteur<T>& Vecteur<T>::operator+=(
    const Vecteur<T>& v) {
    for (int i = 0; i < taille; ++i) {
        data[i] += v.data[i];
    }
    return *this;
```

Solutions pour implémentation séparée

Solution 1 : Inclusion dans header

```
// vecteur.h
template <typename T>
class Vecteur {
    // Déclaration
};

#include "vecteur.txx"
```

Avantages

- Simplicité
- Fonctionne toujours

Inconvénients

- Temps de compilation ++
- Recompilation fréquente

Solution 2 : Instanciation explicite

```
// vecteur.cpp
#include "vecteur.h"
#include "vecteur.txx"

// Instanciations explicites
template class Vecteur<int>;
template class Vecteur<double>;
template class Vecteur<string>;
```

Avantages

- Compilation rapide
- Code unique

Inconvénients

- Liste exhaustive types
- Pas flexible

Recommandation : Solution 1 (inclusion) dans la plupart des cas. Solution 2 pour grandes bibliothèques avec types connus.

Métaprogrammation Template

Calculs à la compilation

Calcul à la compilation - Factorielle

Principe

- Utiliser les templates pour calculer à la compilation
- Pas de coût à l'exécution
- Résultat dans le code compilé

Implémentation récursive

```cpp
template <int N>
struct Factorial {
    static const long value =
        N * Factorial<N - 1>::value;
};

// Spécialisation : cas de base
template <>
struct Factorial<0> {
    static const long value = 1;
};
```

Utilisation

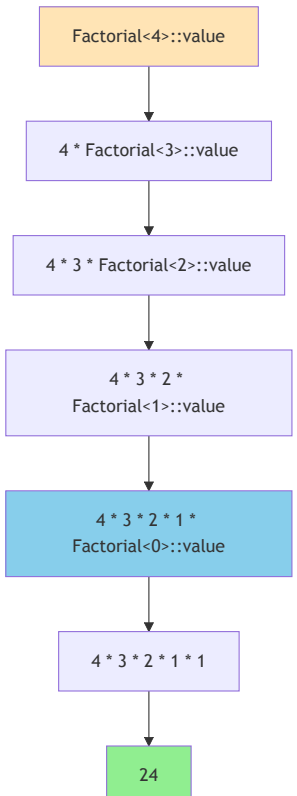
```cpp
int main() {
    // Calculé à la compilation
    long f5 = Factorial<5>::value;  // 120
    long f8 = Factorial<8>::value;  // 40320

    // Pas de calcul à l'exécution !

    // Mais :
    int n;
    cin >> n;
    long fn = Factorial<n>::value;
    // ERREUR : n doit être constant
}
```

Avantage : Pas de boucle runtime !

```cpp
// Équivalent généré par le compilateur :
long f5 = 120;    // Constante !
long f8 = 40320;  // Constante !
```

Métaprogrammation - Arbre de récursion

Expansion à la compilation



Code généré (simplifié)

```
struct Factorial_0 {
    static const long value = 1; };

struct Factorial_1 {
    static const long value =
        1 * Factorial_0::value; // = 1 };

struct Factorial_2 {
    static const long value =
        2 * Factorial_1::value; // = 2 };

struct Factorial_3 {
    static const long value =
        3 * Factorial_2::value; // = 6 };

struct Factorial_4 {
    static const long value =
        4 * Factorial_3::value; // = 24 };

long f4 = Factorial_4::value; // Puis remplace :
long f4 = 24; // Constante ! // Par :
```

Calcul à la compilation - Puissance

Puissance entière

```cpp
template <int Base, int Exp>
struct Power {
    static const long value =
        Base * Power<Base, Exp - 1>::value;
};

template <int Base>
struct Power<Base, 0> {
    static const long value = 1;
};
```

Utilisation

```cpp
long p = Power<2, 10>::value;  // 1024
long q = Power<3, 4>::value;   // 81

// Utile pour les dimensions
const int KB = Power<2, 10>::value;
const int MB = Power<2, 20>::value;
```

Optimisation

```cpp
// Version optimisée (logarithmique)
template <int Base, int Exp>
struct PowerOpt {
    static const long value =
        (Exp % 2 == 0) ?
            PowerOpt<Base * Base, Exp / 2>::value :
            Base * PowerOpt<Base, Exp - 1>::value;
};

template <int Base>
struct PowerOpt<Base, 0> {
    static const long value = 1;
};
```

Avantages métaprogrammation

- Aucun coût runtime
- Erreurs à la compilation
- Optimisations impossibles autrement

Concepts Avancés

Paramètres par défaut, Traits, Policies

Paramètres template par défaut

Principe

- Comme paramètres de fonction
- Valeurs par défaut pour types

Exemple : Point générique

```
template <typename T = double,
          std::size_t Dim = 2>
class Point {
    T coords[Dim];

public:
    Point() {
        for (size_t i = 0; i < Dim; ++i)
            coords[i] = T(); }

    T& operator[](size_t i) {
        return coords[i];  }

    size_t dimension() const {
        return Dim;  }
};
```

Utilisation

```
Point<> p1;                 // Point<double, 2>
Point<float> p2;            // Point<float, 2>
Point<double, 3> p3;        // Point<double, 3>
Point<int, 4> p4;           // Point<int, 4>

p1[0] = 3.14;  // double
p2[0] = 2.71f; // float
p3[2] = 1.0;   // 3D
p4[3] = 42;    // 4D entier

// Types identiques
Point<> a;
Point<double, 2> b;
// a et b ont le même type
```

Cas d'usage

- Simplifier API
- Cas commun par défaut
- Flexibilité préservée

Manipulation de types - typedef

Récupérer le type utilisé

```cpp
template <typename T, std::size_t N>
class Point {
public:
    typedef T value_type;
    typedef std::size_t size_type;

    // Utilisation interne
    value_type coords[N];
};
```

Extraction du type

```cpp
int main() {
    typedef Point<int, 2>::value_type VT;
    // VT est maintenant int

    VT x = 42;  // int x = 42
}
```

Mot-clé typename

```cpp
struct A {
    typedef int sign; };

struct B {
    int sign;  // Variable, pas type
};

template <typename T>
class MyClass {
public:
    // Erreur sans typename
    // T::sign signe;

    // Correct
    typename T::sign signe;
};

MyClass<A> ma;  // OK, A::sign est un type
MyClass<B> mb;  // Erreur, B::sign n'est pas un type
```

Règle : `typename` indique qu'un nom dépendant est un type

Traits - Propriétés non intrusives

Concept

- Donner des informations sur un type
- Sans modifier le type lui-même
- Pattern très utilisé en STL

Exemple : détection double précision

```
template <typename T>
struct IsDoublePrecision {
    static const bool value = false;
};

// Spécialisation pour double
template <>
struct IsDoublePrecision<double> {
    static const bool value = true;
};
```

Utilisation

```
template <typename T>
void process(T x) {
    if (IsDoublePrecision<T>::value) {
        // Code spécifique double
        cout << "Double précision\n";
    } else {
        // Code générique
        cout << "Autre type\n";} }

process(3.14);   // Double précision
process(3.14f);  // Autre type
process(42);     // Autre type
```

STL type_traits

```
#include <type_traits>

is_integral<int>::value     // true
is_floating_point<double>::value // true
is_pointer<int*>::value     // true
is_same<int, int>::value    // true
```

Traits - Exemple STL

Iterator traits

```cpp
template <typename Iterator>
struct iterator_traits {
    typedef typename Iterator::value_type
            value_type;
    typedef typename Iterator::difference_type
            difference_type;
    typedef typename Iterator::pointer
            pointer;
    typedef typename Iterator::reference
            reference;
};

// Spécialisation pour pointeurs
template <typename T>
struct iterator_traits<T*> {
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef T& reference;
};
```

Utilisation dans algorithmes STL

```cpp
template <typename Iter>
void algo(Iter first, Iter last) {
    typename iterator_traits<Iter>::value_type temp;
    typename iterator_traits<Iter>::difference_type
        distance = last - first;
    // Algorithme générique
}

vector<int> v;
algo(v.begin(), v.end());  // Iter = vector<int>::iterato

int arr[10];
algo(arr, arr + 10);       // Iter = int*
```

Avantages

- Fonctionne pour iterateurs et pointeurs
- Uniformité STL
- Code générique réutilisable

Traits STL - type_traits (C++11)

Bibliothèque standard

```cpp
#include <type_traits>

// Propriétés des types
is_integral<int>::value           // true
is_floating_point<double>::value  // true
is_pointer<int*>::value           // true
is_class<string>::value           // true

// Comparaison
is_same<int, int>::value          // true
is_same<int, long>::value         // false

// Conversions
is_convertible<int, double>::value // true

// Modification de types
remove_const<const int>::type      // int
remove_pointer<int*>::type         // int
add_pointer<int>::type             // int*
```

Utilisation avec if constexpr (C++17)

```cpp
template <typename T>
void process(T value) {
    if constexpr (is_integral<T>::value) {
        cout << "Entier: " << value << "\n"; }
    else if constexpr (is_floating_point<T>::value) {
        cout << "Flottant: " << value << "\n"; }
    else {
        cout << "Autre type\n"; } }

process(42);      // Entier
process(3.14);    // Flottant
process("hello"); // Autre type
```

SFINAE (C++11)

```cpp
template <typename T>
typename enable_if<is_integral<T>::value, T>::type
increment(T value) {
    return value + 1;
}
```

Policies - Stratégies de comportement

Concept

- Paramétrer comportement par classe policy
- Alternative au polymorphisme dynamique
- Décision à la compilation

Exemple : politique d'affichage

```
struct CartesianDisplay {
    template <typename T, int N>
    static void display(const Point<T, N>& p) {
        cout << "(";
        for (int i = 0; i < N; ++i) {
            cout << p[i];
            if (i < N - 1) cout << ", "; }
        cout << ")"; } };

struct PolarDisplay {
    template <typename T>
    static void display(const Point<T, 2>& p) {
        T r = sqrt(p[0]*p[0] + p[1]*p[1]);
        T theta = atan2(p[1], p[0]);
        cout << "(r=" << r << ", θ=" << theta << ")"; } }
```

Utilisation

```
template <typename T, int N,
          typename DisplayPolicy = CartesianDisplay>
class Point {
    T coords[N];

public:
    void display() const {
        DisplayPolicy::display(*this); } };

Point<double, 2> p1;            // Cartésien
Point<double, 2, PolarDisplay> p2;  // Polaire

p1.display(); // (3.0, 4.0)
p2.display(); // (r=5.0, θ=0.927)
```

Avantages

- Pas de virtual
- Optimisation complète
- Flexibilité maximale
- Utilisé intensivement en STL

Exemple complet : Vecteur avec Policies

Policies de gestion mémoire

```
template <typename T>
struct DefaultAlloc {
    static T* allocate(size_t n) {
        return new T[n];
    }
    static void deallocate(T* p) {
        delete[] p;
    }
};

template <typename T>
struct PoolAlloc {
    static T* allocate(size_t n) {
        // Allocation depuis pool
    }
    static void deallocate(T* p) {
        // Retour au pool
    }
};
```

Classe avec policy

```
template <typename T,
          typename Alloc = DefaultAlloc<T>>
class Vecteur {
    T* data;
    size_t sz;

public:
    Vecteur(size_t n) : sz(n) {
        data = Alloc::allocate(n);
    }

    ~Vecteur() {
        Alloc::deallocate(data);
    }
};

// Utilisation
Vecteur<int> v1(100);                  // new/delete
Vecteur<int, PoolAlloc<int>> v2(100); // pool
```

STL allocators : même principe

Applications : Bibliothèque

Conteneur générique de documents

```cpp
template <typename DocType>
class Catalogue {
    vector<shared_ptr<DocType>> documents;

public:
    void ajouter(shared_ptr<DocType> doc) {
        documents.push_back(doc);
    }

    template <typename Predicat>
    vector<shared_ptr<DocType>>
    rechercher(Predicat pred) const {
        vector<shared_ptr<DocType>> results;
        for (const auto& doc : documents) {
            if (pred(doc)) {
                results.push_back(doc);
            }
        }
        return results;
    }
};
```

Utilisation

```cpp
Catalogue<Document> cat;

auto livres = cat.rechercher( // Recherche par lambda (C+
    [](shared_ptr<Document> d) {
        return d->getType() == "Livre";
    }
);

auto cpp = cat.rechercher( // Recherche par titre
    [](shared_ptr<Document> d) {
        return d->getTitre().find("C++")
               != string::npos;
    }
);
```

Avantages

- Type-safe
- Performance
- Expressivité

Résumé

Templates de fonctions

- Généricité sur fonctions
- Déduction automatique de type
- Spécialisation possible

Templates de classes

- Conteneurs génériques
- Paramètres types et valeurs
- Membres templates

Métaprogrammation

- Calculs à compilation
- Aucun coût runtime
- Optimisations avancées

Concepts avancés

- Paramètres par défaut
- Typedef et typename
- Traits (propriétés)
- Policies (comportements)

Avantages généraux

- Type-safe
- Performance maximale
- Code réutilisable
- Expressivité

STL

- Basée sur templates
- Conteneurs, algorithmes, itérateurs
- Exemple parfait de programmation générique

La prochaine fois

STL en profondeur

Conteneurs

- vector, list, deque
- map, set, unordered_*
- Choix et performance

Itérateurs

- Catégories
- Adaptateurs
- Algorithmes génériques

Algorithmes

- sort, find, transform
- Foncteurs
- Lambdas C++11

Questions ?

N'hésitez pas à poser vos questions sur les templates et la programmation générique

Prochaine session : STL - Standard Template Library

-->