Formation C++ Ubisoft - Module 5a

Romain Arcila[1,2]
Charles de Rousiers[1]

10 mai 2009

[1] INRIA Grenoble
[2] Liris - CNRS Lyon

Plan

1. Introduction
2. Template
3. Traits, policies et constraints
4. STL - Introduction
5. STL - String
6. STL - Flux
7. STL - Conteneurs
8. STL - Algorithmes
9. STL - Foncteurs
10. STL - Autres

Objectifs

Vue globale des templates

- Syntaxe
- Concepts
- Fonctionnalités

Vue globale de la STL

- Structure de données
- Algorithmes
- Utilisation efficace

Plan

1. Introduction
2. Template
3. Traits, policies et constraints
4. STL - Introduction
5. STL - String
6. STL - Flux
7. STL - Conteneurs
8. STL - Algorithmes
9. STL - Foncteurs
10. STL - Autres

Déroulement

Matin

- Présentation des templates
- Utilisation des templates

Après-midi

- Introduction de la STL
- STL en détail

Plan

1. Introduction
2. Template
3. Traits, policies et constraints
4. STL - Introduction
5. STL - String
6. STL - Flux
7. STL - Conteneurs
8. STL - Algorithmes
9. STL - Foncteurs
10. STL - Autres

Concept

Template ?

En français : patron de conception

- Abstraction du type lors du codage
- Réutilisation de code

→ **Objectif** : généricité

Idée principale : écrire du code une fois, utilisation multiple

Concept

Exemple : Code récurrent

Pour int

```
void Swap(int& _a, int& _b)     {
        int tmp = _a;
        _a = _b;
        _b = _tmp;
}
```

Pour float

```
void Swap(float& _a, float& _b) {
        float tmp = _a;
        _a = _b;
        _b = _tmp;
}
```

Pour class A

```
void Swap(A& _a, A& _b) {
        A tmp = _a;
        _a = _b;
        _b = _tmp;
}
```

Les paramètres

Marre de réécrire toujours le même code ? Factorisation à l'aide de templates

Quelles sont les données variants ?

- Type des paramètres
- Type de la variable locale

$\longrightarrow$ Transformation des types variants en type **génériques** !

Exemple code factorisé

```
template<typename T>
void Swap(T& _a, T& _b) {
        T tmp = _a;
        _a = _b;
        _b = _tmp;
}
```

$\Rightarrow$ Écriture en template : code **indépendante du type**

Les paramètres

De quelles natures peuvent être les paramètres génériques ?

Type générique

- Class
- Struct
- Type primitif

Type constant

- Type intégral (char, wchar_t, int, long, short et leurs versions signées et non signées) ou énuméré
- Pointeur ou référence d'objet
- Pointeur ou référence de fonction
- Pointeur sur membre

$\longrightarrow$ Tous les types assimilables à une valeur entière

Les paramètres

Déclaration des types templates

Syntaxe

template <class|typename nom[=type][, class|typename nom[=type][...]>

Remarques

- Equivalence entre **class** et **typename**
- Valeur par défaut nécessaire pour tous les paramètres à droite de la première valeur par défaut
- Possibilité d'avoir des paramètres génériques

Les paramètres

Exemple : Les différents paramètres génériques

Code

```
template <class A, typename B=float>
void foo1() {
        A a;
        B b;
}

template <int i>
void foo2() {
        int b = i + 2;
}

template <class A, int i, void (*f)(int)>
void foo3() {
        A a;
        int b = i + 2;
        f(b);
}
```

Les paramètres

Exemple : Utilisation de différents type générique pour paramètre un pixel

Code

```
struct Pixel<class Storage, int Channels>
{
        Storage values[Channels];

        Pixel<Storage, Channels>& operator *=(float _factor)<
        {
                for(int i=0;i<Channels;++i)
                        values[i] *= _factor;
                return this;
        }
};
```

Syntaxe

Que peut-on templatiser ?

1. Des fonctions
2. Des classes / structures
3. Des fonctions membres

Syntaxe : fonction

Principales étapes :

- Déclaration des paramètres templates
- Définition / déclaration de la fonction

Déclaration

```
template <parametres_template>
type fonction(parametres_fonction);
```

Définition

```
template <parametres_template>
type fonction(parametres_fonction)
{
utilisation parametre
}
```

Syntaxe : fonction

Exemple : fonction template

Code

```
template<typename T>
T max(T _a, T _b);

void Transfert(int* _values1, int* _values2, int _n)
{
        for(int i=0;i<_n;++i)
                _values[i] = Max<int,int>(_values1[i],_values2[i]);
}

template<typename T>
T max(T _a, T _b)
{
        return _a>_b?_a:_b;
}
```

Syntaxe : classe

Principales étapes :

- Déclaration des paramètres templates
- Définition / déclaration de la fonction
- Pour chaque méthode définie à l'extérieur :
  - Rappeler les paramètres génériques
  - Rappeler l'instanciation de la classe à l'aide des paramètres génériques

Déclaration

```
template <parametres_template>
class | struct | union nom;
```

Définition

```
template <parametres_template>
class Classe
{
utilisation des parametres
}
```

Syntaxe : classe

Définition de méthode externes

```
template <parametres_template>
type Classe<parametres>::Nom(parametres_methode)
{
...
}
```

Remarque

- Les fonctions membres n'ont pas besoin de définir la liste des paramétres templates des types génériques de leurs classes
- On peut cependant les définir si les types sont différents
- Le destructeur d'une classe ne peut être template

Syntaxe : classe

Exemple : Classe template

Code

```
template<typename T>
class Stack;

template<typename T>
class Stack
{
public:
        void Push(const T& _item);
        T Pop();
        ...
private:
        T* elements;
        int nElements;
};

template<typename T>
void Stack<T>::Push(const T& _item) {
        elements[nElements++] = _item;
}

template<typename T>
T Stack<T>::Pop() {
        return elements[nElements--];
}

void Test() {
        Stack<int> s;
        s.Push(1);
        int a = s.Pop(1);
}
```

Syntaxe : fonction membre

Principales étapes :

- Définition de la classe
- Déclaration des paramètres templates pour la fonction
- Définition de la fonction
- Si méthode définit à l'extérieur de la classe :
  - Rappeler les paramètres génériques
  - Définir la méthode comme une méthode normale

Déclaration au sein de la classe

```
template <parametres_template>
type Nom(parametres_methode);
```

Définition de méthode externes

```
template <parametres_template>
type Classe::Nom(parametres_methode)
{
...
}
```

Syntaxe : classe

Exemple : Fonction membre template

Code

```
struct A
{
        void Foo1;

    template <class T>
    void Foo2(T _value);
};

void A::Foo1(){
        ...
}

template <class T>
void A::Foo2(T _value) {
        ...
}

void Test() {
        A a;
        a.Foo1();
        a.Foo1<float>(1.26f);
}
```

Syntaxe : fonction membre

Fonction membre template dans une classe template ?

$\longrightarrow$ Mixte entre les deux syntaxe précédentes

Déclaration au sein de la classe

```
template <parametres_template>
type Nom(parametres_methode);
```

Définition de méthode externes

```
template <parametres_fonction>
template <parametres_class>
type Classe<parametres_class>::Nom(parametres_methode)
{
...
}
```

Syntaxe : classe

Exemple : Fonction membre template dans une classe template

Code

```
template<class X>
struct A
{
        void Foo1;

    template <class T>
    void Foo2(T _value);
};

template<class X>
void A<X>::Foo1(){
        ...
}

template <class T>
template <class X>
void A<X>::Foo2(T _value) {
        ...
}

void Test() {
        A<int> a;
        a.Foo1();
        a.Foo1<float>(1.26f);
}
```

Syntaxe

Possibilité d'avoir des paramètres de template génériques

Syntaxe

```
template< class T, template<class X> class U >
```

Exemple : Utilisation pour un code indépendant de la structure de donnée

Code

```
template <class T>
class Tableau {
        ...
};

template <class U, class V, template <class T> class C=Tableau>
class Dictionnaire {
    C<U> Clef;
    C<V> Valeur;
};
```

Instanciation

Instanciation

Utilisation d'une classe ou une fonction template pour un jeu de paramètres effectifs

Deux types d'instanciation possibles

- Instanciation **implicite**
- Instanciation **explicite**

Remarque

Si aucune instanciation d'une classe ou d'une fonction, code de la classe ou de la fonction ne sera jamais :

- Compilé (Sauf vérification syntaxique)
- Intégré dans l'exécutable

$\longrightarrow$ Vérification du typage à compilation (typage fort)

Instanciation : Implicite

Implicite

Utilisation du contexte courant pour déterminer les paramètres template à utiliser (paramètres effectifs)
$\longrightarrow$ Si aucune ambiguïté : génération du code

Exemple : Instanciation d'une fonction

Code

```
int i=Max(2,3); // OK
int i=Max(2,3.0); // Ambiguite Min<int> ou Min<double>
```

Instanciation : Implicite

Résolution de l'ambiguïté

Pour lever l'ambiguité préciser manuellement le type des paramètres à utiliser

Exemple : Lever d'ambiguïté lors d'une instanciation

Code

```
int i=Max<int>(2,3.0); //Ici conversion de 3.0 en entier
```

Instanciation : Implicite

Exemple : Instanciation d'une classe template, avec une des méthodes non instanciée

Code

```
template <class T>
class A {
public:
    void f(void);
    void g(void);
};

template <class T>
void A<T>::f(void) {
    cout << "A<T>::f() appelee" << endl;
}

int main(void) {
    A<char> a;
    a.f();
    return 0;
}
```

Instanciation : Explicite

Explicite

Instanciation du template **forcée** par l'utilisateur
→ Oblige le compilateur à générer le code même si celui-ci n'est pas utilisé

Syntaxe

- Pour une classe
  ```
  template Stack<int>
  ```
- Pour une fonction
  - Si détermination possible pas le compilateur à partir des paramètres :
    ```
    template int Max(int, int);
    ```
  - Si paramètres fournis directement pas l'utilisateur
    ```
    template int Max<int>(int,int);
    ```
  - Si tous les paramètres ont une valeur par défaut
    ```
    template int Max<>(int,int);
    ```

Instanciation

Problème

Nécessite une définition complète du template pour l'instanciation

- Besoin de la déclaration
- Besoin de la définition

Trois conséquences directes :

1. Classes et fonctions template non considérées comme des fonctions et classes normales
   → Séparation non possible des déclarations et définitions
2. Compilations multiples des instances des templates
   → Ralenti la compilation
3. Instances de templates présentent en multiples exemplaires dans les objets générés
   → Accroît la taille de l'exécutable

Instanciation

Solution

1. Utilisation de fichiers complémentaires pour la définition des template (headers étendus : .inl, .tpp, ...)
2. Utilisation de l'exportation de templates au travers d'entêtes précompilés (améliore la vitesse de compilation)
3. Peut être gérée par l'éditeur des liens qui répertorie les codes commums pour regrouper les définitions

Autre solution

- Désactivation de l'instanciation implicite
- Regroupement dans une fichier de toutes les instanciations explicites

→ Manière la plus sûre et la plus recommandée

Instanciation

Localisation du code

Problème

- Instanciation nécessite une définition complète du template
- Séparation déclaration / définition non possible en .hpp / .cpp

Solution

Utilisation de headers étendus (include de .inl, .tpp, ...)

Instanciation

Fichier .hpp

```
void Foo();

template<typename T>
void GenericFoo();

template<typename T>
class A {
        void MemFoo();
};

#include "File.inl"
```

Fichier .inl

```
template<typename T>
void GenericFoo() {
        ...
}

template<typename T>
void A<T>::MemFoo() {
        ...
}
```

Instanciation

Fichier .cpp

```
void Foo() {
        ...
}
```

Instanciation

Instanciation réalisée lors de la phase de compilation

Conséquences

- Tous les paramètres doivent être connus à la phase de compilation
- Pas possible de laisser la détermination à l'exécution

Exemple : Code non valide car nécessite une résolution à l'exécution

Code

```
template<typename T, int numberOfElement>
struct Array
{
        T Values[n];
};

void foo(int _n)
{
        Array<char,_n> tab;
        tab.Values[0] = 'c';
}
```

Instanciation

Export de fonction

Permet de séparer la déclaration de la définition d'une template

Fichier .hpp

```
export template<typename T>
int Foo(int, int);
```

Fichier .cpp

```
export template<typename T>
int Foo(int, int) { }
```

Attention

- Si une fonction est déclarée comme inline, ne peut pas être export
- **Ne marche sur aucun compilateur !**

Instanciation

Ordre d'appel

Si une fonction template peut être instanciée avec le même prototype qu'une autre fonction non template

1. Appel en premier de la fonction non template
2. Sinon obligation de donner les paramètres pour assurer l'appel de la fonction template

Instanciation

Exemple : Ordre d'appel

Code

```
#include <iostream>

using namespace std;

struct A {
    void f(int);

    template <class T>
    void f(T){
        cout << "Template" << endl;
    }
};

// Fonction non template :
void A::f(int) {
    cout << "Non_template" << endl;
}

// Fonction template :
template <>
void A::f<int>(int) {
    cout << "Specialisation_f<int>" << endl;
}

int main(void) {
    A a;
    a.f(1);    // Appel de la version non-template de f.
    a.f('c'); // Appel de la version template de f.
    a.f<>(1); // Appel de la version template specialisee de f.
    return 0;
}
```

Template et surcharge

Surcharge ?

Définition des plusieurs fonctions ayant le même nom mais une signature différente

Les fonctions templates peuvent être surchargées comme toutes autres fonctions

- Par des **fonctions normales**
- Par d'autre **fonctions templates**

Exemple : Surcharge d'une fonction normale

Exemple

```
void Foo(int a) { ... }

template<typename T>
void Foo(int a, T a) { ... }
```

Template et surcharge

Exemple : Surcharge d'une fonction template

Exemple

```
template<typename T>
void Foo(int a) { ... }

template<typename T>
void Foo(int a, T a) { ... }
```

Template et amitié

Amitié ?

Une relation d'amitié permet l'accès aux données privées d'une classe ou d'une structure

Une fonction template peut être déclarée amie

- D'une **classe normale**
- D'une **classe template**

Remarque

Si une classe définit une relation d'amitié avec une fonction template, toutes les instanciations possèderont cette relation d'amitié

Template et amitié

Exemple : Différents types de relation d'amitié

Exemple

```
template <class T>
class A {
        friend void Function1();
        friend void Function2(const A<T>&);

        template <typename U>
        friend void Function5(const A<T>&, const U &);
};
```

Template et héritage

Attention

Les fonctions templates ne peuvent pas être déclarées virtuelles
$\longrightarrow$ Pas possible de faire du polymorphisme sur des fonctions template !

Astuce

- Définir une méthode non template virtuelle
- Appeler la fonction template depuis la fonction virtuelle

Code

```
class B
{
    virtual void f(int);
};

class D : public B
{
    template <class T>
    void f(T);      // Cette fonction ne redefinit pas B::f(int).

    void f(int i) { // Cette fonction surcharge B::f(int).
        f<>(i);     // Elle appelle de la fonction template.
    }
};
```

Spécialisation

Spécialisation

Modification du comportement du template pour un certain jeu de paramètres donnés

Deux types de spécialisation :

- **Totale** : tous les paramètres sont spécialisés
- **Partielle** : seuls quelques paramètres sont spécialisés

Syntaxe

- Laisser entre <> uniquement les paramètres restant template
- Préciser au niveau de la classe ou de la fonction les paramètres d'instanciation

Spécialisation

Exemple : Spécialisation totale

Code

```
template <class T1, class T2, int I>
class A
{
};

template <>
class A<int, float, 5>
{

};
```

Spécialisation

Exemple : Spécialisation partielle

Code

```
template <class T1, class T2, int I>
class A
{
};

template <class T, int I>
class A<T, T*, I>
{
};

template <class T1, class T2, int I>
class A<int, T2, I>
{
};
```

Spécialisation

Remarques

- Nombre de paramètres templates devient variable
- Nombre de valeurs fournies pour la spécialisation est toujours constant

Une valeur ne peut pas être exprimée en fonction d'une paramètre template de spécialisation

Exemple

```
template <int I, int J>
struct B
{
};

template <int I>
struct B<I, I*2> // Erreur !
{                // Specialisation incorrecte !
};
```

Spécialisation

Attention

- Pas possible de faire une spécialisation partielle avec des dépendances entre les paramètres
- Pas de valeur par défaut dans les spécialiations
- Pas possible de laisser tous les paramètres identiques dans une spécialisation

Exemple : dépendance des paramètres non valide

```
template <class T, T t>
struct C
{
};

template <class T>
struct C<T, 1>;  // Erreur !
                 // Specialisation incorrecte !
```

Spécialisation

Que peut-on spécialiser ?

- Les classes
- Les fonctions membres de classes
- Les fonctions membres de classes template spécialisées

Qu'est qui est interdit ?

- Spécialisation de fonctions
- Spécialisation de fonctions membres d'une classe template non spécialisée

Typename

typename

Utilisation dans deux cas de figure :

- Définition de type générique
- Indication de manipulation de type générique

Exemple : Utilisation du typename

Code

```
class A
{
        public: typedef int Y;     // Y est un type defini dans la classe A.
};

template <class T>
class X
{
        typename T::Y i;  // La classe template X suppose que le
                          // type generique T definisse un type Y.
};
```

Typename

class ou *typename* ?

- Utilisation indifférente
- *typename* quand possibilité d'utiliser des types primitifs
- *class* dans le cas contraire

Toujours possibilité de remplacer *class* par *typename* mais pas l'inverse

Typedef

Manipulation de nom long et complexe avec les templates
→ *Bonnepratique* : utilisation des *typedef*

Code : à éviter

```
std::map<int,std::list<std::vector<std::string > > > collection;
std::map<int,std::list<std::vector<std::string > > >::iterator it;
std::list<std::vector<std::string > obj = it->second;
```

Code : à préférer

```
typedef std::vector<std::string> TStringVector;
typedef std::list<TStringVector> TStringVecList;
typedef std::map<int,TStringVecList> TCollection;

TCollection collection;
TCollection::iterator it;
TStringVecList obj = it->second;
```

Permet d'améliorer dans le code :

- La lisibilité
- La maintenance

Typedef

Parfois souhait d'avoir des *typedef* générique

$\longrightarrow$ Pas possible : introduction dans la prochaine norme du C++

Exemple

```
typedef template<typename T> std::map<T,std::vector<T> > Collection;

Collection<T> collection;
```

Astuce : Encapsulation dans une *struct*

Exemple

```
template<typename T>
struct Collection
{
typedef std::map<T,std::vector<T> > Type;
};

Collection<T>::Type collection;
```

Template keyword

Attention

L'appel de la fonction doit être précédée de *template* si le nom d'une fonction membre template apparaît après :

- .
- $->$
- ::

Si oublie du mot-clé, interprétation comme l'opérateur $<$

Pourquoi ?

- Permet de faciliter l'étape de parsing.
- Sinon le compilateur ne peut pas savoir qu'il s'agit d'une fonction membre template

Template keyword

Exemple

```
class X {
public:
        template <int j> struct S {
                void h() {
                        cout << "member template's member function: " << j << endl;
                }
        };

        template <int i> void f() {
                cout << "Primary: " << i << endl;
        }
};

template<> void X::f<20>() {
        cout << "Specialized, non-type argument = 20" << endl;
}

template<class T> void g(T* p) {
        p->template f<100>();
        p->template f<20>();
        typename T::template S<40> s;
        s.h();
}
```

Plan

1. Introduction
2. Template
3. Traits, policies et constraints
4. STL - Introduction
5. STL - String
6. STL - Flux
7. STL - Conteneurs
8. STL - Algorithmes
9. STL - Foncteurs
10. STL - Autres

Introduction

Contexte

- Template permet d'écrire du code générique
- Mais permet aussi de mettre en place des astuces pour faciliter l'écriture de code

Présentation de trois **idioms** utiles pour la rédaction de template

- Traits
- Policies
- Constaints

Traits

Traits

Permet de rattacher des informations à un type donné
$\longrightarrow$ Connaissance à la compilation de **données supplémentaire sur le type**

Principe

Utilisation de la spécialisation des templates

- Définition de la structure d'infromation
- Spécialisation des méthodes pour les types souhaités

Traits

Exemple : Obtenir les bornes maximales d'un type numérique

Code

```
template<typename T>
struct Limits {
        T Max();
        T Min();
};

template<>
struct Limits<int> {
        T Max() { return 1<<31-1; }
        T Min() { return -1<<31; }
};

struct Limits<short> {
        T Max() { return 1<<15-1; }
        T Min() { return -1<<15; }
};
```

Traits

Ajout d'informations sous la forme :

- De fonctions
- De données membres
- De type (avec des *typedef*s)

Exemple : Détection si un type est de type *void*

title

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

template<>
struct is_void< void >{
        static const bool value = true;
};
```

Traits

Exemple : Character traits

Code

```
template <class charT>
struct CharTraits { };

struct CharTraits<char> {
        typedef char charType;
        typedef int  intType;
        static inline intType eof() { return EOF; }
};
```

Traits

Type constant

Possibilité également de rattacher des informations à un type constant

- Valeur entière (int, short, ...)
- Énumération
- ...

Traits

Exemple : Utilisation avec une énumération

Code

```
enum PixelType { PIXEL_RGBA8, PIXEL_RGBA16, PIXEL_DEPTH24 };

template<PixelType T>
struct PixelTraits {
    typedef int Type;
    static int Channels() ;
    static size_t Bytes() ;
    static FormatType Format();
};

template<PixelType type>
void Foo() {
        PixelTraits<type>::Type* pixel = image.GetPixel(0,0);
        int nChannels = PixelTraits<Type>::Channels();
        for (int i=0; i<nChannels; ++i)
                pixel[i] = // Do something...
};

template<PIXEL_RGBA8>
struct PixelTraits {
   typedef char Type;
    static int Channels() { return 4; }
    static size_t Bytes() { return sizeof(Type); }
    static FormatType Format() {return FORMAT_RGBA; }
};
```

Policy

Contexte

Template : concevoir du code réutilisable → Mais pas toujours évident !

- Actions non communes
- Contraintes supplémentaires
- Besoins en performance différents

Solution

- Avoir une base générique
- Avoir des comportements personnalisés

Adaptation du comportement avec les comportements personnalisés

Policy

Exemple : SmartPointer Que faire lorsque l'opérateur de déférencement est invoqué et que le pointeur est NULL

- Lever une exception ?
- Faire un assert ?
- Ne rien faire ?

$\Rightarrow$ Dépend de l'application, du contexte, ...

Solution

- Utilisation des policies pour gérer les différents comportements
- Éviter de coder 3 smarts pointeurs très peu différents
- Repousse le choix à son utilisation concrète (choix plus approprié)

Utilisation du comportement spécialisé

- Par héritage
- Par utilisation directe

Policy

Exemple : Politique de déférencement d'un smart pointer

Code

```
template<class T>
struct NullCheck{
        static T* CheckObject(const T* _obj) {
                return _obj;
        }
};

template<class T>
struct AssertCheck {
        static T* CheckObject(const T* _obj) {
                assert(_obj != NULL)
                return _obj;
        }
};

template<class T>
struct ExceptionCheck {
        static T* CheckObject(const T* _obj)
        {
                if(_obj == NULL) throw NullObject();
                return _obj;
        }
};
```

Policy

Code

```
template<class T, template class CheckPolicy<T> >
class SmartPointer {
public :
        T*operator*() {
                return CheckPolicy<T>::CheckObject(obj);
        }
private :
        T* obj;
};

template<class T, template class CheckPolicy<T> >
class SmartPointer : protected CheckPolicy<T> {
public :
        T*operator*() {
                return CheckObject(obj);
        }
private :
        T* obj;

};
```

Policy

Code

```
int main() {
        SmartPointer<B,NullCheck> myNullPtr;
        B b1 = *myPointer;

        SmartPointer<B, AssertCheck> myAssertPtr;
        B b2 = *myAssertPtr;

        SmartPointer<B, ExceptionCheck> myExptPtr;
        B b3 = *myExptPtr;

        return 0;
}
```

Policy

Autre exemple : Accumulation de valeurs

Code

```
template <typename T>
struct Addition {
    static void Accumuler(T& Resultat, const T& Valeur) {
        Resultat += Valeur;
    }
};

template <typename T, typename Operation>
T Accumulation(const T* Debut, const T* Fin) {
    T Resultat = 0;
    for ( ; Debut != Fin; ++Debut)
        Operation::Accumuler(Resultat, *Debut);
    return Resultat;
}
```

Constraint

Contexte

Erreur dans les templates

- Peu lisible
- Très longue

Les types générique nécessitent parfois d'avoir :

- Des constucteurs
- Des méthodes
- Des données membres
- Des opérateurs

→ Possibilité de vérifier la présence de ces données pour faciliter le debuggage

Solution

Vérification au plus tôt des éléments nécessaires :

1. Mettre tous les éléments requis dans une fonction membre
2. Référencer cette méthode dans un constructeur ou dans le destructeur

Constraint

Exemple : Vérification sur un container

Code

```
template<class A>
struct SortedContainer {
        static void constraints() {
                // A must be default-constructible
                A a, b;

                // A must be comparable
                if( a < b ) { }

                // A must have a const member function named index returning an int
                int (A::*index)() const = &A::index;

                // you could also say : int i = a.index(); but it is less precise
        }

        ~SortedContainer() {
                // trigger constraints checks
                void (&c)() = constraints;
        }

        // implementation ...
};

SortedContainer<int> foo; // error ! no index member function...
```

Plan

1. Introduction
2. Template
3. Traits, policies et constraints
4. STL - Introduction
5. STL - String
6. STL - Flux
7. STL - Conteneurs
8. STL - Algorithmes
9. STL - Foncteurs
10. STL - Autres

Présentation

STL : Standard Template Library (bibliothèque générique standard)

- Ensemble de librairies
- Défini par la norme ISO
- Disponible dans chaque implémentation du langage C++.
- Utilisation importante des templates pour avoir une forte généricité (paramétrisation importante et facile)

Contenu

Principaux éléments :

- Chaînes de caractères
- Types numériques (complexes, ...)
- Utilitaires généraux (objets fonctions et les auto pointeurs)
- Gestion flux d'entrée sortie.
- Support du langage comme le RTTI.
- Conteneurs pour stocker et manipuler des collections d'objets.
- Itérateurs pour travailler sur des collections d'objets.
- Algorithmes de consultation et de manipulation de collections d'objets.

Namespace

Namespace

- Tous les éléments de la STL utilise le namespace **std : :**
  Sauf opérateur *new* et *delete*
- Collision de nom faible : utilisation de la directive **using namespace**

Plan

1. Introduction
2. Template
3. Traits, policies et constraints
4. STL - Introduction
5. STL - String
6. STL - Flux
7. STL - Conteneurs
8. STL - Algorithmes
9. STL - Foncteurs
10. STL - Autres

Introduction

Classe template la gestion de chaînes de caractères

string

Prototype : *basic_string* < *charT* , *traits*, *Alloc* >

- **charT** : le type de caractère manipulé
- **traits** : informations ratachéau type de caractère
- **alloc** : allocateur pour la gestion interne de la mémoire

Par défaut **char_traits** : déclare des opérations de bas niveau sur les caractères :

- Comparaison
- Recherche
- Déplacement
- Copie
- etc...

En-tête < *string* >

Présentation

Spécialisations de *char_traits*

- **char_traits<char>** : pour les caractères standards
- **char_traits<wchar_t>** : pour les caractères unicode

Instanciation de chaînes de charactères

- *typedef basic_string < char > string*
- *typedef basic_string < wchar_t > wstring*

Chaîne c-like

Chaine c-like

- Issue du C
- Tableau de caractères terminé par le caractère \0

Fichiers en-tête concernés par la gestion de chaînes de caractères c-like :

- < *cstdlib* >
- < *cstring* >
- < *cctype* >
- < *cwtype* >
- < *cwchar* >

Remarque

Préférez l'utilisation des *string* aux chaînes *char**

- Simple à utiliser
- Sûr
- Complet
- Performant

Fonctionnalités

Description des fonctionnalités offertes par la classe *string*

explicit string()
Construction par défaut (chaîne vide)

string(const char s, size_t n)
Construction avec les n premiers éléments d'un tableau de caractères (par défaut : tous les caractères)

string(size_t n, char c)
Construction par répétition d'un caractère

string(const string &str, size_t pos = 0, size_t n = npos)
Construction avec n caractères d'une autre chaîne (par défaut : jusqu'au bout) à partir du caractère de rang pos (par défaut : le premier)

Fonctionnalités

template<class inIter> string(inIter begin, inIter end)
Construction avec les caractères fournis par une itération (au sens des itérateurs de la STL)

string()
Destruction

string &operator=(const string &str)
string &operator=(const char *s)
string &operator=(char c)
Affectation

Fonctionnalités

itérateur begin()
itérateur_constant begin() const
itérateur end()
itérateur_constant end() const
itérateur_inverse rbegin()
itérateur_inverse_constant rbegin() const
itérateur_inverse rend()
itérateur_inverse_constant rend() const
Iterateurs

size_t size() const
size_t length() const
Nombre de caractères (ces deux fonctions sont les mêmes)

size_t max_size() const
Nombre maximum de caractères

Fonctionnalités

void resize(size_t n, char c)
void resize(size_t n)
Changement du nombre de caractères : l'allongement se fait par recopie de c ou, par défaut \0

const char &operator[](size_t pos) const
char &operator[](size_t pos)
const char &at(size_t n) const
char &at(size_t n)
Accès au ième caractère (reference est « char & », const_reference est « const char & ») :

Fonctionnalités

string &operator+=(const string &str)
string &operator+=(const char *s)
string &operator+=(char c)
string &append(const string &str)
string &append(const string &str, size_t pos, size_t n)
string &append(const char *s, size_t n)
string &append(const char *s)
string &append(size_t n, char c)
template¡class inIter¿ string &append(inIter first, inIter last)
void push_back(const char)
Concaténation avec une chaîne, un char * , un char ou le produit d'une itération

string &assign(const string&)
string &assign(const string &str, size_t pos, size_t n)
string &assign(const char *s, size_t n)
string &assign(const char *s)
string &assign(size_t n, char c)
template¡class inIter¿ string &assign(inIter first, inIter last)
Modification des caractères d'une chaîne

Fonctionnalités

Exemple : append / operator +=

Code

```
string name ("John");
string family ("Smith");
name += " K. ";          // c-string
name += family;          // string
name += '\n';            // character
```

Fonctionnalités

string &insert(size_t pos1, const string &str)
string &insert(size_t pos1, const string &str,_t pos2, size_t n)
string &insert(size_t pos, const char *s, size_t n)
string &insert(size_t pos, const char *s)
string &insert(size_t pos, size_t n, char c)
itérateur insert(itérateur p, char c)
void insert(itérateur p, size_t n, char c)
template¡class inIter¿ void insert(itérateur p, inIter first, inIter last)
Insertion parmi les caractères d'une chaîne

string &erase(size_t pos = 0, size_t n = npos)
itérateur erase(itérateur position)
itérateur erase(itérateur first, itérateur last)
Suppression de caractères au milieu d'une chaîne

Fonctionnalités

Exemple : insert

Code

```
string str="to_be_question";
string str2="the_";
string str3="or_not_to_be";
string::iterator it;

// used in the same order as described above:
str.insert(6,str2);                 // to be (the )question
str.insert(6,str3,3,4);             // to be (not )the question
str.insert(10,"that_is_cool",8);    // to be not (that is )the question
str.insert(10,"to_be_");            // to be not (to be )that is the question
str.insert(15,1,':');               // to be not to be(:) that is the question
it = str.insert(str.begin()+5,','); // to be(,) not to be: that is the question
str.insert (str.end(),3,'.');       // to be, not to be: that is the question (...)
str.insert (it+2,str3.begin(),str3.begin()+3); // (or )
```

Fonctionnalités

string &replace(size_t pos1, size_t n1, const string &str)
string &replace(size_t pos1, size_t n1, const string &str, size_t pos2, size_t n2)
string &replace(size_t pos, size_t n1, const char *s, size_t n2)
string &replace(size_t pos, size_t n1, const char *s)
string &replace(size_t pos, size_t n1, size_t n2, char c)
string &replace(itérateur i1, itérateur i2, const string &str)
string &replace(itérateur i1, itérateur i2, const char *s, size_t n)
string &replace(itérateur i1, itérateur i2, const char *s)
string &replace(itérateur i1, itérateur i2, size_t n, char c)
template<class inIter> string &replace(itérateur i1, itérateur i2, inIter j1, inIter j2)
size_t copy(char *s, size_t n, size_t pos = 0) const
void swap(string)
Remplacement de caractères d'une chaîne

const char c_str() const
const char data() const
Obtention explicite du char * sous-jacent. Dans le cas de c_str, un caractère nul

Fonctionnalités

size_t find (const string &str, size_t pos = 0) const
size_t find (const char *s, size_t pos, size_t n) const
size_t find (const char *s, size_t pos = 0) const
size_t find (char c, size_t pos = 0) const
Recherche de chaînes et de caractères dans une chaîne. Fonctions donnant la position la plus à gauche (« première occurrence »)

size_t rfind(const string &str, size_t pos = npos) const
size_t rfind(const char *s, size_t pos, size_t n) const
size_t rfind(const char *s, size_t pos = npos) const
size_t rfind(char c, size_t pos = npos) const
Fonctions donnant la position la plus à droite (« dernière occurrence »)

size_t find_first_of(const string &str, size_t pos = 0) const
size_t find_first_of(const char *s, size_t pos, size_t n) const
size_t find_first_of(const char *s, size_t pos = 0) const
size_t find_first_of(char c, size_t pos = 0) const
Première occurrence d'un caractère d'un ensemble

Fonctionnalités

Exemple : find

Code

```
string str ("There are two needles in this haystack with needles.");
string str2 ("needle");
size_t found;

// different member versions of find in the same order as above:
found=str.find(str2);
if (found!=string::npos)
cout << "first 'needle' found at: " << int(found) << endl;

found=str.find("needles are small",found+1,6);
if (found!=string::npos)
cout << "second 'needle' found at: " << int(found) << endl;

found=str.find("haystack");
if (found!=string::npos)
cout << "'haystack' also found at: " << int(found) << endl;

found=str.find('.');
if (found!=string::npos)
cout << "Period found at: " << int(found) << endl;

// let's replace the first needle:
str.replace(str.find(str2),str2.length(),"preposition");
cout << str << endl;
```

Fonctionnalités

size_t find_last_of (const string &str, size_t pos = npos) const
size_t find_last_of (const char *s, size_t pos, size_t n) const
size_t find_last_of (const char *s, size_t pos = npos) const
size_t find_last_of (char c, size_t pos = npos) const
Dernière occurrence d'un caractère d'un ensemble

size_t find_first_not_of(const string &str, size_t pos = 0) const
size_t find_first_not_of(const char *s, size_t pos, size_t n) const
size_t find_first_not_of(const char *s, size_t pos = 0) const
size_t find_first_not_of(char c, size_t pos = 0) const
Première occurrence d'un caractère qui n'est pas dans un ensemble

Fonctionnalités

size_t find_last_not_of (const string &str, size_t pos = npos) const
size_t find_last_not_of (const char *s, size_t pos, size_t n) const
size_t find_last_not_of (const char *s, size_t pos = npos) const
size_t find_last_not_of (char c, size_t pos = npos) const
Dernière occurrence d'un caractère qui n'est pas dans un ensemble

int compare(const string &str) const
int compare(size_t pos1, size_t n1, const string &str) const
int compare(size_t pos1, size_t n1, const string &str, size_t pos2, size_t n2) const
int compare(const char *s) const
int compare(size_t pos1, size_t n1, const char *s, size_t n2 = npos) const
Comparaison de chaînes

Fonctionnalités

Exemple : Compare

Code

```
string str1 ("green apple");
string str2 ("red apple");

if (str1.compare(str2) != 0)
cout << str1 << " is not " << str2 << "\n";

if (str1.compare(6,5,"apple") == 0)
cout << "still, " << str1 << " is an apple\n";

if (str2.compare(str2.size()-5,5,"apple") == 0)
cout << "and " << str2 << " is also an apple\n";

if (str1.compare(6,5,str2,4,5) == 0)
cout << "therefore, both are apples\n";
```

Fonctionnalités

Exemple : Utilisation de *substr*

Code

```
string str="We think in generalities, but we live in details .";
                    // quoting Alfred N. Whitehead
string str2, str3;
size_t pos;

str2 = str.substr (12,12); // " generalities "

pos = str.find("live");     // position of "live" in str
str3 = str.substr (pos);   // get from "live" to the end

cout << str2 << ' ' << str3 << endl;

return 0;
```

Conversion String en C-like

Remarque

Pas de conversion implicite d'une string en *char**
$\rightarrow$ **Utiliser la fonction membre** *c_str*

C-like vs String

Exemple : Construction d'un chemin d'accès à un fichier(version char *) :

Code

```
#include <string.h>
void main()
{
        char tmp[80];
        char * Lecteur = "c:/";
        char * Path = "home/mesdocuments/";
        char * NomFichier = "Fichier.txt";
        strcpy( tmp , Lecteur );
        strcat( tmp , Path );
        strcat( tmp , NomFichier );
        // fopen( tmp ,...
}
```

Exemple : Construction d'un chemin d'accès à un fichier (version string) :

Code

```
#include <string>
using namespace std;
void main()
{
        string Lecteur = "c:/";
        string Path = "home/mesdocuments/";
        string NomFichier = "Fichier.txt";
        string tmp = Lecteur + Path + NomFichier;
}
```

C-like vs String

Exemple : Attribut d'une classe de type chaîne de caractères dynamique (version char *) :

Code

```
#include <string>
class CPersonne {
        char * m_Nom;
        void CopyNom( const char * Nom )        {
                m_Nom = new char[strlen(Nom)+1];
                strcpy( m_Nom , Nom );
        }
public:
        CPersonne( const char * Nom ) {
                CopyNom( Nom );
        }
        //une variable allouee dynamiquement implique la regle des 3:
        // le destructeur,
        // le constructeur de copie,
        // l'operateur d'affectation
        ~CPersonne() {
                delete [] m_Nom;
        }
        CPersonne( const CPersonne & tmp )      {
                CopyNom( tmp.m_Nom );
        }
        operator=( const CPersonne & tmp )      {
                if( &tmp != this )      {
                        delete [] m_Nom;
                        CopyNom( tmp.m_Nom );
                }
        }
};
```

C-like vs String

Exemple : Attribut d'une classe type chaîne de caractères dynamique (version string) :

Code

```
#include <string>
using namespace std;
void main()
{
        string Lecteur = "c:/";
        string Path = "home/mesdocuments/";
        string NomFichier = "Fichier.txt";
        string tmp = Lecteur + Path + NomFichier;
}
```

Plan

1. Introduction
2. Template
3. Traits, policies et constraints
4. STL - Introduction
5. STL - String
6. STL - Flux
7. STL - Conteneurs
8. STL - Algorithmes
9. STL - Foncteurs
10. STL - Autres

Introduction

Flux

Canal sur lequel on peut écrire ou lire des données

Manipulation par les opérateurs de flux :

- **operator** >> : Pour l'extraction de données depuis le flux
  stream operator << (*stream*, *data*)
- **operator** << : Pour l'insertion de données dans le flux
  stream operator >> (*stream*, *data*)

Chaînage des opérateurs sur le flux

```
stream out;
out << data0 << data1 << data2;
```

Classes de manipulation des flux standards

- **istream** : flux en lecture seule
- **ostream** : flux en écriture seule
- **iostream** : flux en lecture et en écriture (hérite de istream et ostream)

Entête : < *istream* >, < *ostream* >

ostream

ostream : Classe de flux de sortie,

Principales méthodes

- **put(char c)**
  Écrit un caractère c sur le flot, renvoie le flot
- **write(char* c, int l)**
  Écrit l caractères sur le flot, depuis l'adresse c (pas de formatage), et renvoie le flot.
  Pas besoin de formatage, donc utilisation possible pour les fichiers binaires.
- << **expression**
  Formate et écrit l'expression sur le flot (types de base automatiquement reconnus), renvoie le flot

Exemple : Utilisation du flux de sortie standard

Code

```
cout << " Total : " << 123.67 << " euros."; cout.put('h').put('e').put('l').put('l').put('o');
cout.write("hello",5).write(" ca va? ",8);
```

istream

istream : Classe de flux d'entrée

Principales méthodes

- **>> variable**
  Extrait les caractères du flot en accord avec variable réceptrice, renvoie le flot
- **get(char c)**
  Lit un caractère sur le flot (quel qu'il soit), et stocke dans c, renvoie le flot
- **get()**
  Extrait un caractère du flot et renvoie EOF si la fin du flot est atteinte.
- **getline(char c, int l, char delim='\ n')**
  Lit une chaîne d'au plus l caractères (si delim pas rencontré, limitée à delim sinon)
  Ajout de '\ 0' en fin de chaîne
- **gcount()**
  Renvoie le nombre de caractères lus lors de *getline*()
- **Read(char* c, int l)**
  Lit "l" caractères sur le flot et les range à l'adresse "c".
  Pas de formatage de donner donc utile pour les fichiers binaires

istream

Exemple : Utilisation du flux d'entrée

Code

```
int a ; double d ; cout ¡¡ "Entrez un entier puis un double : " ; cin ¿¿ a ¿¿ d ;
```

Flux standards

Flux standards en C++

- **istream cin** : Entree standard (clavier par defaut)
- **ostream cout** : Sortie standard (terminal par defaut)
- **ostream cerr** : Canal d'erreur (terminal par defaut)
- **ostream clog** : Canal d'erreur buffeurisé

Manipulation à l'aide des opérateurs de flux : operator<< et operator>>
En-tête : < *iostream* >

Caractéristiques

- Ouverture au lancement du programme
- Contrôlent les insertions et les extractions dans stdin, stdout et stderr

Fichier

Classes pour la manipulation de fichier

- **ifstream** : flux pour la lecture de fichier
- **ofstream** : flux pour l'écriture de fichier
- **fstream** : flux pour la lecture et l'écriture de fichier

→ Entête < *fstream* >

Ouverture/fermeture : Politique RAII

- Ouverture du fichier à la construction
- Fermeture à la destruction

Exemple

```
ofstream myfile("example.txt");
myfile << "Writing this to a file.\n";
```

Fichier

Ouverture/fermeture : Méthodes traditionnelles

- *open* : Ouvre le fichier
- *close* : Ferme le fichier

Exemple

```
ofstream myfile;
myfile.open ("example.txt");
myfile << "Writing this to a file.\n";
myfile.close();
```

Remarque : Lors de l'ouverture, création du fichier si il n'existe pas

Fichier

Remarque

Utilisation d'option lors de l'ouverture de fichier

- *ios* :: *in* : Ouverture en lecture seule
- *ios* :: *out* : Ouverture en écriture seule
- *ios* :: *app* : Ajout de données en fin de fichier
- *ios* :: *ate* : Se place en fin de fichier après ouverture
- *ios* :: *nocreate* : Le fichier doit exister
- *ios* :: *trunc* : Écrase le fichier s'il existe
- ios : :binary : Ouverture en mode binaire

Options combinable avec l'opérateur |

Fichier

Exemple : Utilisation de flux de fichier

Code

```
// Ecriture
ofstream fw("res.txt",ios::out);
fw << " Total : " << 123.67 << " euros.";
fw.close();

// Lecture
#define MAX_SIZE 255
char buf[MAX_SIZE];
ifstream fr("res.txt",ios::in);
if (fr)
        while (fr.getline(buf,MAX_SIZE))
                cout << fr << "\n";
fr.close();
```

Gestion des erreurs

À chaque flux est associé un ensemble de bits d'erreur

- **goodbit** : Aucune erreur
- **eofbit** : Fin de flux atteind
- **failbit** : Prochaine opération d'E/S impossible
- **badbit** : Flux non récupérable

Remarque

Lorsque le flot est dans un état d'erreur :

- Aucune opération ne peut être effectuée
- Obligation de corriger l'erreur pour utiliser à nouveau le flux

Gestion des erreurs

Méthodes pour consulter l'état des bits d'erreur :

- **eof()** : renvoie l'état du bit eofbit
- **bad()** : renvoie l'état du bit badbit
- **fail()** : renvoie l'état du bit failbit
- **good()** : renvoie 1 si aucun des 3 bits précédents n'est activé
- **rdstate()** : renvoie le statut d'erreur du flot
- **clear(b)** : active le bit passé en paramètre, et met tous les autres à 0

Opérateurs de flux

Surcharge des opérateurs de redirection << et >>.

Prototype

```
ostream & operator << (ostream&, expression_type_classe)
istream & operator >> (istream&, & type_classe)
```

Dans les surcharges des opérateurs de flux, faire attention à :

- Flux passé en 1er argument à l'opérateur
- Retour du flux modifié
- Gestion des erreurs de flux

Manipulateurs

Manipulateurs

Permet le formatage des données sur le flux

- Formatage par défaut pour les types primitifs
- Modification du formatage à l'aide de manipulateurs

En-tête : < *ios* >

Deux types de manipulateurs

- Avec paramètres
- Sans paramètre

Prototypes

```
istream & nom_manipulateur()
ostream & nom_manipulateur()
istream & nom_manipulateur(argument)
ostream & nom_manipulateur(argument)
```

Manipulateurs

Manipulateurs non paramétriques

- *dec* : E/S numérotation décimale
- *hex* : E/S numérotation hexadécimale
- *oct* : E/S numérotation octale
- *endl* : S saut de ligne, vide le tampon
- *ends* : S fin de chaine (end string)
- *flush* : S vide le tampon
- *ws* : S ignore les espaces (white space)

Manipulateurs

Manipulateurs paramétriques

- *setbase*(*int*) : E/S définit la base de conversion
- *resetiosflags*(*long*) : E/S remet à zéro les bits désignés par l'argument
- *setiosflags*(*long*) : E/S active les bits désignés par l'argument
- *setfill*(*int*) : E/S fixe le caractère de remplissage
- *setprecision*(*int*) : E/S précision des nombres flottants
- *setw*(*int*) : E/S largeur d'affichage

Exemple : Utilisation de manipulateur

Code

```
cout << " 100 en decimal : " << dec << 100;
cout << " 100 en octal : " << oct << 100;
cout << " 100 en hexadecimal : " << hex << 100;
const double PI = 3.14159265358979323;
cout  << "PI a 4 decimales pres : " << setw(4) << PI;
```

Itérateur de flux

Existence d'itérateur de flux :

- **istream_iterator** : itérateur sur flux d'entrée
- **ostream_iterator** : itérateur sur flux de sortie
- **istreambuf_iterator** : itérateur sur flux d'entrée bufferisé
- **ostreambuf_iterator** : itérateur sur flux de sortie bufferisé

Principe

Sucharge de l'opérateur ++ pour extraire ou envoyer des données sur le flux

Exemple : Utilisation d'un *ostream_iterator*

Exemple

```
vector<int> myvector;
for (int i=1; i<10; ++i) myvector.push_back(i*10);

ostream_iterator<int> out_it (cout,", ");
copy ( myvector.begin(), myvector.end(), out_it );
```

Stringstream

Stringstream

Interface permettant de manipuler une string comme un flux.

Existence de différents types :

- **istringstream** : sucharge l'opérateur de flux >>
- **ostringstream** : sucharge l'opérateur de flux <<
- **stringstream** : sucharge les opérateurs de flux << et >>

En-tête : < *sstream* >

Conversion

Principal intérêt :

- Convertion d'objet en string
- Conversion de string en objet

Stringstream

Exemple : Conversion en string

Code

```
template<typename T>
std::string to_string( const T & Value )
{
    std::ostringstream oss;
    oss << Value;
    return oss.str();
}

int main()
{
    std::string num = to_string( 10 );
}
```

Stringstream

Exemple : Conversion depuis une string

Code

```
template<typename T>
bool from_string( const std::string & Str, T & Dest )
{
    std::istringstream iss( Str );
    return iss >> Dest != 0;
}

int main()
{
    int dix;
    from_string( "10", dix );
}
```

Plan

1. Introduction
2. Template
3. Traits, policies et constraints
4. STL - Introduction
5. STL - String
6. STL - Flux
7. STL - Conteneurs
8. STL - Algorithmes
9. STL - Foncteurs
10. STL - Autres

Introduction

Conteneur

Permet de stocker des objets avec :

- Organisation particulière des objets (ordonnée, séquentielle, ...)
- Interface pour manipuler les objets

Stockage

Les conteneurs de la STL stockent des copies des objets donc les objets nécessitent :

- Constructeur par défaut (pour le remplissage)
- Constructeur par copie (pour l'insertion)

Autre solution : Manipulation de pointeurs, mais attention aux fuites mémoire

Catégories

Conteneurs séquentiels

Stockent les éléments de manière séquentielle

- Listes (*list*) ($< list >$)
- Vecteurs (*vector*) ($< vector >$)
- Listes à double entrée (*deque*) ($< deque >$)

Adaptateurs de séquence

Stockent les éléments de manière séquentielle mais propose une interface restrictive

- Piles (*stack*) ($< stack >$)
- Files (*queue*) ($< queue >$)
- Files de priorité (*priority_queue*) ($< priority\_queue >$)

Catégories

Conteneurs associatifs

Stockent les éléments de manière ordonnée à l'aide d'une clé donnant accès à l'objet

- Ensembles (*set*) ($< set >$)
- Multi-ensembles (*multiset*) ($< multiset >$)
- Tables associatives (*map*) ($< map >$)
- Tables associatives multiples (*multimap*) ($< multimap >$)

Conteneurs de bits

Stockent des bits

- Vecteurs de bits ($vector < bool >$)
- Ensembles de bits (*bitset*) ($< bitset >$)

Construction, copie, et destruction

Les conteneurs de la STL possèdent les objets stocké.

Conséquences

- Lors de la desctruction du conteneur, destruction de tous les objets stockés
- Si le conteneur stocke des pointeurs, seule les pointeurs sont supprimés (risque de fuite mémoire)
- Une copie de conteneur entraine une copie des objets stockés

Si stockage d'objet direct (non pointeur), le type d'objet doit assurer :

- Support du constructeur par copie
- Support de l'opérateur d'affectation

Construction, copie, et destruction

Exemple : Manipulation d'une collection d'objet

Code

```
#include <iostream>
#include <vector>
using namespace std;
struct CEntier {
        int m_i;
        CEntier(int i = 0) : m_i(i)
        { cout << "CEntier(" << m_i <<") [" << this << "]\n"; }
        CEntier(const CEntier &p) : m_i(p.m_i)
        { cout << "CEntier(CEntier &) [" << this << " <-- " << (void *)&p << "]\n"; }
        ~CEntier()
        { cout << "~CEntier() [" << this << "]\n"; }
};
void Display(vector<CEntier> v) {
        for (int i = 0; i < v.size(); i++)
                cout << '(' << v[i].m_i << ") ";
        cout << '\n';
}
void main() {
        cout << "Debut du programme\n";
        CEntier a(1), b(2), c(3);
        cout << "Creation du vecteur\n";
        vector<CEntier> *v = new vector<CEntier>;
        v->push_back(a);
        v->push_back(b);
        v->push_back(c);
        cout << "Appel de la fonction Display(vecteur)\n";
        Display(*v);
        cout << "Destruction du vecteur\n";
        delete v;v=0;
        cout << "Fin du programme\n";
}
```

Construction, copie, et destruction

Trace

```
Debut du programme
CEntier(1) [0012FF70]
CEntier(2) [0012FF6C]
CEntier(3) [0012FF68]
Creation du vecteur
CEntier(CEntier &) [004918A0 <-- 0012FF70]
CEntier(CEntier &) [004918A4 <-- 0012FF6C]
CEntier(CEntier &) [004918A8 <-- 0012FF68]
Appel de la fonction Display(vecteur)
CEntier(CEntier &) [00491910 <-- 004918A0]
CEntier(CEntier &) [00491914 <-- 004918A4]
CEntier(CEntier &) [00491918 <-- 004918A8]
(1) (2) (3)
~CEntier() [00491910]
~CEntier() [00491914]
~CEntier() [00491918]
Destruction du vecteur
~CEntier() [004918A0]
~CEntier() [004918A4]
~CEntier() [004918A8]
Fin du programme
~CEntier() [0012FF68]
~CEntier() [0012FF6C]
~CEntier() [0012FF70]
```

Principales méthodes

Principales méthode communes à tous les conteneurs

bool empty() const
Renvoie true si le conteneur ne contient pas d'éléments

bool empty() const
Renvoie true si le conteneur n'a pas d'élément.

size_t size() const
Renvoie le nombre d'élément dans le conteneur.

Remarques

size() et **empty()** existent pour tous les conteneurs et s'exécute en temps constant

Principales méthodes

void resize(size_t nouvelle_taille, T valeur)
Changement de la taille du conteneur

size_t capacity() const
Retourne le nombre d'éléments que le conteneur peut maintenir sans nécessiter une réallocation.

void reserve(size_t n)
Modifie la taille de l'espace réservé pour le conteneur (celui dont la taille est donnée par capacity).

size_t max_size() const
Retourne le nombre maximum d'éléments que le conteneur peut contenir.

Principales méthodes

T& operator[](clé)
const T& operator[](clé) const
T& at(clé)
const T& at(clé) const
Accès indexé (Map et multimap : clé de stockage, autre : clé entière)

T& front()
const T& front() const
Accès à l'élément qui est en tête.

Principales méthodes

T& back()
const T& back() const
Accès à l'élément qui est en queue.

void push_front(const T &)
void push_back(const T &)
Ajout d'un élément en tête [resp. en queue].

void pop_front()
void pop_back()
Suppression de l'élément en tête [resp. en queue].

Principales méthodes

Exemple : Utilisation des méthodes

Code

```
#include <iostream>
#include <vector>
using namespace std;
void main()
{
        vector<int> v;
        for (int i = 0; i < 18; i++){
                cout << "size: " << v.size() << " - - capacity: " << v.capacity()
                << " - - max: " << v.max_size() << "\n";
                v.push_back(i);}
}
```

Principales méthodes

Trace

```
size: 0 -- capacity: 0 -- max: 1073741823
size: 1 -- capacity: 1 -- max: 1073741823
size: 2 -- capacity: 2 -- max: 1073741823
size: 3 -- capacity: 4 -- max: 1073741823
size: 4 -- capacity: 4 -- max: 1073741823
size: 5 -- capacity: 8 -- max: 1073741823
size: 6 -- capacity: 8 -- max: 1073741823
size: 7 -- capacity: 8 -- max: 1073741823
size: 8 -- capacity: 8 -- max: 1073741823
size: 9 -- capacity: 16 -- max: 1073741823
size: 10 -- capacity: 16 -- max: 1073741823
size: 11 -- capacity: 16 -- max: 1073741823
size: 12 -- capacity: 16 -- max: 1073741823
size: 13 -- capacity: 16 -- max: 1073741823
size: 14 -- capacity: 16 -- max: 1073741823
size: 15 -- capacity: 16 -- max: 1073741823
size: 16 -- capacity: 16 -- max: 1073741823
size: 17 -- capacity: 32 -- max: 1073741823
```

Itérateurs

Itérateur

Objets facilitant l'accès et les opérations sur les éléments d'un conteneur

Principales méthodes des containeurs relatives aux itérateurs :

iterateur insert(itérateur , const T& x)
void insert(itérateur , size_t n, const T& x)
Insertion d'une valeur x [resp. de n copies d'une valeur x].

iterator erase(itérateur)
void clear()
Suppression d'une valeur [resp. de toutes les valeurs].

iterateur begin()
const itérateur begin() const
iterateur end()
const itérateur end() const
Iterateur positionné sur le premier (resp. après le dernier) élément.

Itérateurs

iterateur rbegin()
const itérateur rbegin() const
iterateur rend()
const itérateur rend() const
Iterateur inverse positionné sur le dernier (resp. après le premier) élément.

iterateur void swap(vector¡T¿ &v)
Echange des valeurs des vecteurs *this et v.

Manipulation des itérateurs

- **Accès donnée** : opérateur $*$ et opérateur $->$
- **Déplacement sur l'élément suivant** : opérateur++ (préfixe et postfixe)
- **Déplacement sur l'élément précédent** : opérateur – (préfixe et postfixe)
- **Comparaison** : opérateur == et !=

Itérateurs

Exemple : Parcours de conteneur

Code

```
list<int> l;
for (int i = 1; i < 10; i++) l.push_back(i);

int sum =0;
list<int>::iterator it;
for( it = l.begin() ; it != l.end() ; it++ ) sum+= *it;
// sum = 1+2+...+9 = 45
```

Exemple : Parcours de conteneur en sens inverse

Code

```
list<int>::reverse_iterator rit;
for( rit = l.rbegin() ; rit != l.rend() ; rit++ ) sum+= *rit;
// sum = 9+8+...+1 = 45
```

Conteneurs séquentiels

Rappels des trois types de conteneurs

- *list*
- *vector*
- *deque*

Performance : Complexité pour les opérations d'accès, d'ajout et de suppression

Accès direct aux éléments

- **vector** : Oui en temps constant
- **list** : Non
- **deque** : Oui en temps constant

Insertion/suppression au début

- **vector** : Temps linéaire
- **list** : Temps constant
- **deque** : Temps constant

Conteneurs séquentiels

Insertion/suppression à la fin

- **vector** : Temps constant
- **list** : Temps constant
- **deque** : Temps constant

Insertion/suppression entre 2 éléments

- **vector** : Temps linéaire
- **list** : Temps constant
- **deque** : Temps linéaire

Remarque sur les utilisations

- Utilisation de *list* pour les insertions/suppressions fréquentes en tout point de la collection
- Utilisation *vector* et *deque* pour les accès directs

Conteneurs séquentiels

Quelques méthodes spécifiques au conteneur **list**

void splice(iterator pos, list¡T¿& x)
void splice(iterator pos, list¡T¿& x, iterator i)
void splice(iterator pos, list¡T¿& x, iterator premier,iterator dernier)
Coupe une liste en deux parties à la position donné. Les éléments supprimés sont insérés dans une nouvelle liste

void remove(const T& valeur)
template ¡class PredicatUn¿ void remove_if(PredicatUn predicat)
Supprime les éléments égaux à une valeur donnée (premier cas) ou vérifiant un prédicat

void unique()
template ¡class PredicatBin¿ void unique(PredicatBin pred)
Supprime tous les éléments consécutif égaux et n'en conserve qu'un seul exemplaire

Conteneurs séquentiels

void sort()
template ¡class Compare¿ void sort(Compare comp)
void merge(list¡T,Allocator¿& x)
template ¡class Compare¿ void merge(list¡T,Allocator¿& x, Compare comp)
Trie ou fusionne des listes ordonnées

void reverse()
Inverse les éléments d'une liste

Adaptateur de conteneur

Rappels des trois types de conteneurs

- *stack*
- *queue*
- *priority_queue*

Principe

Utilisation d'une conteneur séquentiel et modification de son interface afin que l'élément extrait :

- D'une *stack* soit le plus récemment inséré.
- D'une *queue* soit le moins récemment inséré.
- D'une *priority_queue* soit le plus prioritaire (priorité déterminée par *Compare*()).

Adaptateur de conteneur

stack

- **Méthodes** : *push*(), *top*(), *pop*()
- **Implémentation** : *vector*, *deque* (par défaut)
- **Insertion/suppression à la fin** : Temps constant
- **Insertion/suppression entre 2 éléments** : Temps linéaire

queue

- **Méthodes** : *front*(), *back*(), *push_back*(), *pop_front*()
- **Implémentation** : *deque* (toujours meilleur choix)
- **Insertion/suppression à la fin** : Temps constant
- **Insertion/suppression entre 2 éléments** : Temps constant

Adaptateur de conteneur

priority_queue

- **Méthodes** : *front*(), *push_back*(), *pop_back*()
- **Implémentation** : *vector* (par défaut), *deque*
- **Insertion/suppression à la fin** : Temps constant
- **Insertion/suppression entre 2 éléments** : Temps linéaire

Adaptateur de conteneur

Exemple : Utilisation d'une file à priorité (Éléments entiers avec nombres paires prioritaires sur les impairs)

Code

```
#include <iostream>
#include <queue>
using namespace std;
struct moindre : public binary_function<int, int, bool>
{
        bool operator()( int a, int b)
        {
        if (a % 2 == 0)
                return b % 2 != 0 ? false : a < b;
        else
                return b % 2 == 0 ? true : a < b;
        }
};
void main()
{
        int x, i;
        priority_queue<int, vector<int>, moindre> p;
        for (i = 0; i < 16; i++)
        {
                cout << (x = rand() % 100) << ' ';
                p.push(x);
        }
        cout << '\n';
        while ( ! p.empty())
        {
                cout << p.top() << ' ';
                p.pop();
        }
        cout << '\n';
```

Adaptateur de conteneur

Trace

```
41 67 34 0 69 24 78 58 62 64 5 45 81 27 61 91
78 64 62 58 34 24 0 91 81 69 67 61 45 41 27 5
```

Conteneurs associatifs

Principe

- Conteneur ordonné
- Association d'une clé à un élément

Rappels des quatre types de conteneurs :

- *set* et *multiset* : stockent des éléments (clé=élément)
- *map* et *multimap* : stockent des associations clé/valeur

Remarque

multiset et *multimap* peuvent contenir des clés équivalentes

Conteneurs associatifs

Conteneur associatif

Définition d'**Ordre total** entre les éléments

Compare() : induit un ordre strict faible sur les clés :

- **Égalité** : $A == B$ si $A < B$ et $A > B$
- **Transitivité** : si $A < B$ et $B < C$ alors $A < C$

Prototype : $bool\ Compare(key, key)$

Comparaison par défaut

Par défaut utilisation du foncteur **less** : utilise l'opérateur $<$

Conteneurs associatifs

Possibilité de redéfinir l'opérateur de comparaison

1. Créer une structure
2. Redéfinir l'opérateur () prenant deux objets
3. Implémenter la comparaison

→ Par défaut *Compare*() utilise le foncteur *less* (*operator* <)

Exemple

```
struct LessString {
        bool operator()(const char* s1, const char* s2) const {
                return strcmp(s1, s2) < 0;
        }
};

typedef map<const char*, int, LessString> TStringMap;
```

Conteneurs associatifs

Exemple : Utilisation d'une map

Code

```
struct ltstr {
        bool operator()(const char* s1, const char* s2) const {
                return strcmp(s1, s2) < 0;
        }
};

int main() {
        map<const char*, int, ltstr> months;

        months["january"] = 31;
        months["february"] = 28;
        months["march"] = 31;
        months["april"] = 30;
        months["may"] = 31;
        months["june"] = 30;
        months["july"] = 31;
        months["august"] = 31;
        months["september"] = 30;
        months["october"] = 31;
        months["november"] = 30;
        months["december"] = 31;

        cout << "june -> " << months["june"] << endl;
        map<const char*, int, ltstr>::iterator cur = months.find("june");
        map<const char*, int, ltstr>::iterator prev = cur;
        map<const char*, int, ltstr>::iterator next = cur;
        ++next;
        --prev;
        cout << "Previous (in alphabetical order) is " << (*prev).first << endl;
        cout << "Next (in alphabetical order) is " << (*next).first << endl;
}
```

Conteneurs de bits

Rappels sur les deux types de conteneurs

- *vector < bool >*
- *bitset*

vector < bool >

- Taille dynamique

bitset

- Taille fixe
- Meilleur utilisation pour les opérations sur les bits
- Bits rangés en mémoire de manière compacte
- Tire profit des opérations logiques bit à bit

Conteneurs de bits

Principales méthode d'un conteneur *bitset*

bitset() ;
bitset(unsigned long val) ;
explicit bitset(string s, size_t pos, size_t nbr)
Construction

bitset<N>& operator&=(const bitset<N>& rhs)
bitset<N>& operator—=(const bitset<N>& rhs)
bitset<N>& operator≐(const bitset<N>& rhs)
bitset<N>& operator<<=(size_t pos)
bitset<N>& operator>>=(size_t pos)
bitset<N> operator<<(size_t pos) const
bitset<N> operator>>(size_t pos) const
Opérations

Conteneurs de bits

bitset<N>& set()
bitset<N>& set(size_t pos, int val = true)
bitset<N>& reset()
bitset<N>& reset(size_t pos)
Activation (*set*) et désactivation (*reset*)

bitset<N> operator () const
bitset<N>& flip()
bitset<N>& flip(size_t pos)
Bascule

unsigned long to_ulong() const
Conversion en ulong

reference operator[](size_t pos)
Accès à un bit

Conteneurs de bits

string to_string() const
Retourne une string de type "1100110"

bool operator==(const bitset<N>& rhs) const
Égalité entre 2 bitset

bool operator !=(const bitset<N>& rhs) const
Différence entre 2 bitset

bool test(size_t pos) const
Retourne true si LE bit est 1

bool any() const ;
Retourne true si au moins UN bit est 1

Conteneurs de bits

bool none() const
Retourne true si tous les bits sont à 0

size_t size() const
Retourne la taille du tableau (l'argument générique)

size_t count() const
Retourne le nombre de bit à 1

Conteneurs de bits

Exemple : Utilisation d'un *bitset*

Code

```
int main() {
        const bitset<12> mask(2730ul);
        cout << "mask = " << mask << endl;

        bitset<12> x;

        cout << "Enter a 12-bit bitset in binary: " << flush;
        if (cin >> x) {
                cout << "x = " << x << endl;
                cout << "As ulong: " << x.to_ulong() << endl;
                cout << "And with mask: " << (x & mask) << endl;
                cout << "Or with mask: " << (x | mask) << endl;
        }
}
```

Plan

1. Introduction
2. Template
3. Traits, policies et constraints
4. STL - Introduction
5. STL - String
6. STL - Flux
7. STL - Conteneurs
8. STL - Algorithmes
9. STL - Foncteurs
10. STL - Autres

Introduction

La STL fournit de nombreux algorithmes sur les éléments des conteneurs

- **Consultation d'une séquence**
  Exemple : recherche d'élément, éléments satisfaisant un prédicat...)
- **Modification d'une séquence**
  Exemple : copie d'éléments...
- **Appliqués aux séquences ordonnées**
  Exemple : tri, nième élément...

Remarques

- La plupart des algorithmes sont *inline* car court
- Représentation d'une séquence par deux itérateurs (début : premier élément, fin : après le dernier élément)
- Quand retourne l'élément après le dernier, l'algorithme a échoué

En-tête < *algorithm* >

Introduction

Exemple : Recherche d'un élément dans une liste

Code

```
list<int> L;
 ...
list<int>::iterator r = find(L.begin(), L.end(), x);
if (r != L.end())
// Trouve
else
// Non trouve
```

Consultation de séquences

for_each

template<class Iter, class Fonc>
fonc for_each(Iter debut, Iter fin, Fonc f)
Action : appel de f sur chaque élément de la séquence [début, fin [.
Retour : f

find

template<class IterIn, class T>
Iter find(Iter debut, Iter fin, const T& valeur)
Action : recherche le premier élément de la séquence [debut, fin [égal à la valeur indiquée.
Retour : un itérateur positionné sur l'élément en question, s'il existe ; la valeur fin sinon.
Contrainte : la relation égalité doit être définie sur le type T.

Consultation de séquences

Exemple : Appliquer une action

Code

```
void myfunction (int i) { cout << " " << i; }

struct myclass {
        void operator() (int i) {cout << " " << i;}
} myobject;

int main () {
        vector<int> myvector;
        myvector.push_back(10);
        myvector.push_back(20);
        myvector.push_back(30);

        cout << "myvector contains:";
        for_each (myvector.begin(), myvector.end(), myfunction);

        cout << "\nmyvector contains:";
        for_each (myvector.begin(), myvector.end(), myobject);
        return 0;
}
```

Trace

```
myvector contains: 10 20 30
myvector contains: 10 20 30
```

Consultation de séquences

Exemple : Trouver un élément dans une collection

Code

```
int myints[] = { 10, 20, 30 ,40 };
int * p;

// pointer to array element:
p = find(myints,myints+4,30);
++p;
cout << "The element following 30 is " << *p << endl;

vector<int> myvector (myints,myints+4);
vector<int>::iterator it;

// iterator to vector element:
it = find (myvector.begin(), myvector.end(), 30);
++it;
cout << "The element following 30 is " << *it << endl;
```

Trace

```
The element following 30 is 40
The element following 30 is 40
```

Consultation de séquences

find_if

template<class Iter, class PredUn>
Iter find_if(Iter first, Iter last, PredUn pred)

Action : recherche le premier élément x de la séquence [debut, fin [tel que pred(x) false

Retour : un itérateur positionné sur l'élément en question, s'il existe ; la valeur fin sinon.

count

template<class Iter, class T>
typename iterator_traits<Iter> : :difference_type count(Iter debut, Iter fin, const T& valeur)

Action : comptage du nombre d'éléments de la séquence [debut, fin [égaux à la valeur indiquée.

Retour : le nombre de tels éléments. **Contrainte** : la relation égalité doit être définie sur le type T.

Consultation de séquences

count_if

template<class Iter, class PredUn>
typename iterator_traits<Iter> : :difference_type count_if(Iter debut, Iter fin, PredUn pred)

Action : comptage du nombre d'éléments x de la séquence [debut, fin [pour lesquels pred(x) false.

Retour : le nombre de tels éléments.

equal

template<class Iter1, class Iter2>
bool equal(Iter1 debut1, Iter1 fin1, Iter2 debut2)

Retour : la réponse à la question : les séquences [debut1, fin1 [et [debut2, fin2 [sont-elles égales ?

Les autres : *find_end*, *find_first_of*, *adjacent_find*, *mismatch*, *search*, *search_n*

Modification de séquences

copy

template<class Iter, class Iter>
Iter copy(Iter debut, Iter fin, Iter result)
Action : copie, en avançant, des éléments de la séquence [debut, fin [sur les éléments de la séquence [result, result + fin - debut [. Equivaut à : pour i allant de 0 à fin – debut - 1, faire *(result + i) = *(debut + i) **Retour** : la valeur result + fin – debut.
Contrainte : result ne doit pas appartenir à l'intervalle [debut, fin [.

swap

template<class T> void swap(T& a, T& b)
Action : échange les éléments a et b.
Contrainte : le type T doit supporter l'affectation

Modification de séquences

Exemple : Trouver un élément dans une collection

Code

```
int myints[]={10,20,30,40,50,60,70};
vector<int> myvector;
vector<int>::iterator it;

myvector.resize(7);  // allocate space for 7 elements

copy ( myints, myints+7, myvector.begin() );

cout << "myvector contains:";
for (it=myvector.begin(); it!=myvector.end(); ++it)
cout << " " << *it;

cout << endl;

return 0;
```

Trace

```
myvector contains: 10 20 30 40 50 60 70
```

Modification de séquences

iter_swap

template<class Iter1, class Iter2>
void iter_swap(Iter1 a, Iter2 b)
Action : échange les éléments *a et *b.

replace

template<class Iter, class T>
void replace(Iter debut, Iter fin,const T& oldVal, const T& newVal)
Action : toutes les occurrences de oldVal dans la séquence [debut, fin [sont remplacées par newVal.
Contrainte : le type T doit supporter l'égalité et l'affectation.

Modification de séquences

replace_if

template<class Iter, class PredUn, class T>
void replace_if(Iter debut, Iter fin, PredUn pred, const T& newVal)
Action : tout x de la séquence [debut, fin [qui vérifie pred(x) false est remplacé par newVal.
Contrainte : le type T doit supporter l'affectation.

fill

template<class Iter, class T>
void fill(Iter debut, Iter fin, const T& valeur)
Action : tous les éléments de la séquence [debut, fin [sont affectés de la valeur indiquée.
Contrainte : le type T doit supporter l'affectation

Modification de séquences

Exemple : Appliquer une action

Code

```
bool IsOdd (int i) { return ((i%2)==1); }

int main () {
        vector<int> myvector;
        vector<int>::iterator it;

        // set some values:
        for (int i=1; i<10; i++) myvector.push_back(i);         // 1 2 3 4 5 6 7 8 9

        replace_if (myvector.begin(), myvector.end(), IsOdd, 0); // 0 2 0 4 0 6 0 8 0

        cout << "myvector contains:";
        for (it=myvector.begin(); it!=myvector.end(); ++it)
        cout << " " << *it;

        cout << endl;

        return 0;
}
```

Trace

```
myvector contains: 0 2 0 4 0 6 0 8 0
```

Modification de séquences

generate

template<class Iter, class Generateur>
void generate(Iter debut, Iter fin, Generateur gen)
Action : tous les éléments de la séquence [debut, fin [sont affectés de la valeur gen().
Contrainte : Generateur est un type objet fonction sans argument.

remove

template<class Iter, class T>
Iter remove(Iter debut, Iter fin, const T& valeur)
Action : suppression des éléments de la séquence [debut, fin [qui sont égaux à la valeur indiquée.
Retour : la fin de la séquence résultante.
Contrainte : le type T supporte l'égalité.

Modification de séquences

unique

template<class Iter>
Iter unique(Iter debut, Iter fin)
Action : remplacement de chaque sous-séquence faite d'éléments (consécutifs) égaux par son premier élément.
Retour : la fin de la séquence résultante.

reverse

template<class IterBid>
void reverse(IterBid debut, IterBid fin)
Action : renversement de la séquence indiquée. Equivaut à :
pour i allant de 0 à (fin – debut) / 2 – 1 faire swap(*(debut + i), *(fin – i – 1))

Modification de séquences

random_shuffle

template<class IterAlea>
void random_shuffle(IterAlea debut, IterAlea fin)

Action : arrangement aléatoire des éléments de la séquence [debut, fin [, selon une distribution uniforme (c'est-à-dire que chacune des (fin – debut) ! permutations possibles a autant de chances d'être employée)

partition

template<class IterBid, class PredUn>
IterBid partition(IterBid debut, IterBid fin, PredUn pred)

Action : réarrange les éléments de la séquence [debut, fin [de telle manière que tous ceux pour lesquels la condition exprimée par pred est vraie se trouvent devant tous ceux pour lesquels cette condition est fausse.

Retour : une valeur d'itérateur r telle que pour toute valeur debut i $<$ r on a pred(*i) != false et pour toute valeur r j $<$ fin on a pred(*j) == false.

Les autres : *copy_backward*, *swap_ranges*, *transform*, *replace_copy*, *replace_copy_if*, *fill_n*, *generate_n*, *remove_if*, *remove_copy*, *remove_copy_if*, *unique_copy*, *reverse_copy*, *rotate*, *rotate_copy*

Algorithmes de séquences ordonnées

sort

template<class IterAlea>
void sort(IterAlea debut, IterAlea fin)
template<class IterAlea, class Compare>
void sort(IterAlea debut, IterAlea fin, Compare comp)
Action : tri des éléments de la séquence [debut, fin [

nth_element

template<class IterAlea >
void nth_element(IterAlea debut, IterAlea nth, IterAlea fin)
template<class IterAlea, class Compare>
void nth_element(IterAlea debut, IterAlea nth, IterAlea fin,Compare comp)
Action : après l'exécution de cet algorithme, l'élément à la position nth est celui qui se trouverait à cet endroit si la séquence [debut, fin [était triée. De plus, aucun élément de [nth, fin [n'est inférieur à aucun élément de [debut, nth [.

Algorithmes de séquences ordonnées

lower_bound

template<class Iter, class T>
Iter lower_bound(Iter debut, Iter fin, const T& valeur)
template<class Iter, class T, class Compare>
Iter lower_bound Iter debut, Iter fin, const T& valeur,Compare comp)
Retour : un itérateur pointant la position la plus à gauche, dans la séquence triée [debut, fin [, à laquelle on peut placer la valeur indiquée sans violer l'ordre.
Contrainte : la séquence [debut, fin [doit être triée.

Algorithmes de séquences ordonnées

merge

template<class Iter1, class Iter2, class Iter>
Iter merge(Iter1 debut1, Iter1 fin1,Iter2 debut2, Iter2 fin2, Iter result)
template<class Iter1, class Iter2, class Iter, class Compare>
Iter merge(Iter1 debut1, Iter1 fin1,Iter2 debut2, Iter2 fin2, Iter result, Compare comp)

Action : fusion des deux séquences triées [debut1, fin1 [et [debut2, fin2 [en une unique séquence triée [result, result + (fin1 – debut1) + (fin2 – debut2) [.
Retour : la valeur result + (fin1 – debut1) + (fin2 – debut2)
Contrainte : les deux séquences [debut1, fin1 [et [debut2, fin2 [doivent être triées.

Algorithmes de séquences ordonnées

includes

template<class Iter1, class Iter2>
bool includes(Iter1 debut1, Iter1 fin1,Iter2 debut2, Iter2 fin2)
template<class Iter1, class Iter2, class Compare>
bool includes(Iter1 debut1, Iter1 fin1,Iter2 debut2, Iter2 fin2, Compare comp)
Retour : la réponse à la question 'les éléments de la séquence [debut1, fin1 [apparaissent-ils tous dans la séquence [debut2, fin2 [?'
Contrainte : les deux séquences [debut1, fin1 [et [debut2, fin2 [doivent être triées.

Algorithmes de séquences ordonnées

set_union

template<class Iter1, class Iter2, class Iter>
Iter set_union(Iter1 debut1, Iter1 fin1,Iter2 debut2, Iter2 fin2, Iter result)
template<class Iter1, class Iter2, class Iter, class Compare>
Iter set_union(Iter1 debut1, Iter1 fin1,Iter2 debut2, Iter2 fin2, Iter result,
Compare comp)

Action : construit la réunion des séquences triées [debut1, fin1 [et [debut2, fin2 [, sous la forme d'une séquence triée rangée à partir de la position result.
Retour : la fin de la séquence résultante.
Contrainte : les séquences [debut1, fin1 [et [debut2, fin2 [doivent être triées.

Algorithmes de séquences ordonnées

min

```
template<class T>
const T& min(const T& a, const T& b)
template<class T, class Compare>
const T& min(const T& a, const T& b, Compare comp)
```

Retour : le plus petit des deux éléments (ou, s'ils sont équivalents, le premier).

min_element

```
template<class Iter>
Iter min_element(Iter debut, Iter fin)
template<class Iter, class Compare>
Iter min_element(Iter debut, Iter fin, Compare comp)
```

Retour : l'itérateur i le plus à gauche tel qu'aucun élément de la séquence ne soit strictement inférieur à *i.

Les autres : *stable_sort*, *partial_sort*, *partial_sort_copy*, *upper_bound*, *equal_range*, *binary_search*, *inplace_merge*, *set_intersection*, *set_difference*, *set_symmetric_difference*, *push_heap*, *pop_heap*, *make_heap*, *sort_heap*, *max*, *max_element*, *lexicographical_compare*, *next_permutation*, *prev_permutation*

Plan

1. Introduction
2. Template
3. Traits, policies et constraints
4. STL - Introduction
5. STL - String
6. STL - Flux
7. STL - Conteneurs
8. STL - Algorithmes
9. STL - Foncteurs
10. STL - Autres

Principe

Functor (foncteur/objet)

Désigne toute entité qui peut être appelée comme une fonction :

- Une fonction
- Un pointeur sur une fonction ou une méthode
- Un objet dont la classe surcharge l'opérateur()

En-tête : < *functional* >

Exemple

```
struct A {  void opertor() {...} };
A a;
a();
```

⟶ Permet d'utiliser un objet de la même manière qu'une fonction

Avantages

Principaux avantages

- Possibilité d'avoir un comportement 'stateful'
- Passage de paramètres supplémentaires hors prototype
- Permet une paramètrisation en template (générique)
- Très utilisé dans la STL

Utilisation principale dans la STL :

- Prédicat
- Comparaison

Types

Nombreux modèles de foncteurs dans la STL (<functional>) :

Opérations arithmétiques

- divides (\)
- logical_and (&&)
- logical_or (||)
- minus (−)
- modulus (%)
- multiplies (∗)
- plus (+)
- logical_not (!)
- negate (−)

Types

Exemple : Utilisation de *plus*

Code

```
// plus example
#include <iostream>
#include <functional>
#include <algorithm>
using namespace std;

int main () {
        int first[]={1,2,3,4,5};
        int second[]={10,20,30,40,50};
        int results[5];
        transform ( first, first+5, second, results, plus<int>() );
        for (int i=0; i<5; i++)
          cout << results[i] << " ";
        cout << endl;
        return 0;
}
```

Trace

```
11 22 33 44 55
```

Types

Opérateur de comparaisons

- equal_to
- greater
- greater_equal
- less
- less_equal
- not_equal_to

Types

Exemple : Utilisation de *less*

Code

```
int foo[]={10,20,5,15,25};
int bar[]={15,10,20};
sort (foo, foo+5, less<int>() ); // 5 10 15 20 25
sort (bar, bar+3, less<int>() ); //   10 15 20
if ( includes ( foo, foo+5, bar, bar+3, less<int>() ) )
cout << "foo includes bar.\n";
```

Implémentation

Deux modèles de base

- Foncteur **unaire**
- Foncteur **binaire**

Unary

```
template<typename T, typename R>
struct unary_function
{
        typedef T argument_type;
        typedef R result_type;
};

result_type unary_function::operator(argument_type _arg)
```

Binary

```
template<typename T1, typename T2, typename R>
struct binary_function
{
        typedef T1 first_argument_type;
        typedef T2 second_argument_type;
        typedef R result_type;
};

result_type binary_function::operator(first_argument_type _arg1, second_argument_type _arg2)
```

Implémentation

Exemple : Implémentation de la classe *plus* dérivant de binary_function

Code

```
template<class T> struct plus : binary_function<T, T, T>
{
        T operator()(const T &x, const T &y) const  { return x + y; }
};
```

Adaptateur

Adaptateur

Utilisation de foncteur pour modifier l'interface :

- De fonctions
- De fonctions membres

Trois types d'adaptateur :

- **ptr_fun**
  Fonction membres statiques ou non-membres
- **mem_fun_ref**
  Fonction membre non-statique avec une référence sur l'objet
- **mem_fun**
  Fonction membre non-statique avec un pointeur sur objet

Remarque : Pour les fonctions membre non statique besoin d'un objet pour invoquer la méthode

Adaptateur

Exemple : mem_fun_ref

L'opérateur () prend en paramètre :

1. Une référence sur l'objet à invoquer
2. Un arguement de la fonction (Optionnel)

Variantes du foncteur :

- const_mem_fun_ref_t
- const_mem_fun1_ref_t
- mem_fun_ref_t
- mem_fun1_ref_t

Adaptateur

Création du foncteur avec la méthode mem_fun_ref

Sans argument

```
template <class S, class T>
mem_fun_ref_t<S,T> mem_fun_ref (S (T::*f)());
```

Avec argument

```
template <class S, class T, class A>
mem_fun1_ref_t<S,T,A> mem_fun_ref (S (T::*f)(A));
```

Adaptateur

Exemple : Utilisation de mem_fun_ref

Code

```
vector<string> numbers;

// populate vector:
numbers.push_back("one");
numbers.push_back("two");
numbers.push_back("three");
numbers.push_back("four");
numbers.push_back("five");

vector <int> lengths (numbers.size());

transform (numbers.begin(), numbers.end(), lengths.begin(), mem_fun_ref(&string::length));

for (int i=0; i<5; i++) {
        cout << numbers[i] << " has " << lengths[i] << " letters.\n";
}
```

Trace

```
one has 3 letters.
two has 3 letters.
three has 5 letters.
four has 4 letters.
five has 4 letters.
```

Lieurs

Lieur

Permet de lier une valeur à un des paramètres, pour une fonction à 1 arguement

Intérêt : possibilité de d'utiliser des foncteurs à deux paramètres quand un foncteur à un paramètre est requis

Exemple

```
int numbers[] = {10,20,30,40,50,10};
int cx;
cx = count\_if (numbers, numbers+6, bind1st(equal\_to<int>(),10) );
cout << "There are " << cx << " elements that are equal to 10.\n";
```

Exemple

```
int numbers[] = {10,-20,-30,40,-50};
int cx;
cx = count\_if ( numbers, numbers+5, bind2nd(less<int>(),0) );
cout << "There are " << cx << " negative elements.\n";
```

Plan

1. Introduction
2. Template
3. Traits, policies et constraints
4. STL - Introduction
5. STL - String
6. STL - Flux
7. STL - Conteneurs
8. STL - Algorithmes
9. STL - Foncteurs
10. STL - Autres

Locale

Locale

La STL propose des éléments pour gérer les particularités de chaque partie du monde :

- Alphabet
- Date et heure
- Unité monétaire
- Unité de mesure
- ...

En-tête : < *locale* >

Valarray

Valarray

- Structure de donnée permettant de faire subir les traitements au tableau lui-même
- Compilateur garantit le traitement sur chaque élément
- Compilateur garantit l'optimisation si il y a plusieurs processeurs

En-tête : < *valarray* >

Quelques fonctionnalités

- Opérateurs binaires de comparaisons
- Opérateurs binaires arithmétiques
- Opérations de décalages et de rotations

Valarray

Exemple : Application d'opération arithmétiques

Code

```
#include <valarray>
using namespace std;
void main()
{
        valarray<double> Tab(3.14,30);
        Tab *= Tab;
        Tab += 2.1;
}
```