CHƯƠNG 6: KHUÔN MẪU (TEMPLATE) VÀ THƯ VIỆN CHUẨN (STL)

Trần Thị Ngân

Email: ngantt@tlu.edu.vn

Nội dung

- 6.1 Nhắc lại về vector
- 6.2 Làm việc với c-string và string
- 6.3 Khuôn mẫu hàm
- 6.4 Khuôn mẫu lớp

NỘI DUNG

6.1 Nhắc lại về vector

6.2 Làm việc với c-string và string

6.3 Khuôn mẫu hàm

6.4 Khuôn mẫu lớp

Cơ bản về vector

- Dùng để lưu trữ tập dữ liệu CÙNG KIỂU, giống mảng, Nhưng vector có thể thay đổi kích thước trong lúc chạy chương trình (không giống như mảng có kích thước cố định)
- Thư viện: #include <vector>
- Ví dụ khai báo
  - *vector<int> A;*
    vector<int> B(10);
    vector<int> C(10,2);

Một số hàm thành viên của vector

http://www.cplusplus.com/reference/vector

Phương thức	Mục đích
v.assign(n,e)	Gán tập giá trị mới cho vector, thay thế nội dung hiện tại của nó đồng thời thay đổi kích thước
v[i] hoặc v.at[i]	Tham chiếu đến phần tử thứ i của vector
v.clear()	Xóa toàn bộ vector
v.pop_back()	Xóa phần tử cuối cùng của vector
v.erase(position)	Xóa phần tử ở vị trí position
v.erase(first, last)	Xóa các phần tử trong khoảng từ first tới last
v.push_back(e)	Thêm phần tử e vào cuối của vector
v.resize(new_size)	Thay đổi kích thước của vector

NỘI DUNG

6.1 Nhắc lại về vector

6.2 Làm việc với c-string và string

6.3 Khuôn mẫu hàm

6.4 Khuôn mẫu lớp

C-string và lớp string

- C-Strings: một kiểu mảng cho chuỗi ký tự
- Các công cụ thao tác ký tự (char)
- Character I/O, cin/getline
- Hàm thành viên: *get, put*
- Một số hàm khác: *pushback, peek, ignore* ...
- Lớp String chuẩn
- Xử lý chuỗi ký tự với lớp String

Hai cách biểu diễn chuỗi (string)

- **C-strings**
  - Một mảng với các phần tử có kiểu cơ sở char
  - Chuỗi được kết thúc với kí tự null, “\0”
  - Là phương thức cũ được kế thừa từ C
- **Lớp String**
  - Sử dụng khuôn mẫu (template)

C-strings

- Một mảng các phần tử với kiểu cơ sở char
- Mỗi phần tử của mảng là một ký tự
- Ký tự mở rộng “\0” (được gọi là ký tự rỗng)
- Là dấu hiệu kết thúc một chuỗi ký tự
- Chúng ta đã sử dụng C-strings!
- Ví dụ: literal “Hello” được lưu trữ như một C-string

Biến c-strings

- Khai báo: char s[10];
- Khai báo một biến c-string để lưu trữ 9 ký tự
- Và kí tự cuối cùng là ký tự kết thúc(null- '\0')
- Chỉ có một điểm khác với mảng chuẩn:
  - C-strings phải chứa ký tự null !
- Khởi tạo một c-strings: char st[10] = "Hi Mom!";

st[0]	st[1]	st[2]	st3]	st[4]	st[5]	st[6]	st[7]	st[8]	st[9]..
H	i		M	o	m	!	\o	?	?

- Không cần thiết phải điền đầy đủ (kích thước) mảng
- Đặt ký tự "\0" ở cuối
- Có thể bỏ qua kích thước mảng

Thao tác với c-string

- Một c-string LÀ một mảng => có thể truy cập thành viên thông qua chỉ số (index)
  Ví dụ: char ourString[5] = "Hi";
  ourString[0] là "H“
  ourString[1] là "i“
  ourString[2] là "\0“
  ourString[3] là không xác định (unknown)
  ourString[4] là không xác định (unknown)
- Chú ý: nếu thực hiện phép gán ourString[2] = ‘b’;
  Ghi đè ký tự “\0” (null) bởi ký tự “b”
- Nếu ký tự null bị ghi đè, c-string không còn hoạt động như c- string nữa! => kết quả không dự đoán được

Nhập dữ liệu

char mystring[10]= “Hello!”;

char newstring [20];

- cin

```
std::cin>>newstring;
```

- getline

```
std::cin.getline(newstring,12);
```

Toán tử gán

- C-strings không giống những biến khác
- Không thể sử dụng phép gán hoặc so sánh
- Chỉ có thể sử dụng toán tử “=” lúc khởi tạo một c-string! char aString[10]; aString = “Hello"; // KHÔNG HỢP LỆ
- Phải sử dụng hàm thư viện cho phép gán:
  - strcpy(aString, "Hello");
- Một hàm được xây dựng sẵn trong <cstring>
- Đặt giá trị của aString bằng với “Hello”
- KHÔNG kiểm tra kích thước!

Toán tử so sánh

- Không thể sử dụng toán tử ‘==“ với c-string

```
char aString[ 10] = “Hello”;
char anotherString[ 10] = “Goodbye”;
aString == anotherString; //Không hợp lệ
```

- Phải sử dụng thư viện hàm:

```
if (strcmp(aString, anotherString))
        cout <<“Strings are NOT same.";
else cout << "Strings are same.";
```

Một số hàm trong cstring

FUNCTION	DESCRIPTION	CAUTIONS
`strcpy`(*Target_String_Var*, *Src_String*)	Copies the C-string value *Src_String* into the C-string variable *Target_String_Var*.	Does not check to make sure *Target_String_Var* is large enough to hold the value *Src_String*.
`strcpy`(*Target_String_Var*, *Src_String*, *Limit*)	The same as the two-argument `strcpy` except that at most *Limit* characters are copied.	If *Limit* is chosen carefully, this is safer than the two-argument version of `strcpy`. Not implemented in all versions of C++.
`strcat`(*Target_String_Var*, *Src_String*)	Concatenates the C-string value *Src_String* onto the end of the C-string in the C-string variable *Target_String_Var*.	Does not check to see that *Target_String_Var* is large enough to hold the result of the concatenation.

Một số hàm trong cstring

FUNCTION	DESCRIPTION	CAUTIONS
`strcat(`*Target_String_Var*, *Src_String*, *Limit*`)`	The same as the two argument `strcat` except that at most *Limit* characters are appended.	If *Limit* is chosen carefully, this is safer than the two-argument version of `strcat`. Not implemented in all versions of C++.
`strlen(`*Src_String*`)`	Returns an integer equal to the length of *Src_String*. (The null character, `'\0'`, is not counted in the length.)	
`strcmp(`*String_1*, *String_2*`)`	Returns 0 if *String_1* and *String_2* are the same. Returns a value < 0 if *String_1* is less than *String_2*. Returns a value > 0 if *String_1* is greater than *String_2* (that is, returns a nonzero value if *String_1* and *String_2* are different). The order is lexicographic.	If *String_1* equals *String_2*, this function returns 0, which converts to `false`. Note that this is the reverse of what you might expect it to return when the strings are equal.
`strcmp(`*String_1*, *String_2*, *Limit*`)`	The same as the two-argument `strcat` except that at most *Limit* characters are compared.	If *Limit* is chosen carefully, this is safer than the two-argument version of `strcmp`. Not implemented in all versions of C++.

Một số hàm trong cstring

- Hàm get(): Đọc một ký tự một lần
- Hàm put(): Hiển thị một ký tự một lần
- Hàm putback(): Giảm vị trí hiện tại trong stream lùi về một ký tự
- Hàm peek(): Trả về ký tự tiếp theo, nhưng không loại bỏ nó khỏi luồng input
- Hàm ignore(): Bỏ qua input, cho đến khi gặp ký tự được chỉ định

Ví dụ:

cin.ignore(1000, "\n"); bỏ qua nhiều nhất 1000 kí tự cho đến khi gặp “\n”

Lớp string

- Được định nghĩa trong thư viện <string>

```
#include <string>
using namespace std;
```

- Biến string và các biểu thức được xử lý giống như những kiểu đơn giản khác
- Có thể gán, so sánh, cộng string s1, s2, s3;
  - s3 = sl + s2;
  - s3 = "Hello Mom!"
- Lưu ý: c-string "Hello Mom!" được tự động chuyển thành kiểu string!

NỘI DUNG

6.1 Nhắc lại về vector

6.2 Làm việc với c-string và string

6.3 Khuôn mẫu hàm

6.4 Khuôn mẫu lớp

Giới thiệu

- Hàm dưới đây in ra một mảng các phần tử số nguyên:

```
void printArray(int* array, int size)
{
  for ( int i = 0; i < size; i++ )
     cout << array[ i ] << ", ";
}
```

Giới thiệu

- Nếu chúng ta muốn in một mảng ký tự thì sao?

```
void printArray(char* array,int size)
{
  for ( int i = 0; i < size; i++ )
     cout << array[ i ] << ", ";
}
```

Giới thiệu

- Nếu chúng ta muốn in một mảng các phần tử thực thì sao?

```
void printArray(double* array,
                int size)
{
  for ( int i = 0; i < size; i++ )
     cout << array[ i ] << ", ";
}
```

Giới thiệu

- Bây giờ nếu chúng ta muốn thay đổi cách hàm in mảng. ví dụ:

```
1, 2, 3, 4, 5
```

to

1 - 2 - 3 - 4 - 5

Giới thiệu

- Bây giờ chúng ta xem xét lớp **`Array`** là lớp bao một mảng các số nguyên.

```
class Array {
    int* pArray;
    int size;
public:
    …
};
```

Giới thiệu

- Điều gì sẽ xảy ra nếu chúng ta muốn sử dụng một lớp **`Array`** bao bọc mảng các số thực?

```
class Array {
    double* pArray;
    int size;
public:
    …
};
```

Giới thiệu

- Điều gì sẽ xảy ra nếu chúng ta muốn sử dụng một lớp **`Array`** bao bọc mảng các bool?

```
class Array {
    bool* pArray;
    int size;
 public:
    …
 };
```

Giới thiệu

- Bây giờ nếu chúng ta muốn thêm một phương thức **`Sum`** vào lớp **`Array`**, chúng ta phải thay đổi cả ba lớp.

Generic Programming (GP)

- GP đề cập đến các chương trình chứa các trừu tượng tổng quát
- Một lập trình trừu tượng tổng quát (function, method, class) có thể được tham số hóa với một kiểu
- Sự trừu tượng như vậy có thể hoạt động với nhiều loại dữ liệu khác nhau

Ưu điểm

- Khả năng tái sử dụng
- Khả năng viết
- Khả năng bảo trì

Khuôn mẫu – Khuôn hình (Templates)

- Trong C ++, lập trình tổng quát được thực hiện bằng cách sử dụng khuôn mẫu (templates)
- Có 02 dạng:
  - Khuôn mẫu hàm (Function Templates)
  - Khuôn mẫu lớp (Class Templates)

- Trình biên dịch tạo ra các bản sao theo loại cụ thể khác nhau từ một khuôn mẫu duy nhất

KHUÔN MẪU (TEMPLATE)

- Cho phép các định nghĩa tổng quát cho hàm và lớp

- Tên kiểu làm tham số thay vì kiểu thực sự

- Định nghĩa chính xác được quyết định ở thời điểm chạy

KHUÔN MẪU HÀM

- Ví dụ hàm đổi chỗ 2 số:

```
void doi_cho(int& var1, int& var2)
{
int temp = var1;
var1 = var2;
var2 = temp;
}
```

Chỉ áp dụng cho các biến **kiểu int**

Nhưng phần mã lệnh làm việc với bất kỳ kiểu nào

KHUÔN MẪU HÀM

Có thể nạp chồng hàm cho kiểu char:

```
void doi_cho (char& var1, char& var2)
{
char temp = var1;
var1 = var2;
var2 = temp;
}
```

Lưu ý:

- Mã lệnh gần giống nhau
- Chỉ khác nhau về kiểu được sử dụng

CÚ PHÁP KHUÔN MẪU HÀM

• **Khuôn mẫu hàm** cho phép "hoán đổi giá trị" cho bất kỳ **kiểu** biến nào:

```
template <class T>
void doi_cho(T& varl, T& var2)
{
T temp = var1;
var1 = var2;
var2 = temp;
}
```

- Dòng đầu tiên là **tiền tố khuôn mẫu**:
- Báo cho trình biên dịch biết đằng sau là khuôn mẫu
- Và **T là một tham số kiểu**

TIỀN TỐ KHUÔN MẪU

template <class T>

- Ở đây, class nghĩa là kiểu, hoặc sự phân lớp
- Dễ bị nhầm lẫn với từ class được sử dụng rộng rãi khi định nghĩa lớp
  - C++ cho phép sử dụng từ khóa **"typename"** ở vị trí từ khóa class
  - Tuy nhiên trong mọi trường hợp nên sử dụng class
- T có thể được thay bằng bất kỳ kiểu nào: Kiểu được định nghĩa trước hoặc người dùng định nghĩa
- Trong thân định nghĩa hàm: T được sử dụng giống như một kiểu bất kỳ

CHIẾN LƯỢC ĐỊNH NGHĨA KHUÔN MẪU

- Phát triển hàm như thông thường
  - Sử dụng các kiểu dữ liệu thật
  - Hoàn thành việc gỡ lỗi hàm nguyên bản
- Sau đó chuyển đổi thành khuôn mẫu: Thay thế các tên kiểu bằng tham số kiểu khi cần
- Ưu điểm:
  - Dễ giải quyết trường hợp cụ thể
  - Tập trung vào thuật toán, thay vì cú pháp khuôn mẫu

ĐỊNH NGHĨA KHUÔN MẪU HÀM

- Khuôn mẫu hàm doi_cho() thực sự là một tập hợp các định nghĩa (Một định nghĩa cho mỗi kiểu có thể có)
- Bộ biên dịch chỉ phát sinh các định nghĩa khi được yêu cầu:
  - Với điều kiện bạn đã định nghĩa cho tất cả các kiểu
  - Viết một định nghĩa mà làm việc cho tất cả các kiểu có thể có

KHUÔN MẪU HÀM

Cú pháp:

template **<class** kieu_du_lieu**>**

kieu_du_lieu_tra_ve ten_ham(danh sach tham so)

{ // phan than ham }

Chú ý:

Khi khai báo có thể dùng typename thay cho từ khóa class

Example – Function Templates

- khuôn mẫu hàm sau in ra một mảng cho hầu hết mọi loại phần tử :

```
template< typename T >
void printArray( T* array, int size )
{
  for ( int i = 0; i < size; i++ )
     cout << array[ i ] << “, ”;
}
```

…Example – Function Templates

```
int main() {
  int iArray[5] = { 1, 2, 3, 4, 5 };
  printArray( iArray, 5 );
     // Instantiated for int[]

  char cArray[3] = { 'a', 'b', 'c' };
  printArray( cArray, 3 );
     // Instantiated for char[]
  return 0;
}
```

Kiểu tham số tường minh

- function template có thể không có bất kỳ tham số nào

```
template <typename T>
T getInput() {
    T x;
    cin >> x;
    return x;
}
```

Kiểu tham số tường minh

```
int main() {
  int x;
  x = getInput();          // Error!

  double y;
  y = getInput();          // Error!
}
```

Kiểu tham số tường minh

```
int main() {
  int x;
  x = getInput< int >();

  double y;
  y = getInput< double >();
}
```

Function Template – Ví dụ

Viết khuôn mẫu hàm tính trung bình cộng của 2 số

Viết hàm main() sử dụng khuôn mẫu cho 2 biến kiểu nguyên, 2 biến kiểu thực

User Specializations

- Template có thể không xử lý thành công với tất cả các kiểu
- Cần được cung cấp cho (các) kiểu cụ thể

Example – User Specializations

```
template< typename T >
bool isEqual( T x, T y ) {
  return ( x == y );
}
```

… Example – User Specializations

```
int main {
  isEqual( 5, 6 );   // OK
  isEqual( 7.5, 7.5 );   // OK
  char s1[]="abc";char s2[]="abc" ;
   cout<<isEqual( s1, s2 )<<endl;
     // Logical Error!
  return 0;
}
```

… Example – User Specializations

```
template< >
bool isEqual< char* >(
   char* x, char* y ) {
  return ( strcmp( x, y ) == 0 );
}
```

… Example – User Specializations

```
int main {
  isEqual( 5, 6 );
     // Target: general template
  isEqual( 7.5, 7.5 );
     // Target: general template

  isEqual( “abc”, “xyz” );
     // Target: user specialization
  return 0;
}
```

Đa kiểu tham số

```
template< typename T, typename U >
T my_cast( U u ) {
  return (T)u;
}

int main() {
  double d = 10.5674;
  int j = my_cast( d );     //Error
  int i = my_cast< int >( d );
  return 0;
}
```

Các kiểu do người dùng định nghĩa (User-Defined Types)

- Bên cạnh các kiểu nguyên thủy, các kiểu do người dùng xác định cũng có thể được chuyển làm đối số kiểu cho các mẫu
- Trình biên dịch thực hiện kiểm tra kiểu tĩnh để chẩn đoán lỗi kiểu

…User-Defined Types

- Xem xét lớp String không có nạp chồng toán tử “==“

```
class String {
  char* pStr;
  …
  // Operator “==“ not defined
};
```

… User-Defined Types

```
template< typename T >
bool isEqual( T x, T y ) {
  return ( x == y );
}

int main() {
  String s1 = “xyz”, s2 = “xyz”;
  isEqual( s1, s2 );// Error!
  return 0;
}
```

…User-Defined Types

```
class String {
  char* pStr;
  …
  friend bool operator ==(
     const String&, const String& );
  };
```

… User-Defined Types

```
bool operator ==( const String& x,
    const String& y ) {
  return strcmp(x.pStr, y.pStr) == 0;
}
```

… User-Defined Types

```
template< typename T >
bool isEqual( T x, T y ) {
  return ( x == y );
}

int main() {
  String s1 = “xyz”, s2 = “xyz”;
  isEqual( s1, s2 );// OK
  return 0;
}
```

Nạp chồng vs khuôn mẫu (Overloading vs Templates)

- Khác kiểu dữ liệu, hoạt động tương tự nhau

=> Cần nạp chồng

- Các kiểu dữ liệu khác nhau, hoạt động giống hệt nhau

=> Cần khuôn mẫu

Overloading vs Templates

- Toán tử ‘+’ là được nạp chồng với kiểu toán hạng khác nhau.
- Một khuôn mẫu hàm duy nhất có thể tính tổng của mảng nhiều kiểu dữ liệu

Overloading vs Templates

```
String operator +( const String& x,
    const String& y ) {
  String tmp;
  tmp.pStr = new char[strlen(x.pStr) +
             strlen(y.pStr) + 1 ];
  strcpy( tmp.pStr, x.pStr );
  strcat( tmp.pStr, y.pStr );
  return tmp;
}
```

Overloading vs Templates

```
String operator +( const char * str1,
    const String& y ) {
  String tmp;
  tmp.pStr = new char[ strlen(str1) +
             strlen(y.pStr) + 1 ];
  strcpy( tmp.pStr, str1 );
  strcat( tmp.pStr, y.pStr );
  return tmp;
}
```

Overloading vs Templates

```
template< class T >
T sum( T* array, int size ) {
  T sum = 0;

  for (int i = 0; i < size; i++)
     sum = sum + array[i];

  return sum;
}
```

Khuôn mẫu in mảng (Print an Array)

```
template< typename T >
void printArray( T* array, int size )
{
  for ( int i = 0; i < size; i++ )
     cout << array[ i ] << ", ";
}
```

Generic Algorithms

- Thuật toán này hoạt động tốt cho các mảng thuộc bất kỳ loại nào

- Chúng tôi có thể làm cho nó hoạt động cho bất kỳ vùng tổng quát chung nào hỗ trợ hai hoạt động
- Toán tử tham khảo (*)
  - Toán tử tăng dần (++)
  - Toán tử con trỏ (*)

…Generic Algorithms

```
template< typename P, typename T >
P find( P start, P beyond, const T& x ) {
  while ( start != beyond &&
                  *start != x )
          start++;
  return start;
}
```

…Generic Algorithms

```
int main() {
  int iArray[5];
  iArray[0] = 15;
  iArray[1] = 7;
  iArray[2] = 987;
  …
  int* found;
  found = find(iArray, iArray + 5, 7);
  return 0;
}
```

NỘI DUNG

6.1 Nhắc lại về vector

6.2 Làm việc với c-string và string

6.3 Khuôn mẫu hàm

6.4 Khuôn mẫu lớp

Khuôn mẫu lớp (Class Templates)

- Một khuôn mẫu lớp đơn cung cấp hàm hoạt động trên các kiểu dữ liệu khác nhau
- Tận dụng tái sử dung các lớp.
- Định nghĩa khuôn mẫu lớp như sau:
  - template< class T > class Xyz { … }; or
  - template< typename T > class Xyz { … };

CLASS TEMPLATE (KHUÔN MẪU LỚP)

- Lớp là sự mở rộng của cấu trúc dữ liệu. Lớp không chỉ lưu trữ dữ liệu mà cả các hàm (phương thức)
- Khai báo

```
template  <class kieu_du_lieu>
class  ten_lop
    {
    . . .
    };
```

VÍ DỤ VỀ CLASS TEMPLATE

```
template<class T>
class Pair
{
    public:
        Pair();
        Pair(T firstVal, T secondVal);
        void setFirst(T newVal);
        void setSecond(T newVal);
        T getFirst() const;
        T getSecond() const;
    private:
        T first;
        T second;
};
```

Khuôn mẫu lớp Pair

- Các đối tượng của lớp khuôn mẫu Pair có "cặp" giá trị kiểu T
- Ta có thể khai báo các đối tượng:

```
Pair<int> score;
Pair<char> seats;
```

- Sau đó có thể sử dụng các đối tượng giống như các đối tượng bất kỳ

Ví dụ sử dụng:

```
score.setFirst(3);
score.setSecond(0);
```

Các thành viên khuôn mẫu lớp

Chú ý: Nếu lớp C là một khuôn mẫu lớp, tất cả các hàm thành viên của nó đều là khuôn mẫu hàm

```
template<class T>
Kiểu dl trả về Tên_lớp <T>::Tên hàm(Danh sách tham số)
{
//Thân hàm
}
```

Ví dụ: Định nghĩa hàm tạo

```
template<class T>
 Pair <T>::Pair()
      {
      }
template<class T>
Pair <T>:: Pair(T firstValue, T secondValue)
         {
         first= firstValue;
           second=secondValue;
         }
```

Ví dụ: Định nghĩa hàm set

```
template<class T>
void Pair <T>::setFirst(T newValue)
        {
        first = newValue;
        }
template<class T>
void Pair <T>::setSecond(T newValue)
        {
        second = newValue;
        }
```

Ví dụ: Định nghĩa hàm get

```
template<class T>
T Pair <T>::getFirst( )
      {
         return first;
      }
template<class T>
T Pair <T>::getSecond( )
         {
            return second;
         }
```

Ví dụ: Hàm main()

```
int main()
{
    int fi,se;
    Pair <int> a(2,1),b;
    cout<<"Nhap vao 1 so nguyen: ";
    cin>>fi;
    cout<<"Nhap vao 1 so nguyen nua: ";
    cin>>se;
    b.setFirst(fi);
    b.setSecond(se);
    if(a.getFirst()==b.getFirst()&&a.getSecond()==b.getSecond())
        cout<<"Hai cap so la bang nhau.";
    else
        cout<<"Hai cap so khong bang nhau.";
    return 0;
}
```

Example – Class Template

- Lớp **`Vector`** có thể lưu trữ các phần tử dữ liệu của nhiều kiểu khác nhau

- Không dùng khuôn mẫu, chúng ta cần phân tách lớp Vector cho mỗi kiểu dữ liệu.

…Example – Class Template

```
class point {
  int x, y;
  public:
  point (int abs =0, int ord =0);
  void display();
  //...
};
```

…Example – Class Template

```
template <class T> class point {
  T x;T y;
  public:
  point (T abs=0, T ord=0);
  void display();
};
```

Bàn đến việc định nghĩa method của class template

- Trong lớp: đ/n như lớp thông thường
- Ngoài lớp: cần nhắc lại cho trình biên dịch C++ biết các tham số kiểu `template <class T>` và tên class được viết lại `point<T>`

…Example – Class Template

Trong lớp

```
template <class T> class point {
  T x;T y;
  public:
  point (T abs=0, T ord=0){
     x = abs;
     y = ord;
  }
  void display();
};
```

…Example – Class Template

```
#include <iostream.h>
//tạo khuôn mẫu lớp
template <class T> class point {
   T x, y;
   public:
   // Định nghĩa hàm thành phần bên trong khuôn mẫu lớp
   point(Tabs=0,Tord=0){
   x=abs;y=ord;
   }
   void display();
   };
   // Định nghĩa hàm thành phần bên ngoài khuôn mẫu lớp
   template <class T> void point<T>::display() {
   cout<<"Toa do: "<<x<<" "<<y<<"\n";
}
```

Sử dụng Class Template

- Một khi khuôn mẫu lớp `point` được định nghĩa, thì khai báo như sau :

```
point<int> ai;
```

khai báo một đối tượng `ai` có hai thành phần toạ độ là kiểu nguyên (**int**) . Điều đó có nghĩa là `point<int>` có vai trò như một kiểu dữ liệu lớp; người ta gọi nó là một lớp thể hiện của khuôn hình lớp `point`. Một cách tổng quát, khi áp dụng một kiểu dữ liệu nào đó với khuôn hình lớp `point` ta sẽ có được một lớp thể hiện tương ứng với kiểu dữ liệu. Như vậy:

```
point<double> ad;
```

định nghĩa một đối tượng `ad` có các toạ độ là số thực; còn với `point<double>` đóng vai trò một lớp và được gọi là một lớp thể hiện của khuôn hình lớp `point`.

…Example – Class Template

```
int main(){
  point<int> ai(3,5);
  ai.display();
  point<char> ac('d','y');
  ac.display();
  point<double> ad(3.5, 2.3);
  ad.display();
  return 0;
}
```

Các tham số khuôn mẫu lớp

- Có 2 loại:
  - Tham số kiểu
  - Tham số biểu thức
- Có nhiều điểm giống khuôn mẫu hàm nhưng các ràng buộc đối với các kiểu tham số lại khác nhau

Các tham số khuôn mẫu lớp

```
template <class T, class U, class V> //danh sách ba tham số kiểu
class try {
   T x;
   U t[5];
   ...
   V fm1 (int, U);
   ...
};
```

Các tham số khuôn mẫu lớp

Một lớp thể hiện được khai báo bằng cách liệt kê đằng sau tên khuôn hình lớp các tham số thực (là tên các kiểu dữ liệu) với số lượng bằng với số các tham số trong danh sách (`template<...>`) của khuôn hình lớp. Sau đây đưa ra một số ví dụ về lớp thể hiện của khuôn hình lớp try:

```
try <int, float, int> // lớp thể hiện với ba tham số int, float, int
try <int,int *, double>// lớp thể hiện với ba tham số int, int *, double
try <char *, int, obj> // lớp thể hiện với ba tham số char *, int, obj
```

Trong dòng cuối ta cuối giả định `obj` là một kiểu dữ liệu đã được định nghĩa trước đó. Thậm chí có thể sử dụng các lớp thể hiện để làm tham số thực cho các lớp thể hiện khác, chẳng hạn:

```
try <float, point<int>, double>
try <point<int>,point<float>, char *>
```

Các tham số khuôn mẫu lớp

```
try <float, point<int>, double>
try <point<int>,point<float>, char *>
```

Cần chú ý rằng, vấn đề tương ứng chính xác được nói tới trong các khuôn hình hàm không còn hiệu lực với các khuôn hình lớp. Với các khuôn hình hàm, việc sản sinh một thể hiện không chỉ dựa vào danh sách các tham số có trong `template<...>` mà còn dựa vào danh sách các tham số hình thức trong tiêu đề của hàm.

Các tham số khuôn mẫu lớp

Một tham số hình thức của một khuôn hình hàm có thể có kiểu, là một lớp thể hiện nào đó, chẳng hạn:

```
template <class T> void fct(point<T>) { ... }
```

Việc khởi tạo mới các kiểu dữ liệu mới vẫn áp dụng được trong các khuôn hình lớp. Một khuôn hình lớp có thể có các thành phần(dữ liệu hoặc hàm) **static**. Trong trường hợp này, cần phải biết rằng, mỗi thể hiện của lớp có một tập hợp các thành phần **static** của riêng mình:

Các tham số khuôn mẫu lớp

- Tham số biểu thức trong khuôn mẫu lớp: tham số thực tế tương ứng với tham số biểu thức phải là một hằng số
- Giả sử chúng ta định nghĩa lớp Arr để thao tác trên mảng chứa đối tượng có kiểu bất kỳ.
  - Tham số 1: tham số kiểu
  - Tham số 2: tham số biểu thức để xác định số lượng phần tử.

Tham số biểu thức

```
template <class T, int n> class Arr {
   T elements[n];
   public:
   ...
};
```

- Tham số kiểu được xác định bởi từ khóa class
- Tham số biểu thức kiểu int. Phải chỉ rõ giá trị trong khai báo các lớp thể hiện
  - Arr <int, 4>

```
class Arr {
   int elements[4];
   public:
   ...
};
```

Các tham số khuôn mẫu lớp

- Có thể khai báo tùy ý các tham số biểu thức trong khuôn mẫu hàm.
- Các tham số này có thể xuất hiện ở bất kỳ nơi nào trong định nghĩa khuôn mẫu lớp
- Khi cụ thể 1 lớp có tham số biểu thức, các tham số thực tế tương ứng truyền vào phải là hằng phù hợp với kiểu dữ liệu đã khai báo

Cụ thể hóa khuôn mẫu lớp

- khuôn mẫu lớp định nghĩa họ các lớp trong đó mỗi lớp chứa đồng thời định nghĩa của chính nó và các hàm thành phần
- Tất cả các hàm thành phần cùng tên sẽ được thực hiện theo cùng một giải thuật
- Nếu muốn 1 hàm thành phần thích ứng tình huống cụ thể nào đó có thể viết 1 đ/n khác

Ví dụ

```
#include <iostream>
#include <conio.h>
using namespace std;
template <class T>
class point {
      T x,y;
      public:
      point(T abs=0,T ord=0){ x=abs;y=ord;}
            void display();
};
template <class T>
void point<T>::display() {
            cout<<"Toa do: "<<x<<" "<<y<<"\n";
}
```

Ví dụ

```
void point<char>::display() {
        cout<<"Toa do: "<<(int)x<<" "<<(int)y<<"\n
}
void main() {
    clrscr();
    point <int> ai(3,5); ai.display();
    point <char> ac('d','y');
    ac.display();
    point <double> ad(3.5, 2.3);
    ad.display();
    getch();
}
```

```
Toa do: 3 5
Toa do: 100 121
Toa do: 3.5 2.3
```

Chú ý

Ta chú ý dòng tiêu đề trong khai báo một thành phần được cụ thể hoá:

```
void point<char>::display()
```

Khai báo này nhắc chương trình dịch sử dùng hàm này thay thế hàm `display()` của khuôn hình lớp `point` (trong trường hợp giá trị thực tế cho tham số kiểu là **char**).

Nhận xét

Nhận xét

(x) Có thể cụ thể hoá giá trị của tất cả các tham số. Xét khuôn hình lớp sau đây:

```
template <class T, int n> class table {
  T tab[n];
 public:
  table() {cout<<" Tao bang\n"; }
  ...
 };
```

Khi đó, chúng ta có thể viết một định nghĩa cụ thể hoá cho hàm thiết lập cho các bảng 10 phần tử kiểu `point` như sau:

```
table<point,10>::table(...) {...}
```

Nhận xét

- Xét lệnh gán giữa 02 đối tượng: hai lớp thể hiện tương ứng với cùng 1 kiểu nếu các tham số kiểu tương ứng nhau 1 cách chính xác và các tham số biểu thức có cùng giá trị (trừ kế thừa)

```
Arr <int, 12> a1;
Arr <float, 12> a2;
```

```
Arr <int, 12> a1;
Arr <int, 16> a2;
```

- Thì không được viết a2 = a1
- Kể cả khai báo