VIỆN ĐIỆN TỬ - VIỄN THÔNG

School of Electronics and telecommunications

# KỸ THUẬT LẬP TRÌNH C/C++

*Chương 5:*

## LẬP TRÌNH TỔNG QUÁT

Giảng viên: TS. Nguyễn Thị Kim Thoa

Học kỳ: 20211

Năm học: 2021-2022

# Nội dung

1. Khuôn mẫu lớp
2. Khuôn mẫu hàm
3. Thuật toán tổng quát

# 1. Khuôn mẫu hàm

## Ví dụ: hàm tìm giá trị lớn nhất

a. Tìm max hai số nguyên

```
int max(const int &a, const int &b){
        return (a > b)? a: b;
}
```

b. Tìm max hai số thực

```
float max(const float &a, const float &b){
        return (a > b)? a: b;
}
```

- Nhận xét: Các hàm tìm max của hai số chỉ khác nhau về kiểu dữ liệu, thuật toán giống nhau.
- Tương tự như vậy có rất nhiều hàm chỉ khác nhau về kiểu dữ liệu, không khác về thuật toán
- Giải pháp: tổng quát hóa các hàm chỉ khác nhau về kiểu $\Rightarrow$ khuôn mẫu hàm

# 1. Khuôn mẫu hàm [...]

## Ví dụ: hàm tìm giá trị lớn nhất

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

Tham số khuôn mẫu

Sử dụng từ khóa **class** hoặc **typename** để khai báo tham số khuôn mẫu

- *Khuôn mẫu hàm inline*

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

- *Sử dụng*

```
int max(5,7);
```

Compiler sẽ tạo một hàm theo khuôn mẫu có dạng *int max(const int&, const int&)*

# 1. Khuôn mẫu hàm [...]

## Ví dụ sử dụng

```
void main() {
    int i1 = 1, i2 = 5;
    double d1 = 1.0, d2 = 2.0;
    double d = max(d1,d2);// max<double>(double,double)
    char c = max('c','a');// max<char>(char, char)
    d = max(i1,d1);      // error: ambiguous
    c = max('c',i1);     // error: ambiguous
    d = max<double>(d1,i1);// OK: explicit qualification
    i2 = max<int>('c',i1);// OK: explicit qualification
//check lại c = max<int>('c',i1);// OK: explicit
qualification

}
```

# 1. Khuôn mẫu hàm [...]

## Áp dụng cho complex?

```
class complex{
      double real, imag;
public:
      complex(double r=0,int i=0);
      double get_real();
      void   set_real(double);
      double get_imag();
      void   set_imag(double);
};


void main{
      complex c1(1.1,2.0);
      complex c2(2.0,2.2);
      complex c = max(c1,c2);
};
```

*Lỗi, vì lớp* ***complex*** *trên chưa định nghĩa phép so sánh > sử dụng trong hàm khuôn mẫu* ***max***

# 1. Khuôn mẫu hàm [...]

- Giải pháp cho trường hợp trên là định nghĩa toán tử so sánh lớn hơn cho lớp complex.
- Một khuôn mẫu hàm cũng có thể được nạp chồng bởi hàm cùng tên hoặc bởi một khuôn mẫu hàm cùng tên (khác số lượng các tham số hoặc kiểu của ít nhất một tham số)
  - Hàm cùng tên để thực hiện cho các thuật toán đặc biệt. (Ví dụ, hàm max giữa hai chuỗi ký tự có thuật toán thực hiện khác với tìm max của hai số int hoặc double)
  - Khuôn mẫu hàm cùng tên

```
template <class T> T max(T a, T b, T c)   {...}
template <class T> T max(T* a, int n)     {...}
```

nhưng không được như thế này:

```
template <class T> T max(T a, T b, T c)   {...}

template <class X> X max(X a, X b, X c)   {...}
```

# 1. Khuôn mẫu hàm [...]

- Tham số khuôn mẫu hàm có thể có hơn một tham số kiểu, ví dụ:

```
template <class A, class B> void swap(A& a, B& b){...}
```

- Ví dụ:

```
template <class A,class B>
void swap( A &a, B &b){
        A temp = a;
      a = b; //đúng khi b tương thích với a
      b = temp; //đúng khi temp tương thích với b
}
void main(){
      int a = 5;
      double b = 10.2;
      swap(a,b); //swap<int,double>(int,double)
      swap(b,a); //swap<double,int>(double,int)
}
```

# Tóm lược về khuôn mẫu hàm

- Khi sử dụng compiler cần biết mã nguồn thực hiện khuôn mẫu hàm, do vậy khai báo và định nghĩa khuôn mẫu hàm nên để ở file header $\Rightarrow$ sử dụng template sẽ công khai hết phần thực hiện
- Mã hàm khuôn mẫu chỉ được thực sự sinh ra khi và chỉ khi khuôn mẫu hàm được sử dụng với kiểu cụ thể
- Một khuôn mẫu hàm được sử dụng nhiều lần với các kiểu khác nhau thì nhiều hàm khuôn mẫu được tạo ra
- Một khuôn mẫu hàm được sử dụng nhiều lần với cùng một kiểu, thì chỉ có một hàm khuôn mẫu được tạo

# Ưu/nhược điểm của khuôn mẫu hàm

- Ưu điểm
  - Tiết kiệm được mã nguồn
  - Tính mở: nâng cao tính sử dụng lại, thuật toán viết một lần sử dụng vô số lần
  - Cho phép xây dựng các thư viên chuẩn rất mạnh như các thư viện thuật toán thông dụng: sao chép, tìm kiếm, sắp xếp, lựa chọn,…
- Nhược điểm
  - Không che giấu được mã nguồn thực thi, vì compiler phải biết mã nguồn khi biên dịch
  - Theo dõi, tìm kiếm lỗi phức tạp, đôi khi lỗi nằm ở phần sử dụng nhưng compiler lại báo trong phần định nghĩa khuôn mẫu hàm

# 2. Khuôn mẫu lớp

## Nghiên cứu lớp số phức

```
#include <iostream.h>
class IntComplex{
    int real,imag;
public:
    Complex(int r = 0, int i =0): real(r),imag(i) {}
    int get_real() const { return real; }
    int get_imag() const { return imag; }
    Complex operator+(const Complex& b) const {
            Complex z(real + b.real, imag + b.image);
            return z;
    }
    Complex operator-(const Complex& b) const {
            return Complex(real - b.real, imag - b.imag);
    }
  ...
};
```

# 2. Khuôn mẫu lớp

## Nghiên cứu lớp số phức [...]

```
class DoubleComplex{
    double real,imag;
public:
    Complex(double r = 0, double i =0): real(r),imag(i)
{}
    double get_real() const { return real; }
    double get_imag() const { return imag; }
    Complex operator+(const Complex& b) const {
            Complex z(real + b.real, imag + b.image);
            return z;
    }
    Complex operator-(const Complex& b) const {
            return Complex(real - b.real, imag - b.imag);
    }
  ...
};
```

**Hai lớp số phức trên khác nhau gì? Giống nhau gì?**

# 2. Khuôn mẫu lớp

- Tương tự như vậy, trong thực tế có rất nhiều cấu trúc dữ liệu chỉ khác nhau về kiểu dữ liệu còn hoàn toàn giống về phép toán, ví dụ như vector, list, complex,...
- Để tiết kiệm mã nguồn thực thi ⇒ tổng quát hóa kiểu dữ liệu cho lớp

```
template <class T>
class Complex{
    T real,imag;
public:
    Complex(T r = 0, T i =0): real(r),imag(i) {}
    T get_real() const { return real; }
    T get_imag() const { return imag; }
    Complex operator+(const Complex& b) const {
            Complex z(real + b.real, imag + b.image);
            return z;
    }
    Complex operator-(const Complex& b) const;
  ...
};
```

# 2. Khuôn mẫu lớp

- Định nghĩa hàm thành viên bên ngoài khai báo lớp

```
template <class T>
Complex<T>::Complex operator-(const Complex<T>& b) const
{
            return Complex<T>(real - b.real, imag -
b.imag);
}
```

- Sử dụng

```
void main{
    Complex<int> c1(1,1), c2(2,3);
    Complex<int> c3 = c1+c2;
    Complex<double> c4(1.0,2.0), c5(3.0,5.0);
    Complex<double> c6 = c4 + c5;
};
```

# Tham số khuôn mẫu

- Tham số khuôn mẫu có thể là hằng hoặc kiểu (để làm tham số mặc định)

```
template <class T = int, class N = 10>
class Array{
    T data[N];
public:
    ...
};
```

Tham số mặc định

- Sử dụng

```
void main(){
    Array<int,10> a;
    Array<int>    b;
    Array<>       c;
}
```

Giống nhau

# Bài tập

- Xây dựng một khuôn mẫu hàm cho phép tìm giá trị lớn nhất, nhỏ nhất, trị tuyệt đối lớn nhất, trị tuyệt đối nhỏ nhất trong một mảng
- Tổng quát hóa kiểu dữ liệu cho lớp array, thực hiện các hàm thành viên cần thiết để có thể
  - Khai báo và khởi tạo giá trị ban đầu
  - Hủy bộ nhớ khi không còn sử dụng
  - Thực hiện các phép toán +,-, +=, -=,…

# 3. Thuật toán tổng quát

Ví dụ: hãy viết hàm sắp xếp các phần tử của một mảng theo thứ tự từ nhỏ đến lớn

```
void sort(int *p, int n){
     for(int i = 0; i < n-1; i++)
          for(int j = n-1; j > i; --j)
               if(p[j] < p[j-1])
                    swap(p[j], p[j-1]);
}

void swap(int &a, int &b){
     int t = a;
     a = b;
     b = t;
}
```

# 3. Thuật toán tổng quát

## Tổng quát hóa kiểu dữ liệu của các phần tử

```
template <class T>
void sort(T *p, int n){
     for(int i = 0; i < n-1; i++)
          for(int j = n-1; j > i; --j)
               if(p[j] < p[j-1])
                    swap(p[j], p[j-1]);
}
void swap(T &a, T &b){
     T t = a;
     a = b;
     b = t;
}
```

# 3. Thuật toán tổng quát

Thuật toán trên áp dụng cho nhiều kiểu dữ liệu có định nghĩa phép so sánh nhỏ hơn

```
int p[100];
sort(p,100);  //OK
char p2[100];
sort(p2,100); //OK
complex<double> p3[100];
sort(p3,100)  //Lỗi, chỉ áp dụng được khi lớp complex định
              //nghĩa phép so sánh nhỏ hơn
```

# 3. Thuật toán tổng quát

- Câu hỏi: làm thế nào để ta có thể sắp xếp lại từ lớn đến nhỏ mà không cần phải viết lại hàm
- Giải pháp 1: cho thêm tham biến vào khai báo hàm

# 3. Thuật toán tổng quát

Giải pháp 1: cho thêm tham biến vào khai báo hàm

```
enum comparetype {less, greater, abs_less, abs_greater};
template <class T>
void sort(T *p, int n,comparetype c){
     for(int i = 0; i < n-1; i++)
          for(int j = n-1; j > i; --j)
               switch(c){
                    case less:
                         if(p[j] < p[j-1])
                              swap(p[j], p[j-1]);
                         break;
                    case greater:
                         if(p[j] > p[j-1])
                              swap(p[j], p[j-1]);
                         break;
                    case abs_less:
                         if(abs(p[j]) < abs(p[j-1])
                              swap(p[j], p[j-1]);
                         break;
                    case abs_greater:
                         if(abs(p[j]) > abs(p[j-1]))
                              swap(p[j], p[j-1]);
                         break;
               }
}
```

# 3. Thuật toán tổng quát

- Nhược điểm của giải pháp 1:
  - Hiệu quả không cao
  - Tốc độ chậm
  - Không có tính năng mở: ví dụ nếu muốn só sánh số phức theo phần thực thì không dùng được cách trên
- Giải pháp 2: tổng quát hóa phép toán

```
template <class T, class Compare>
void sort(T *p, int n, Compare comp){
     for(int i = 0; i < n-1; i++)
          for(int j = n-1; j > i; --j)
               if(comp(p[j], p[j-1]))
                    swap(p[j], p[j-1]);
}
```

- Kiểu Compare có thể là
  - Một hàm
  - Một đối tượng thuộc lớp có định nghĩa lại toán tử gọi hàm

# 3. Thuật toán tổng quát

## Kiểu Compare là một hàm

```
template <class T>
inline bool less(const T &a, const T &b){
      return a < b; //return operator<(a,b)
}


template <class T>
inline bool greater(const T &a, const T &b){
      return a > b; //return operator>(a,b)
}
```

## Sử dụng

```
int v[100];
double d[100];
sort(v,100,less<int>);
sort(d,100,greater<double>)
```

*Một hàm*

# 3. Thuật toán tổng quát

## So sánh số phức

```
template <class T>
inline bool less_real(const complex &a, const complex &b){
     return (a.get_real() < b.get_real()) //return operator>(a,b)
}
```

## Sử dụng

```
complex<double> c[100];
Sort(c,100,less_real<double>);
```

# 3. Thuật toán tổng quát

**Kiểu compare là một lớp có định nghĩa lại toán tử gọi hàm**

```
template <class T>
struct Less {
      bool operator()(const T &a, const T &b){
            return a<b;
      }
};

template <class T>
struct Greater {
      bool operator()(const T &a, const T &b){
            return a>b;
      }
};
```

**Sử dụng**

```
int v[100];
double d[100];
sort(v,100,Less<int>());
sort(d,100,Greater<double>())
```

*Một đối tượng*

# @ Bài tập

- Áp dụng tổng quát hóa kiểu dữ liệu và tổng quát hóa phép toán để xây dựng các hàm cần thiết thực hiện các phép toán cộng, trừ, nhân, chia từng phần tử của hai mảng. Sau đó viết chương trình minh họa cách sử dụng.
- Gợi ý:
  - *Hàm thực hiện phép toán*

```
template <class T, class OP>
void operation(T *p1, T *p2, T *result,int n, OP op){...}
```

  - *Định nghĩa phép toán theo dạng hàm*

```
template <class T>
T add(const T &a, const T &b){...}
...
```

  - *Hoặc định nghĩa phép toán theo dạng đối tượng*

```
template <class T>
struct Add{...};
...
```