

# Template

Lập trình hướng đối tượng

## Tài liệu đọc

- n Eckel, Bruce. *Thinking in C++, 2<sup>nd</sup> Ed.* Vol. 1.
  - .. Chapter 16: Introduction to Templates
- n Dietel. *C++ How to Program, 4<sup>th</sup> Ed.*
  - .. Chapter 11: Templates



# Giới thiệu về khuôn mẫu

- n Giới thiệu
- n Lập trình tổng quát (generic programming)
- n Lập trình tổng quát trong C
- n C++ template
- n Khuôn mẫu hàm
- n Khuôn mẫu lớp
- n Các tham số template khác
- n Template sử dụng template



# Giới thiệu

- n Trong suốt khoá học, ta đã nói về phương pháp hướng đối tượng như là một cơ chế cho việc trừu tượng hoá
  - .. Nhóm các đối tượng có cùng tập hành vi và thuộc tính lại với nhau, và tương tác với chúng theo cùng kiểu
- n Với đa hình và thừa kế, ta có thể biểu diễn mối quan hệ giữa các lớp đối tượng tương tự và tương tác với chúng một cách thống nhất
  - .. Ta có thể lệnh cho một chiếc xe chạy bằng máy thực hiện việc "drive", và hành vi thích hợp sẽ được gọi tùy theo loại xe đang nói đến
- n Để kết thúc, ta sẽ giới thiệu một kiểu trừu tượng hoá khác



# Lập trình tổng quát

- n Ta đã tập trung vào việc trừu tượng hoá các chi tiết để tạo được các lớp đối tượng - gồm những thứ có cùng tập thuộc tính và hành vi
- n Mọi ý tưởng được bàn đến đều xoay quanh đối tượng - biểu diễn khái niệm hoặc biểu diễn trong C++
- n Bây giờ, ta nói về một phương pháp lập trình mà trừu tượng hoá chính các lớp đối tượng.
- n Đây là mức tiếp theo trong quá trình logic phát triển của trừu tượng hoá.



# Lập trình tổng quát

- n Lập trình tổng quát là phương pháp lập trình độc lập với chi tiết biểu diễn dữ liệu
  - Tư tưởng là ta định nghĩa một khái niệm không phụ thuộc một biểu diễn cụ thể nào, và sau đó mới chỉ ra kiểu dữ liệu thích hợp làm tham số
- n Qua các ví dụ, ta sẽ thấy đây là một phương pháp tự nhiên tuân theo khuôn mẫu hướng đối tượng theo nhiều kiểu

# Lập trình tổng quát

- n Ta đã quen với ý tưởng có một phương thức được định nghĩa sao cho khi sử dụng với các lớp khác nhau, nó sẽ đáp ứng một cách thích hợp
  - .. Khi nói về đa hình, nếu phương thức "draw" được gọi cho một đối tượng bất kỳ trong cây thừa kế Shape, định nghĩa tương ứng sẽ được gọi để đối tượng được vẽ đúng
  - .. Trong trường hợp này, mỗi hình đòi hỏi một định nghĩa phương thức hơi khác nhau để đảm bảo sẽ vẽ ra hình đúng
- n Nhưng nếu định nghĩa hàm cho các kiểu dữ liệu khác nhau nhưng không cần phải khác nhau thì sao?

# Lập trình tổng quát

- n Ví dụ, xét hàm sau:

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

- .. Hàm
- .. Nếu ta muốn thực hiện việc tương tự cho một kiểu dữ liệu khác, chẳng hạn float?

```
void swap(float& a, float& b) {  
    float temp;  
    temp = a; a = b; b = temp;  
}
```

- .. Có thực sự cần cần cả hai phiên bản không?

# Lập trình tổng quát

- n Ví dụ khác: ta định nghĩa một lớp biểu diễn cấu trúc ngăn xếp cho kiểu `int`

```
class Stack {  
    public:  
        Stack();  
        ~Stack();  
        void push(const int& i);  
        void pop(int& i);  
        bool isEmpty() const;  
        ...  
};
```

- n Ta thấy khai báo và định nghĩa của `Stack` phụ thuộc tại một mức độ nào đó vào kiểu dữ liệu `int`
  - .. Một số phương thức lấy tham số và trả về kiểu `int`
  - .. Nếu ta muốn tạo ngăn xếp cho một kiểu dữ liệu khác thì sao?
  - .. Ta có nên định nghĩa lại hoàn toàn lớp `Stack` (kết quả sẽ tạo ra nhiều lớp chẳng hạn `IntStack`, `FloatStack`, ...) hay không?

# Lập trình tổng quát

- n Ta thấy, trong một số trường hợp, đưa chi tiết về kiểu dữ liệu vào trong định nghĩa hàm hoặc lớp là điều không có lợi
  - .. Trong khi ta cần các định nghĩa khác nhau cho "draw" của `Point` hay `Circle`, vấn đề khác hẳn với trường hợp một hàm chỉ có nhiệm vụ hoán đổi hai giá trị
- n Thực ra, khái niệm lập trình tổng quát học theo sự sử dụng một phương pháp của lớp cơ sở cho các thể hiện của các lớp dẫn xuất
  - .. Ví dụ, trong cây thừa kế khi, ta muốn cùng một phương thức `eatBanana()` được thực thi, bất kể con trỏ/tham chiếu đang chỉ tới một `Monkey` hay `LazyMonkey`
- n Với lập trình tổng quát, ta tìm cách mở rộng sự trừu tượng hoá ra ngoài địa hạt của các cây thừa kế

# Lập trình tổng quát trong C

## Sử dụng trình tiền xử lý của C

- .. Trình tiền xử lý thực hiện thay thế text trước khi dịch
- .. Do đó, ta có thể dùng `#define` để chỉ ra kiểu dữ liệu và thay đổi tại chỗ khi cần

```
#define TYPE int

void swap(TYPE & a, TYPE & b) {
    TYPE temp;
    temp = a; a = b; b = temp;
}
```

Trình tiền xử lý sẽ thay mọi "TYPE" bằng "int" trước khi thực hiện biên dịch

## Hai hạn chế:

- .. nhầm chán và dễ lỗi
- .. chỉ cho phép đúng một định nghĩa trong một chương trình

# C++ template

- n Template (khuôn mẫu) là một cơ chế thay thế mã cho phép tạo các cấu trúc mà không phải chỉ rõ kiểu dữ liệu
- n Từ khoá template được dùng trong C++ để báo cho trình biên dịch rằng đoạn mã theo sau sẽ thao tác một hoặc nhiều kiểu dữ liệu chưa xác định
  - .. Từ khoá template được theo sau bởi một cặp ngoặc nhọn chứa tên của các kiểu dữ liệu tùy ý được cung cấp

```
template <typename T>
// Declaration that makes reference to a data type "T"
```

```
template <typename T, typename U>
// Declaration that makes reference to a data type "T"
// and a datatype "U"
```

Chú ý:  
Một lệnh template chỉ có hiệu quả đối với khai báo ngay sau nó

# C++ template

## n Hai loại khuôn mẫu cơ bản:

- Function template – khuôn mẫu hàm cho phép định nghĩa các hàm tổng quát dùng đến các kiểu dữ liệu tùy ý
- Class template – khuôn mẫu lớp cho phép định nghĩa các lớp tổng quát dùng đến các kiểu dữ liệu tùy ý

## n Ta sẽ mô tả từng loại trước khi đi bàn đến những phức tạp của lập trình khuôn mẫu

# Khuôn mẫu hàm

## n Khuôn mẫu hàm là dạng khuôn mẫu đơn giản nhất cho phép ta định nghĩa các hàm dùng đến các kiểu dữ liệu tùy ý

## n Định nghĩa hàm `swap()` bằng khuôn mẫu:

```
template <typename T>
void swap(T & a, T & b) {
    T temp;
    temp = a; a = b; b = temp;
}
```

## n Phiên bản trên tương tự như phiên bản `swap()` bằng C sử dụng `#define`, nhưng nó mạnh hơn nhiều

# Khuôn mẫu hàm

- n Thực chất, khi sử dụng template, ta đã định nghĩa một tập vô hạn các hàm chồng nhau với tên **swap()**
- n Để gọi một trong các phiên bản này, ta chỉ cần gọi nó với kiểu dữ liệu tương ứng

```
int x = 1, y = 2;  
float a = 1.1, b = 2.2;  
...  
swap(x, y); // Invokes int version of swap()  
swap(a, b); // Invokes float version of swap()
```

# Khuôn mẫu hàm

- n Chuyện gì xảy ra khi ta biên dịch mã?
  - Trước hết, sự thay thế "T" trong khai báo/định nghĩa hàm **swap()** không phải thay thế text đơn giản và cũng không được thực hiện bởi trình biên dịch
  - Việc chuyển phiên bản mẫu của **swap()** thành các cài đặt cụ thể cho **int** và **float** được thực hiện bởi trình biên dịch



# Khuôn mẫu hàm

- n Hãy xem xét hoạt động của trình biên dịch khi gặp lời gọi `swap()` thứ nhất (với hai tham số `int`)
  - .. Trước hết, trình biên dịch tìm xem có một hàm `swap()` được khai báo với 2 tham số kiểu `int` hay không
    - n Nó không tìm thấy một hàm thích hợp, nhưng tìm thấy một template có thể dùng được
  - .. Tiếp theo, nó xem xét khai báo của template `swap()` để xem có thể khớp được với lời gọi hàm hay không
    - n Lời gọi hàm cung cấp hai tham số thuộc cùng một kiểu (`int`)
    - n Trình biên dịch thấy template chỉ ra hai tham số thuộc cùng kiểu `T`, nên nó kết luận rằng `T` phải là kiểu `int`
    - n Do đó, trình biên dịch kết luận rằng template khớp với lời gọi hàm

# Khuôn mẫu hàm

- n Khi đã xác định được template khớp với lời gọi hàm, trình biên dịch kiểm tra xem đã có một phiên bản của `swap()` với hai tham số kiểu `int` được sinh ra từ template hay chưa
  - .. Nếu đã có, lời gọi được liên kết (bind) với phiên bản đã được sinh (lưu ý: khái niệm liên kết này giống với khái niệm ta đã nói đến trong đa hình tĩnh)
  - .. Nếu không, trình biên dịch sẽ sinh một cài đặt của `swap()` lấy hai tham số kiểu `int` (thực ra là viết đoạn mã mà ta sẽ tạo nếu ta tự mình viết) – và liên kết lời gọi hàm với phiên bản vừa sinh.

# Khuôn mẫu hàm

- n Vậy, đến cuối quy trình biên dịch đoạn mã trong ví dụ, sẽ có hai phiên bản của `swap()` được tạo (một cho hai tham số kiểu `int`, một cho hai tham số kiểu `float`) với các lời gọi hàm của ta được liên kết với phiên bản thích hợp
  - .. Vậy, ta có thể đoán rằng có chi phí phụ về thời gian biên dịch đối với việc sử dụng template
  - .. Ngoài ra còn có chi phí phụ về không gian liên quan đến mỗi cài đặt của `swap()` được tạo trong khi biên dịch
  - .. Tuy nhiên, tính hiệu quả của các cài đặt đó cũng không khác với khi ta tự cài đặt chúng.

# Khuôn mẫu hàm

- n Cần ghi nhớ rằng tuy trình biên dịch đã tạo các phiên bản của `swap()` cho các tham số `int` và `float`, không tồn tại các hàm `swap(int,int)` hay `swap(float, float)`
  - .. Thay vào đó, có một hàm `swap<>()` được dùng để tạo hai hàm `swap<int>()` và `swap<float>()`
- n Khi được dùng với một cấu trúc template, cặp ngoặc nhọn được dùng để chỉ rõ kiểu dữ liệu cần đến
- n Thực tế, ta có thể sửa đoạn mã trước để gọi các hàm trên một cách tường minh:

```
int x = 1, y = 2;
float a = 1.1, b = 2.2;
...
swap<int>(x, y);    // Invokes int version of Swap()
swap<float>(a, b); // Invokes float version of Swap()
```

# Khuôn mẫu lớp

- n Tương tự với khuôn mẫu hàm với tham số thuộc các kiểu tùy ý, ta cũng có thể định nghĩa khuôn mẫu lớp (class template) sử dụng các thể hiện của một hoặc nhiều kiểu dữ liệu tùy ý
  - .. Ta cũng có thể định nghĩa template cho `struct` và `union`
- n Khai báo một khuôn mẫu lớp cũng tương tự với khuôn mẫu hàm

# Khuôn mẫu lớp

Ví dụ, ta sẽ tạo một cấu trúc cặp đôi giữ một cặp giá trị thuộc kiểu tùy ý

- n Trước hết, xét khai báo `Pair` cho một cặp giá trị kiểu `int`:
- n Ta có thể sửa khai báo trên thành một khuôn mẫu lấy kiểu tùy ý: Tuy nhiên hai thành viên `first` và `second` phải thuộc cùng kiểu
- n Hoặc ta có thể cho phép hai thành viên nhận các kiểu dữ liệu khác nhau:

```
struct Pair {  
    int first;  
    int second;  
};
```

```
template <typename T>  
struct Pair {  
    T first;  
    T second;  
};
```

```
template <typename T, typename U>  
struct Pair {  
    T first;  
    U second;  
};
```

# Khuôn mẫu lớp

- n Để tạo các thể hiện của template Pair, ta phải dùng ký hiệu cặp ngoặc nhọn
  - .. Khác với khuôn mẫu hàm khi ta có thể bỏ qua kiểu dữ liệu cho các tham số, đối với khuôn mẫu class/struct/union, chúng phải được cung cấp tường minh

```
Pair p; // Not permitted
Pair<int, int> q; // Creates a pair of ints
Pair<int, float> r; // Creates a pair with an int and a float
```

- n Tại sao đòi hỏi kiểu tường minh?
  - .. Các lệnh trên làm gì? - cấp phát bộ nhớ cho đối tượng
  - .. Nếu không biết các kiểu dữ liệu được sử dụng, trình biên dịch làm thế nào để biết cần đến bao nhiêu bộ nhớ?

# Khuôn mẫu lớp

- n Cũng như khuôn mẫu hàm, không có struct Pair mà chỉ có các struct có tên Pair<int, int>, Pair<int, float>, Pair<int, char>, ...
- n Quy trình tạo các phiên bản struct Pair từ khuôn mẫu cũng giống như đối với khuôn mẫu hàm
- n Khi trình biên dịch lần đầu gặp khai báo dùng Pair<int, int>, nó kiểm tra xem struct đó đã tồn tại chưa, nếu chưa, nó sinh một khai báo tương ứng.
  - .. Đối với các khuôn mẫu cho class, trình biên dịch sẽ sinh cả các định nghĩa phương thức cần thiết để khớp với khai báo class.

# Khuôn mẫu lớp

- n Một khi đã tạo được một thể hiện của một khuôn mẫu class/struct/union, ta có thể tương tác với nó như thể nó là thể hiện của một class/struct/union thông thường.

```
Pair<int, int> q;  
Pair<int, float> r;  
q.first = 5;  
q.second = 10;  
r.first = 15;  
r.second = 2.5;
```

- n Tiếp theo, ta sẽ tạo một template cho lớp Stack đã được mô tả trong các slice trước

# Khuôn mẫu lớp

- n Khi thiết kế khuôn mẫu (cho lớp hoặc hàm), thông thường, ta nên tạo một phiên bản cụ thể trước, sau đó mới chuyển nó thành một template
  - Ví dụ, ta sẽ bắt đầu bằng việc cài đặt hoàn chỉnh Stack cho số nguyên
- n Điều đó cho phép phát hiện các vấn đề về khái niệm trước khi chuyển thành phiên bản cho sử dụng tổng quát
  - khi đó, ta có thể test tương đối đầy đủ lớp Stack cho số nguyên để tìm các lỗi tổng quát mà không phải quan tâm đến các vấn đề liên quan đến template

# Stack cho số nguyên

## n Khai báo và định nghĩa lớp **Stack** cho kiểu **int**

- Bắt đầu bằng một ngăn xếp đơn giản

```
class Stack {
public:
    Stack();
    ~Stack();
    void push(const int& i) throw (logic_error);
    void pop(int& i) throw (logic_error);
    bool isEmpty() const;
    bool isFull() const;
private:
    static const int max = 10;
    int contents[max];
    int current;
};
```

```
Stack::Stack() { this->current = 0; }

Stack::~~Stack() {}

void Stack::push(const int& i) throw(logic_error) {
    if (this->current < this->max) {
        this->contents[this->current++] = i;
    }
    else {
        throw logic_error("Stack is full.");
    }
}

void Stack::pop(int& i) throw(logic_error) {
    if (this->current > 0) {
        i = this->contents[--this->current];
    } else {
        throw logic_error("Stack is empty.");
    }
}

bool Stack::isEmpty() const { return (this->current == 0;) }

bool Stack::isFull() const { return (this->current == this->max); }
```

# Template Stack

- n Chuyển khai báo và định nghĩa trước thành một phiên bản tổng quát:

```
template <typename T>
class Stack {
public:
    Stack();
    ~Stack();
    void push(const T& i) throw (logic_error);
    void pop(T& i) throw (logic_error);
    bool isEmpty() const;
    bool isFull() const;
private:
    static const int max = 10;
    T contents[max];
    int current;
};
```

Thêm lệnh **template** để nói rằng một phần của kiểu sẽ được chỉ rõ sau

```
template <typename T>
Stack<T>::Stack() { this->current = 0; }
```

Mỗi phương thức cần một lệnh **template** đặt trước

```
template <typename T>
Stack<T>::~~Stack() {}
```

```
template <typename T>
void Stack<T>::push(const T& i) {
    if (this->current < this->max) {
        this->contents[this->current++] = i;
    }
    else {
        throw logic_error("Stack is full.");
    }
}
```

Mỗi khi dùng toán tử phạm vi, cần một ký hiệu ngoặc nhọn kèm theo tên kiểu

Ta đang định nghĩa một lớp **Stack<type>**, chứ không định nghĩa lớp **Stack**

```
template <typename T>
void Stack<T>::pop(T& i) {
    if (this->current > 0) {
        i = this->contents[--this->current];
    } else {
        throw logic_error("Stack is empty.");
    }
}
```

Thay thế kiểu của đối tượng được lưu trong ngăn xếp (trước là **int**) bằng kiểu tùy ý **T**

```
template <typename T>
bool Stack<T>::isEmpty() const { return (this->current == 0); }
```

```
template <typename T>
bool Stack<T>::isFull() const { return (this->current == this->max); }
```



# Template Stack

- n Sau đó, ta có thể tạo và sử dụng các thể hiện của các lớp được định nghĩa bởi template của ta:

```
int x = 5, y;  
char c = 'a', d;  
  
Stack<int> s;  
Stack<char> t;  
  
s.push(x);  
t.push(c);  
s.pop(y);  
t.pop(d);
```



## Các tham số khuôn mẫu khác

- n Ta mới nói đến các lệnh template với tham số thuộc "kiểu" typename
- n Tuy nhiên, còn có hai "kiểu" tham số khác
  - .. Kiểu thực sự (ví dụ: `int`)
  - .. Các template



# Các tham số khuôn mẫu khác

- n Nhớ lại rằng trong cài đặt `Stack`, ta có một hằng `max` quy định số lượng tối đa các đối tượng mà ngăn xếp có thể chứa
  - .. Như vậy, mỗi thể hiện sẽ có cùng kích thước đối với mọi kiểu của đối tượng được chứa
- n Nếu ta không muốn đòi hỏi mọi `Stack` đều có kích thước tối đa như nhau?
- n Ta có thể thêm một tham số vào lệnh `template` chỉ ra một số `int` (giá trị này sẽ được dùng để xác định giá trị cho `max`)

```
template <typename T, int I>
// Specifies that one arbitrary type T and one int I
// will be parameters in the following statement
```

- .. Lưu ý: ta khai báo tham số `int` giống như trong các khai báo khác

# Các tham số khuôn mẫu khác

- n Sửa khai báo và định nghĩa trước để sử dụng tham số mới:

```
template <typename T, int I>
class Stack {
public:
    Stack();
    ~Stack();
    void push(const T& i) throw (logic_error);
    void pop(T& i) throw (logic_error);
    bool isEmpty() const;
    bool isFull() const;
private:
    static const int max = I;
    T contents[max];
    int current;
};
```

Khai báo tham số mới

Sử dụng tham số mới để xác định giá trị `max` của một lớp thuộc một kiểu nào đó

# Các tham số khuôn mẫu khác

```
template <typename T, int I>
Stack<T, I>::Stack() { this->current = 0; }

template <typename T, int I>
Stack<T, I>::~~Stack() {}

template <typename T, int I>
void Stack<T, I>::push(const T& i) {
    if (this->current < this->max) {
        this->contents[this->current++] = i;
    }
    else {
        throw logic_error("Stack is full.");
    }
}
...
```

Sửa tên lớp dùng cho các toán tử phạm vi

Sửa các lệnh template

# Các tham số khuôn mẫu khác

- n Giờ ta có thể tạo các thể hiện của các lớp Stack với các kiểu dữ liệu và kích thước đa dạng

```
Stack<int, 5> s; // Creates an instance of a Stack
                // class of ints with max = 5
Stack<int, 10> t; // Creates an instance of a Stack
                // class of ints with max = 10
Stack<char, 5> u; // Creates an instance of a Stack
                // class of chars with max = 5
```

- Lưu ý rằng các lệnh trên tạo thể hiện của 3 lớp khác nhau



# Các tham số khuôn mẫu khác

- n Các ràng buộc khi sử dụng các kiểu thực sự làm tham số cho lệnh template:
  - Chỉ có thể dùng các kiểu số nguyên, con trỏ, hoặc tham chiếu
  - Không được gán trị cho tham số hoặc lấy địa chỉ của tham số



# Các tham số khuôn mẫu khác

- n Loại tham số thứ ba cho lệnh template chính là một template
- n Ví dụ, xét thiết kế khuôn mẫu cho một lớp Map (ánh xạ ánh xạ các khoá tới các giá trị)
  - Lớp này cần lưu các ánh xạ từ khoá tới giá trị, nhưng ta không muốn chỉ ra kiểu của các đối tượng được lưu trữ ngay từ đầu
  - Ta sẽ tạo Map là một khuôn mẫu sao cho có thể sử dụng các kiểu khác nhau cho khoá và giá trị
  - Tuy nhiên, ta cần chỉ ra lớp chứa (container) là một template, để nó có thể lưu trữ các khoá và giá trị là các kiểu tùy ý

# Các tham số khuôn mẫu khác

- n Ta có thể khai báo lớp Map:

```
template< typename Key, typename Value,  
         template <typename T> Container>  
class Map {  
    ...  
private:  
    Container<Key> keys;  
    Container<Value> values;  
    ...  
};
```

- n Sau đó có thể tạo các thể hiện của Map như sau:

```
Map< string, int, Stack> wordcount;
```

- .. Lệnh trên tạo một thể hiện của lớp Map<string, int, Stack> chứa các thành viên là một tập các string và một tập các int (giả sử còn có các đoạn mã thực hiện ánh xạ mỗi từ tới một số int biểu diễn số lần xuất hiện của từ đó)
- .. Ta đã dùng template Stack để làm container lưu trữ các thông tin trên

# Các tham số khuôn mẫu khác

- .. Như vậy, khi trình biên dịch sinh các khai báo và định nghĩa thực sự cho các lớp Map, nó sẽ đọc các tham số mô tả các thành viên dữ liệu
- .. Khi đó, nó sẽ sử dụng khuôn mẫu Stack để sinh mã cho hai lớp Stack<string> và Stack<int>
- .. Đến đây, ta phải hiểu rõ tại sao container phải là một khuôn mẫu, nếu không, làm thế nào để có thể dùng nó để tạo các loại stack khác nhau?