TEMPLATE

GENETIC PROGRAMMING

- Lập trình tổng quát là phương pháp lập trình độc lập với chi tiết biểu diễn dữ liệu:
  - Tư tưởng là ta định nghĩa 1 khái niệm không phụ thuộc 1 biểu diễn cụ thể nào, và sau đó mới chỉ ra kiểu dữ liệu cụ thể nào, và sau đó mới chỉ ra kiểu dữ liệu thích hợp làm tham số.
- Qua các ví dụ, ta sẽ thấy đây là 1 phương pháp tự nhiên tuân theo khuôn mẫu hướng đối tượng theo nhiều kiểu.

GENETIC PROGRAMMING

- Ta đã quen với ý tưởng có 1 phương thức được định nghĩa sao cho khi sử dụng với các lớp khác nhau, nó sẽ đáp ứng 1 cách thích hợp:
  - Khi nói về đa hình, nếu phương thức "draw" được gọi cho 1 đối tượng bất kỳ trong cây kế thừa Shape, định nghĩa tương ứng sẽ được gọi để đối tượng được vẽ đúng;
  - Trong trường hợp này, mỗi hình đòi hỏi 1 định nghĩa phương thức hơi khác nhau để đảm bảo sẽ vẽ ra hình đúng.
- Nhưng nếu định nghĩa hàm cho các kiểu dữ liệu khác nhau nhưng không cần phải khác nhau thì sao?

GENETIC PROGRAMMING

- Ví dụ: xét hàm sau

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

  - Hàm trên chỉ cần hoán đổi giá trị chứa trong 2 biến int;
  - Nếu ta muốn thực hiện việc tương tự cho 1 kiểu dữ liệu khác, chẳng hạn float?
  - Có thực sự cần đến 2 phiên bản không?

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

GENETIC PROGRAMMING

- Ví dụ: ta định nghĩa 1 lớp biểu diễn cấu trúc ngăn xếp cho kiểu int;
- Ta thấy khai báo & định nghĩa của Stack phụ thuộc tại 1 mức độ nào đó vào kiểu dữ liệu int:
  - Một số phương thức lấy tham số & trả về kiểu int;
  - Nếu ta muốn tạo ngăn xếp cho 1 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?

GENETIC PROGRAMMING

- Ví dụ: ta định nghĩa 1 lớp biểu diễn cấu trúc ngăn xếp cho kiểu int;
- Ta thấy khai báo & định nghĩa của Stack phụ thuộc tại 1 mức độ nào đó vào kiểu dữ liệu int:
  - Một số phương thức lấy tham số & trả về kiểu int;
  - Nếu ta muốn tạo ngăn xếp cho 1 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?

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

GENETIC PROGRAMMING

- Ta thấy, trong 1 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 trường hợp 1 hàm chỉ có nhiệm vụ hoán đổi 2 giá trị.
- Thực ra, khái niệm lập trình tổng quát học theo sự sử dụng 1 phương pháp của lớp cơ sở cho các thể hiện của các lớp dẫn xuất:
  - Ví dụ: trong cây thừa kế, ta muốn cùng 1 phương thức eatBanana() được thực thi, bất kể con trỏ /tham chiếu đang chỉ tới 1 Monkey hay LazyMonkey.
- Với lập trình tổng quát, ta tìm cách mở rộng sự trừu tượng hóa ra ngoài địa hạt của các cây thừa kế.

GENETIC PROGRAMMING

- Sử dụng tiền xử lý của C:
  - Tiền xử lý thực hiện thay thế text trước khi dịch;
  - Do đó, có thể dùng #define để chỉ ra kiểu dữ liệu & 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

- Hạn chế:
  - Nhàm chán & dễ lỗi;
  - Chỉ cho phép đúng 1 định nghĩa trong 1 chương trình.

C++ TEMPLATE

- Template là 1 cơ chế thay thế mã cho phép tạo các cấu trúc mà không phải chỉ rõ kiểu dữ liệu;
- Từ khóa template được dùng trong C++ để báo cho trình biên dịch rằng đoạn mã theo sau sẽ thao tác 1 hoặc nhiều kiểu dữ liệu chưa xác định:
  - Từ khóa template được theo sau bởi 1 cặp <> 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 data type "U"
```

  - Chú ý: 1 lệnh template chỉ có hiệu quả đối với khai báo **ngay sau** nó.

FUNCTION TEMPLATE

- Function template: khuôn mẫu hàm cho phép định nghĩa các hàm tổng quát dùng đến các kiểu dữ liệu tùy ý;
- Định nghĩa hàm HV() bằng khuôn mẫu hàm:

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

- Phiên bản trên khá giống với phiên bản swap() bằng C sử dụng #define, nhưng nó mạnh hơn nhiều.

FUNCTION TEMPLATE

- Thực chất, khi sử dụng template, ta đã định nghĩa 1 tập hợp vô hạn các hàm chồng nhau với tên **HV()**;
- Để gọi 1 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;
HV(x, y);    //Invokes int version of HV()
HV(a, b);    //Invokes float version of HV()
```

- Khi biên dịch mã:
  - Sự thay thế "**T**" trong khai báo/định nghĩa hàm **HV()** không phải thay thế text đơn giản & cũng không được thực hiện bởi tiền xử lý;
  - Việc chuyển phiên bản mẫu **HV()** thành các cài đặt cụ thể cho **int** & **float** được thực hiện bởi trình biên dịch.

FUNCTION TEMPLATE

- Hoạt động của trình biên dịch khi gặp lời gọi hàm **HV(int, int)**:
  - Trình biên dịch tìm xem có 1 hàm **HV()** được khai báo với 2 tham số kiểu **int** hay không:
    - Nó không tìm thấy 1 hàm thích hợp, nhưng tìm thấy 1 template có thể dùng được.
  - Tiếp theo, nó xem xét khai báo của template HV() để xem có thể khớp được với lời gọi hàm hay không:
    - Lời gọi hàm cung cấp 2 tham số thuộc cùng 1 kiểu **int**;
    - Trình biên dịch thấy template chỉ ra 2 tham số thuộc cùng kiểu T, nên nó kết luận rang T phải là kiểu int;
    - Do đó, trình biên dịch kết luận rang template khớp với lời gọi hàm.

FUNCTION TEMPLATE

- Khi đã xác định được template khớp với lời gọi hàm, trình biên dịch kiểm tra xem đã có 1 phiên bản của HV() với 2 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 1 cài đặt HV() lấy 2 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.

FUNCTION TEMPLATE

- Vậy, đến cuối quy trình biên dịch đoạn mã trong ví dụ, sẽ có 2 phiên bản của HV() được tạo (1 cho 2 tham số kiểu int, 1 cho 2 tham số kiểu float) với các lời gọi hàm của ta được liên kết với phiên bản thích hợp:
  - Có chi phí phụ về thời gian biên dịch đối với việc sử dụng template;
  - Có chi phí phụ về không gian liên quan đến mỗi cài đặt của HV() 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.

FUNCTION TEMPLATE

- Vậy, đến cuối quy trình biên dịch đoạn mã trong ví dụ, sẽ có 2 phiên bản của HV() được tạo (1 cho 2 tham số kiểu int, 1 cho 2 tham số kiểu float) với các lời gọi hàm của ta được liên kết với phiên bản thích hợp:
  - Có chi phí phụ về thời gian biên dịch đối với việc sử dụng template;
  - Có chi phí phụ về không gian liên quan đến mỗi cài đặt của HV() 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.

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

CLASS TEMPLATE

- Class template: khuôn mẫu lớp cho phép sử dụng các thể hiện của 1 hoặc nhiều kiểu dữ liệu tùy ý;
  - Có thể định nghĩa class template cho struct & union.
- Khai báo class template tương tự như function template:
  - Ví dụ: tao 1 struct Pair như sau:
    - Khai báo Pair cho 1 cặp giá trị kiểu int;
    - Ta có thể sửa khai báo trên thanh 1 khuôn mẫu lấy kiểu tùy ý. Tuy nhiên 2 thành viên first & second phải thuộc cùng kiểu;
    - Hoặc ta có thể cho phép 2 thành viên nhận các kiểu dữ liệu khác nhau.

CLASS TEMPLATE

- Class template: khuôn mẫu lớp cho phép sử dụng các thể hiện của 1 hoặc nhiều kiểu dữ liệu tùy ý;
  - Ví dụ:

```
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;
};
```

CLASS TEMPLATE

- Để tạo các thể hiện của template Pair, ta phải dùng ký hiệu cặp ngoặc nhọn:
  - Khác với function template khi ta có thể bỏ qua kiểu dữ liệu cho các tham số , đối với class template 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 & a float
```

- Tại sao đòi hỏi kiểu tường minh?
  - 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?

CLASS TEMPLATE

- Cũng như function template, không có struct Pair mà chỉ có các struct có tên Pair<int, int>, Pair<int, float>, Pair<int, char>, ...
- Quy trình tạo các phiên bản struct Pair từ class template cũng giống như đối với function template;
- Khi trình biên dịch lần đầu gặp khai báo dùng Pair<int, int>, nó sẽ kiểm tra xem struct đó đã tồn tại chưa, nếu chưa, nó sinh ra 1 khai báo tương ứng:
  - Đối với các class template cho class, trình biên dịch sẽ sinh cả các định nghĩa phương thức cần thiết để khớp với khai báo class.

CLASS TEMPLATE

- Khi đã tạo được 1 thể hiện của 1 class template, ta có thể tương tác với nó như thể nó là thể hiện của 1 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;
```

- Tiếp theo, ta sẽ tạo 1 class template cho lớp Stack:

CLASS TEMPLATE

- Khi thiết kế class template thường ta tạo nên 1 phiên bản cụ thể trước, sau đó mới chuyển nó thành 1 template:
  - Ví dụ: cài đặt hoàn chỉnh lớp Stack cho số nguyên.
- Điều đó cho phép phát hiện các vấn đề về khái niệm trước khi chuyển thành phiên bản tổng quát:
  - Khi đó, ta có thể test tương đối đầu 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.

CLASS TEMPLATE

- Khai báo định nghĩa lớp Stack cho số nguyên:

```
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;
};
```

CLASS TEMPLATE

- Khai báo định nghĩa lớp Stack cho số nguyên:

```
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);    }
```

CLASS TEMPLATE

- Template Stack:

```
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;
};
```

CLASS TEMPLATE

- Template Stack:

```
template <typename T>
Stack<T>::Stack(){
   this->current = 0;
}
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 phương thức cần một lệnh **template** đặt trước

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**

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

CLASS TEMPLATE

- Template 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.");
   }
}
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);
}
```

CLASS TEMPLATE

- Template Stack:
  - Sau đó, ta có thể tạo & sử dụng các thể hiện của các lớp đã được định nghĩa bởi template:

```
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 CỦA TEMPLATE

- Ta mới nói đến các lệnh template với tham số thuộc kiểu **typename**;
- Tuy nhiên, còn có 2 kiểu tham số khác:
  - Kiểu thực sự (int, float, ...);
  - Các template khác.

CÁC THAM SỐ CỦA TEMPLATE

- Trong cài đặt template Stack, ta có 1 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.
- Nếu ta không muốn đòi hỏi mọi Stack đều có kích thước tối đa như nhau?
- Ta có thể thêm 1 tham số vào lệnh template chỉ ra 1 số int (giá trị này sẽ được dùng để xác định giá trị cho max): **template <typename T, int I>**
- Lưu ý: ta khai báo tham số int giống như trong các khai báo khác

CÁC THAM SỐ CỦA TEMPLATE

- Sửa khai báo & định nghĩa template Stack:

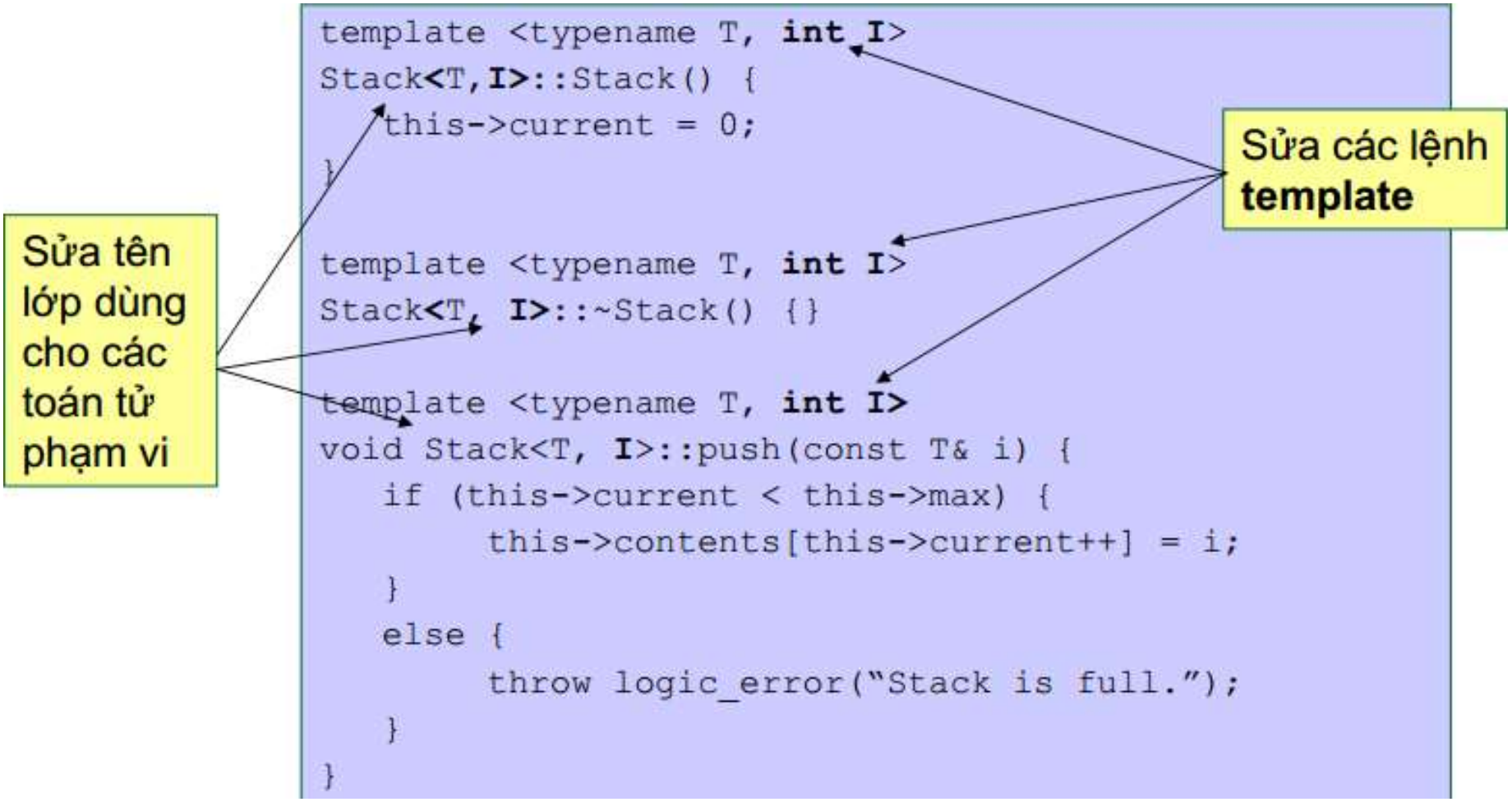
```
template <typename T, int I>
class Stack {
   public:
       Stack();
       ~Stack();
       void push(const T& i) throw (logic_error);
       void pop(T& i) throw (logic_error);
       bool isEmpty() const;
       bool isFull() const;
   private:
       static const int max = I;
       T contents[max];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

- Sửa khai báo & định nghĩa template Stack:



CÁC THAM SỐ CỦA TEMPLATE

- Ta có thể tạo các thể hiện của các lớp Stack với các kiểu dữ liệu & 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 CỦA TEMPLATE

- Các ràng buộc khi sử dụng các kiểu thực sự làm tham số cho lệnh template:
  - Chỉ có thể dùng kiểu số nguyên, con trỏ hoặc tham chiếu;
  - Không được gán giá trị cho tham số hoặc lấy địa chỉ của tham số.
- Loại tham số cho template là 1 template khác:
  - Ví dụ: thiết kế template cho lớp Map (ánh xạ) ánh xạ các khóa tới các giá trị:
    - Lớp này cần lưu các ánh xạ từ khóa tới giá trị, nhưng ta không muốn chỉ ra kiểu của các đối tượng được lưu trữ ngay từ đầu;
    - Cần tạo Map là 1 class template sao cho có thể sử dụng các kiểu khác nhau cho khóa & giá trị;
    - Tuy nhiên, cần chỉ ra lớp chứa (container) là 1 class template, để nó có thể lưu trữ các khóa & giá trị là các kiểu tùy ý.

CÁC THAM SỐ CỦA TEMPLATE

- Khai báo lớp Map:

```
template <typename K, typename V,
          template <typename T> Container>
class Map
{
    private:
        Container<K> keys;
        Container<V> value;
};
```

- Tạo thể hiện của lớp Map:

Map<string, int, Stack> wordcount;

  - Lệnh trên tạo 1 thể hiện của lớp Map<string, int, Stack> chứa các thành viên là 1 tập các string & 1 tập các int (giả sử còn có các đoạn mã thực hiện ánh xạ mỗi từ tới 1 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 CỦA TEMPLATE

- Như vậy, khi trình biên dịch sinh ra các khai báo & 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 template Stack để sinh mã cho 2 lớp **Stack<string>** & **Stack<int>**;
- Đến đây, ta phải hiểu rõ tại sao container phải là 1 template, nếu không, làm thế nào để có thể dùng nó để tạo các loại stack khác nhau.

Thank You !

IT FACULTY