CHƯƠNG 7.
TEMPLATE, EXCEPTION

Template

1. Lập trình tổng quát
2. Lập trình tổng quát trong C++
3. C++ template
4. Khuôn mẫu hàm
5. Khuôn mẫu lớp

Giới thiệu

- Ví dụ xét hàm hoán vị như sau:

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

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

Giới thiệu

- 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;
    //...
};
```

Giới thiệu

- 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

- 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.
- Như vậy trong một số trường hợp, đưa chi tiết về kiểu dữ liệu vào trong định nghĩa hàm hoặc lớp là điều không có lợi.

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;
}
```

Lập trình tổng quát trong C

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

- Sử dụng trình tiền xử lý của C
  - Nhàm chán và dễ lỗi
  - Chỉ cho phép đúng một định nghĩa trong một chương trình.

C++ Template

- Template (khuôn mẫu) là một cơ chế thay thế cho phép tạo các cấu trúc mà không phải chỉ rõ kiểu dữ liệu ngay từ đầu.

- Từ khóa template được dùng trong C++ để báo cho trình biên dịch biết rằng đoạn mã theo sau sẽ thao tác một hoặc nhiều kiểu dữ liệu chưa xác định

C++ Template

- Từ khóa template được theo sau bởi một cặp ngoặc nhọn chứa tên của các kiểu dữ liệu tùy ý được cung cấp.

```
template <typename T>
template <typename T, typename U>
```

- Một lệnh template chỉ có hiệu quả đối với khai báo ngay sau nó

C++ Template

- 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.

Khuôn mẫu hàm

- 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.
- Ví dụ sau định nghĩa hàm swap() bằng khuôn mẫu:

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

Khuôn mẫu hàm

- Thực chất, khi sử dụng template, ta đã định nghĩa một tập "vô hạn" các hàm chồng nhau với tên swap()
- 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); //Gọi hàm swap() với kiểu int
swap(a, b); //Gọi hàm swap() với kiểu float
```

Khuôn mẫu hàm

- 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 tiền xử lý.
  - 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

- 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? →không tìm thấy nhưng tìm thấy một template có thể dùng được.

Khuôn mẫu hàm

- Tiếp theo, nó xem xét khai báo của template swap() để xem có thể khớp được với lời gọi hàm hay không?
  - Lời gọi hàm cung cấp hai tham số thuộc cùng một kiểu (int)
  - Trình biên dịch thấy template chỉ ra hai tham số thuộc cùng kiểu T, nên nó kết luận rằng T phải là kiểu int
  - Do đó, trình biên dịch kết luận rằng template khớp với lời gọi hàm

Khuôn mẫu hàm

- 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 ra
  - Nếu không, trình biên dịch sẽ sinh một cài đặt của swap() lấy hai tham số kiểu int - và liên kết lời gọi hàm với phiên bản vừa sinh.

Khuôn mẫu hàm

- Như vậy, đến cuối quy trình biên dịch đoạn mã trong ví dụ, sẽ có hai phiên bản của swap() được tạo với các lời gọi hàm của ta được liên kết với phiên bản thích hợp.
  - Chi phí về thời gian biên dịch đối với việc sử dụng template?
  - Chi phí về không gian liên quan đến mỗi cài đặt của swap() được tạo trong khi biên dịch?

Khuôn mẫu lớp

- 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.
- Việc khai báo một khuôn mẫu lớp cũng tương tự với khuôn mẫu hàm

```
template <class T> class ClassName {
        definition
}
```

Khuôn mẫu lớp

- Ví dụ: ta sẽ tạo một cấu trúc cặp đôi giữ một cặp giá trị thuộc kiểu tùy ý.
- Trước hết, xét khai báo Pair cho một cặp giá trị kiểu int như sau:

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

Khuôn mẫu lớp

- Ta có thể sửa khai báo trên thành một khuôn mẫu lấy kiểu tùy ý:

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

- Tuy nhiên hai thành viên first và second phải thuộc cùng kiểu

Khuôn mẫu lớp

- Ta có thể cho phép hai thành viên nhận các kiểu dữ liệu khác nhau:

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

Khuôn mẫu lớp

- Để tạo các thể hiện của template Pair, ta phải dùng ký hiệu cặp ngoặc nhọn (khác với khuôn mẫu hàm)

```
Pair p;                 // Không được
Pair<int, int> q;       // Creates a pair of ints
Pair<int, float> r;     // Creates a pair with an int and
    a float
```

Khuôn mẫu lớp

- 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.
- Điều đó cho phép phát hiện các vấn đề về khái niệm trước khi chuyển thành phiên bản cho sử dụng tổng quát.

Khuôn mẫu lớp – Ví dụ

❖ Ví dụ: Xét lớp Stack với số nguyên

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

Khuôn mẫu lớp – Ví dụ

```
Stack::Stack() { this->current = 0; }
Stack::~Stack() {}
void Stack::push(const int& i) {
    if (this->current < this->max)  this->contents[this->current++] = i;
}
void Stack::pop(int& i) {
    if (this->current > 0) i = this->contents[--this->current];
}
bool Stack::isEmpty() const { return (this->current == 0;) }
bool Stack::isFull() const {
    return (this->current == this->max);
}
```

Khuôn mẫu lớp – Ví dụ

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

Khuôn mẫu lớp – Ví dụ

```
template <class T>
Stack<T>::Stack() {
    this->current = 0;
}
template <class T>
Stack<T>::~Stack() { }
template <class T>
void Stack<T>::push(const T& i)
{
    if (this->current < this->max)
        this->contents[this->current++] = i;
}
```

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 phải định nghĩa lớp Stack

Khuôn mẫu lớp – Ví dụ

```
template <class T>
void Stack<T>::pop(T& i) {
    if (this->current > 0)
        i = this->contents[--this->current];
}
template <class T>
bool Stack<T>::isEmpty() const {
    return (this->current == 0;)
}
template <class T>
bool Stack<T>::isFull() const {
    return (this->current == this->max);
}
```

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

Khuôn mẫu lớp – Ví dụ

```
int x = 5,

char c = 'a',

Stack<int>

Stack<char>

s.push(x);

t.push(c);

s.pop(y);

t.pop(d);
```

Các tham số khuôn mẫu khác

- Phần trước chúng ta chỉ mới nói đến các lệnh template với tham số thuộc "kiểu" class.
- Tuy nhiên, chúng ta có thể sử dụng các tham số kiểu và tham số biểu thức trong khuôn mẫu lớp

```
template <class T, int elements>
Stack <double, 100> s;
```

Các tham số khuôn mẫu khác

- Trong cài đặt Stack, ta có một hằng max quy định số lượng tối đa các đối tượng mà ngăn xếp có thể chứa →mỗi thể hiện sẽ có cùng kích thước đối với mọi kiểu của đối tượng được chứa.
- Ta không muốn mọi Stack đều có kích thước tối đa như nhau →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 M>
```

Các tham số khuôn mẫu khác

```
template <typename T, int M>
class Stack {
        public:
                Stack();
                ~Stack();
                void push(const T& i);
                void pop(T& i);
                bool isEmpty() const;
                bool isFull() const;
        private:
                static const int max = M;
                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;
}
```

Sửa các lệnh template

Sửa tên lớp dùng cho các toán tử phạm vi

Các tham số khuôn mẫu khác

- 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;

Stack<int, 10> t;

Stack<char, 5> u;
```

Ngoại lệ (Exception)

1. Giới thiệu
2. Cách xử lý lỗi truyền thống
3. Ngoại lệ trong C++
4. Kiểm soát ngoại lệ
5. Lớp ngoại lệ exception

Giới thiệu

- Mọi đoạn chương trình đều tiềm ẩn khả năng sinh lỗi
  - Lỗi chủ quan: do lập trình sai
  - Lỗi khách quan: do dữ liệu, do trạng thái của hệ thống
- Lỗi có 2 loại?
- Ngoại lệ (Exception): các trường hợp hoạt động không bình thường

Cách xử lý lỗi truyền thống

- Cài đặt mã xử lý tại nơi phát sinh ra lỗi
  - Làm cho chương trình trở nên khó hiểu
  - Không phải lúc nào cũng đầy đủ thông tin để xử lý
  - Không nhất thiết phải xử lý
- Truyền trạng thái lên mức trên
  - Thông qua tham số, giá trị trả lại hoặc biến tổng thể (flag)
  - Dễ nhầm
  - Khó hiểu

Cách xử lý lỗi truyền thống

```
int devide(int num, int denom, int& error){
   if (0 != denom){
        error = 0;
        return num/denom;
   } else {
        error = 1;
        return 0;
   }
}
```

Cách xử lý lỗi truyền thống

- Khó kiểm soát được hết các trường hợp
  - Lỗi số học
  - Lỗi bộ nhớ
  - …
- Lập trình viên thường quên không xử lý lỗi
  - Bản chất con người
  - Thiếu kinh nghiệm, cố tình bỏ qua

C++ Exception

- Exception – Ngoại lệ là cơ chế thông báo và xử lý lỗi giải quyết được các vấn đề gặp phải ở trên.
- Tách được phần xử lý lỗi ra khỏi phần thuật toán chính.
- Cho phép một hàm có thể thông báo về nhiều loại ngoại lệ
- Cơ chế ngoại lệ mềm dẻo hơn kiểu xử lý lỗi truyền thống

Các kiểu ngoại lệ

- Một ngoại lệ là một đối tượng chứa thông tin về một lỗi và được dùng để truyền thông tin đó tới cấp thực thi cao hơn.
- Ngoại lệ có thể thuộc kiểu dữ liệu bất kỳ của C++
  - Có sẵn, chẳng hạn int, char*, …
  - Hoặc kiểu người dùng tự định nghĩa (thường dùng)
  - Các lớp ngoại lệ trong thư viện <exception>

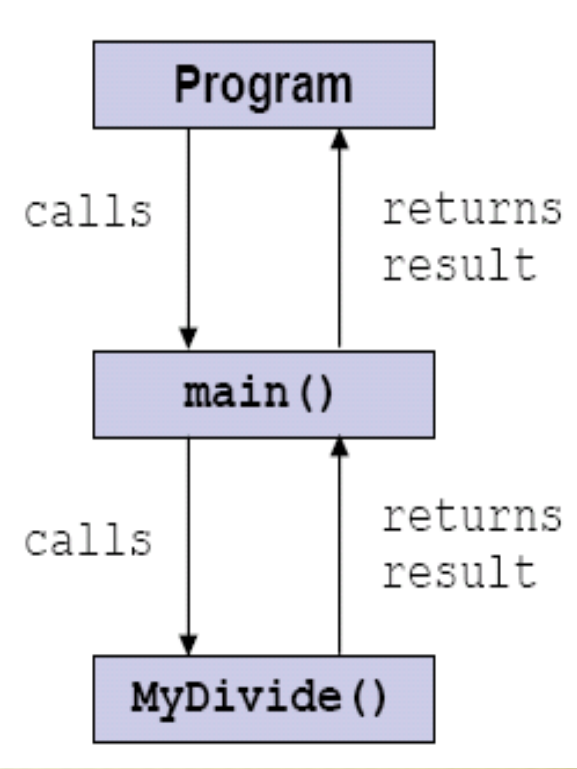
Cơ chế ngoại lệ

- Quá trình truyền ngoại lệ từ ngữ cảnh thực thi hiện hành tới mức thực thi cao hơn gọi là ném một ngoại lệ (throw an exception).
  - Vị trí trong mã của hàm nơi ngoại lệ được ném được gọi là điểm ném (throw point)
- Khi một ngữ cảnh thực thi tiếp nhận và truy nhập một ngoại lệ, nó được coi là bắt ngoại lệ (catch the exception)

Cơ chế ngoại lệ

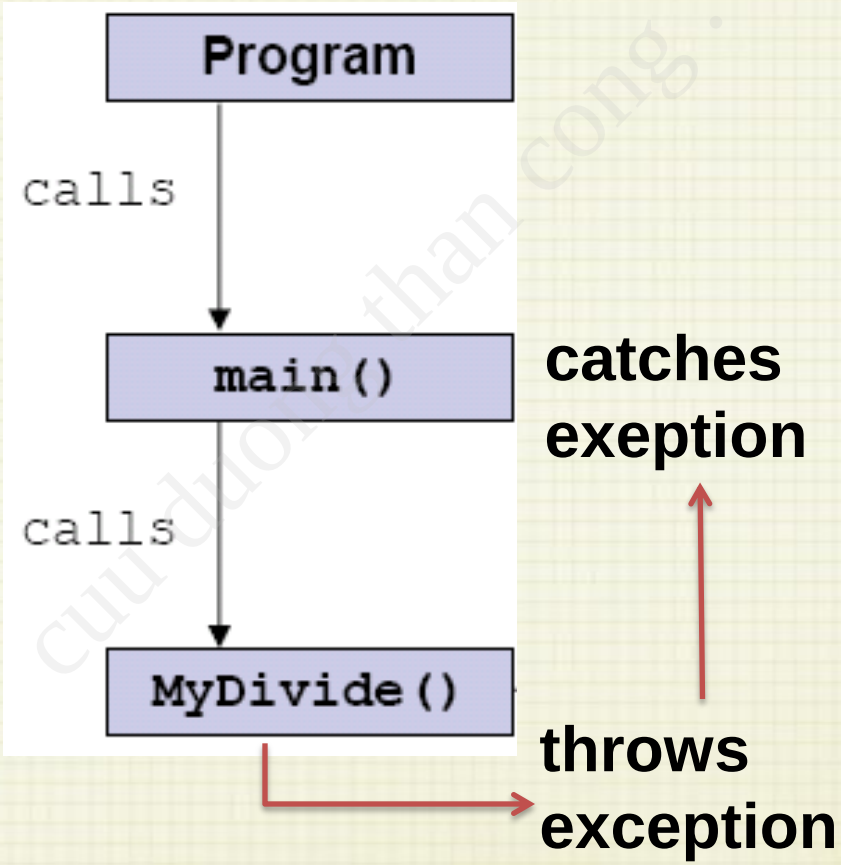
- Quy trình gọi hàm và trả về trong trường hợp bình thường:

```
void main() {
int x, y;
cout << “Nhập 2 số: ”;
cin >> x >> y;
cout << “Kết quả x/y=”;
cout<< MyDivide(x, y) << “\n”;
}
```



Cơ chế ngoại lệ

- Quy trình ném và bắt ngoại lệ:



Cú pháp xử lý ngoại lệ

- Cơ chế xử lý ngoại lệ của C++ có 3 tính năng chính:
  - Khả năng tạo và ném ngoại lệ (sử dụng từ khoá throw)
  - Khả năng bắt và giải quyết ngoại lệ (sử dụng từ khoá catch)
  - Khả năng tách logic xử lý ngoại lệ trong một hàm ra khỏi phần còn lại của hàm (sử dụng từ khoá try)

Ném ngoại lệ – throw

- Để ném một ngoại lệ, ta dùng từ khóa throw, kèm theo đối tượng mà ta định ném.
- Ta có thể dùng mọi thứ làm ngoại lệ, kể cả giá trị thuộc kiểu có sẵn.

```
double MyDivide(double numerator, double denominator){
    if (denominator == 0.0) {
        throw string("The denominator cannot be 0.");
    } else {
        return numerator / denominator;
    }
}
```

Kiểm soát ngoại lệ

- Khối try – catch dùng để:
  - Tách phần giải quyết lỗi ra khỏi phần có thể sinh lỗi
  - Quy định các loại ngoại lệ được bắt tại mức thực thi hiện hành

```
try {
    // Code that could generate an exception
}
catch (<Type of exception>) {
    // Code that resolves an exception of that type
};
```

Kiểm soát ngoại lệ

- Có thể có nhiều khối catch, mỗi khối chứa mã để giải quyết một loại ngoại lệ cụ thể:

```
try {
    // Code that could generate an exception
}
catch (<Exception type1>) {
    // Code that resolves a type1 exception
}
catch (<Exception type2>) {
    // Code that resolves a type2 exception
}
catch (<Exception typeN>) {
    // Code that resolves a typeN exception
};
```

Kiểm soát ngoại lệ – Ví dụ

```
void main() {
        int x, y;
        double result;
        cout << “Nhập 2 số: ”;
        cin >> x >> y;
        try {
                result = MyDivide(x, y);
                cout << “Kết quả x/y = ”<< result << “\n”;
        }
        catch (string &s) {
                 cout<<s<<endl;  //resolve error
        };
}
```

Kiểm soát ngoại lệ – Ví dụ

```
void main(){
        int x, y;
        double result;
        bool success;
        do {
                success = true;
                cout << "Nhập 2 số: ";
                cin >> x >> y;
                try {
                        result = Divide(x, y);
                        cout << "Kết quả x/y = "<< result << "\n";
                }
                catch (string& s) {
                        cout << s << endl;
                        success = false;
                };
        } while (success == false);
}
```

So khớp ngoại lệ

- Khi một ngoại lệ được ném từ trong một khối try, hệ thống xử lý ngoại lệ sẽ kiểm tra các kiểu được liệt kê trong khối catch theo thứ tự liệt kê:
  - Khi tìm thấy kiểu đã khớp, ngoại lệ được coi là được giải quyết, không cần tiếp tục tìm kiếm.
  - Nếu không tìm thấy, mức thực thi hiện hành bị kết thúc, ngoại lệ được chuyển lên mức cao hơn.

So khớp ngoại lệ

- Khi tìm các kiểu dữ liệu khớp với ngoại lệ, trình biên dịch nói chung sẽ không thực hiện đổi kiểu tự động.
  - Nếu một ngoại lệ kiểu float được ném, nó sẽ không khớp với một khối catch cho ngoại lệ kiểu int
- Một đối tượng hoặc tham chiếu kiểu dẫn xuất sẽ khớp với một lệnh catch dành cho kiểu cơ sở
  - Nếu một ngoại lệ kiểu Car được ném, nó sẽ khớp với một khối catch cho ngoại lệ kiểu MotorVehicle

So khớp ngoại lệ

```
try {
    //…
}
catch (MotorVehicle& mv) {…}
catch (Car& c) {…}
catch (Truck& t) {…};
```

- Vấn đề gặp phải?
- Mọi ngoại lệ là đối tượng được sinh từ cây MotorVehicle sẽ khớp lệnh catch đầu tiên (các lệnh còn lại sẽ không bao giờ chạy)

So khớp ngoại lệ

- Nếu muốn bắt các ngoại lệ dẫn xuất tách khỏi ngoại lệ cơ sở, ta phải xếp lệnh catch cho lớp dẫn xuất lên trước:

```
try {
    //…
}
catch (Car& c) {…}
catch (Truck& t) {…}
catch (MotorVehicle& mv) {…};
```

So khớp ngoại lệ

- Nếu ta muốn bắt tất cả các ngoại lệ được ném (kể cả các ngoại lệ ta không thể giải quyết)?
- Để có một lệnh catch bắt được mọi ngoại lệ, ta đặt dấu ba chấm bên trong lệnh catch.

```
catch(…){
    //…
};
```

- Chỉ nên sử dụng nó cho lệnh catch cuối cùng trong một khối try-catch.

Lớp exception

- Để tích hợp hơn nữa các ngoại lệ vào ngôn ngữ C++, lớp exception đã được đưa vào thư viện chuẩn.
  - Sử dụng #include <exception> và namespace std
- Sử dụng thư viện này, ta có thể ném các thể hiện của exception hoặc tạo các lớp dẫn xuất từ đó.
- Lớp exception có một hàm ảo what(), có thể định nghĩa lại what() để trả về một xâu ký tự.

Lớp exception

- Một số lớp ngoại lệ chuẩn khác được dẫn xuất từ lớp cơ sở exception.
- File header <stdexcept> (cũng thuộc thư viện chuẩn C++) chứa một số lớp ngoại lệ dẫn xuất từ exception. Trong đó có hai lớp quan trọng được dẫn xuất trực tiếp từ exception:
  - runtime_error
  - logic_error

Lớp exception

- runtime_error: Các lỗi trong thời gian chạy (các lỗi là kết quả của các tình huống không mong đợi, chẳng hạn: hết bộ nhớ)
- logic_error: Các lỗi trong logic chương trình (chẳng hạn truyền tham số không hợp lệ)
- Thông thường, ta sẽ dùng các lớp này (hoặc các lớp dẫn xuất của chúng) thay vì dùng trực tiếp exception

Lớp exception

- runtime_error có các lớp dẫn xuất sau:
  - range_error điều kiện sau (post-condition) bị vi phạm
  - overflow_error xảy ra tràn số học
  - bad_alloc không thể cấp phát bộ nhớ
- logic_error có các lớp dẫn xuất sau:
  - domain_error điều kiện trước (pre-condition) bị vi phạm
  - invalid_argument tham số không hợp lệ được truyền cho hàm
  - length_error tạo đối tượng lớn hơn độ dài cho phép
  - out_of_range tham số ngoài khoảng

Lớp exception

- Ta có thể viết lại hàm MyDivide() để sử dụng các ngoại lệ chuẩn tương ứng như sau:

```
double MyDivide(double numerator, double denominator)
{
    if (denominator == 0.0) {
        throw invalid_argument("The denominator cannot
    be 0.");
    } else {
        return numerator / denominator;
    }
}
```

Lớp exception

```
void main() {
        int x, y;
        double result;
        do {
                cout << “Nhập 2 số:  ”; cin >> x >> y;
                try {
                        result = MyDivide(x, y);
                        cout << “x/y = ” << result << “\n”;
                }
                catch (invalid_argument& e) {
                        cout << e.what() << endl;
                };
        } while (1);
}
```

Ưu điểm exception trong C++

- Dễ sử dụng
  - Dễ dàng chuyển điều khiển đến nơi có khả năng xử lý ngoại lệ
  - Có thể "ném" nhiều loại ngoại lệ
- Tách xử lý ngoại lệ khỏi thuật toán
  - Tách mã xử lý
  - Sử dụng cú pháp khác
- Không bỏ sót ngoại lệ ("ném" tự động)
- Chương trình dễ đọc hơn, an toàn hơn

Q & A