Mục lục

A. KHUÔN MẪU (TEMPLATE)

7.1 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?

- Có thực sự cần đến nhiều phiên bản không?
- Như vậy ta phải *viết rất nhiều định nghĩa hàm hoàn toàn tương tự nhau*, chỉ có kiểu *dữ liệu là thay đổi.*

❖ Mở rộng ví dụ trên: **Bài toán:**

- Viết hàm hoán vị 2 số nguyên
- Viết hàm hoán vị 2 số thực
- Viết hàm hoán vị 2 kí tự
  - **Cách giải quyết:** Sử dụng chồng hàm swap

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

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

```
void hoanvi(char &a, char &b)
{
    char temp = a;
    a = b;
    b = temp;
}
```

- **Chồng hàm gây ra tình trạng "lặp lại mã"**
  → Sử dụng khuôn mẫu hàm (Function Template)
- Khuôn mẫu hàm cho phép *viết 1 lần và sử dụng cho nhiều kiểu dữ liệu khác nhau*
- Bộ dịch sẽ *sinh ra nhiều phiên bản khác nhau của 1 khuôn mẫu hàm khi hàm được gọi* **(hàm thể hiện)**

❖ **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;
      //...
};
```

- 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?

7.2 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.

❖ **Sử dụng trình tiền xử lý của C**
- Trình tiền xử lý thực hiện **thay thế text** *(thay thế mọi TYPE bằng int)* **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;
}
```

- ***Hạn chế:*** Nhàm chán và dễ lỗi; **Chỉ cho phép đúng một định nghĩa** trong một chương trình.

7.3 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*
- Khuôn mẫu là một trong những tính năng phức tạp nhất và mạnh nhất của ngôn ngữ C++
- Là công cụ hỗ trợ tái sử dụng code
- Là cơ chế cho phép **lập trình tổng quát**
- C++ hỗ trợ 2 loại khuôn mẫu: **khuôn mẫu hàm và khuôn mẫu lớp**
  - 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/lớp là hàm/lớp **tổng quát**, không phụ thuộc vào kiểu dữ liệu

7.4 Khuôn mẫu hàm (Function Template)

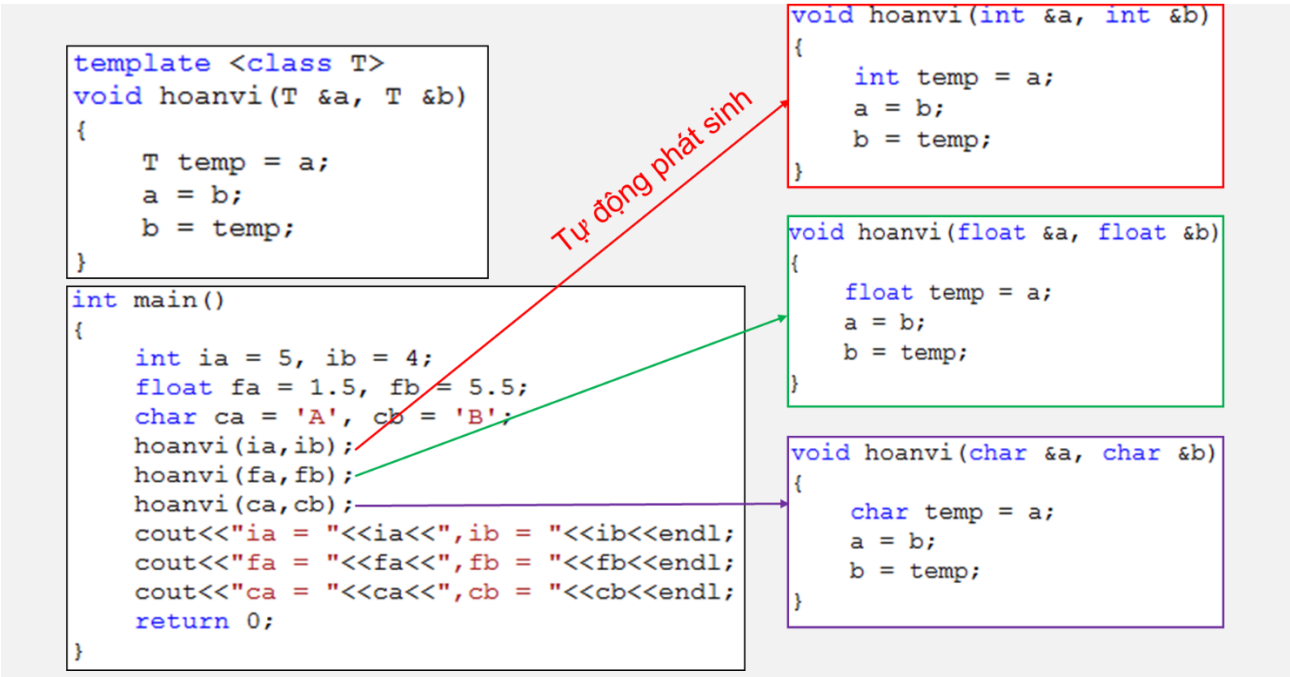
❖ Từ khóa template được theo sau bởi một cặp ngoặc nhọn chứa tên của các kiểu dữ liệu tùy ý được cung cấp.

```
//Khuôn mẫu hàm với 1 tham số kiểu dữ liệu T
template <class T>
return-type funcname (arguments of type T)
{
        //thân khuôn mẫu hàm
}
```

```
//Khuôn mẫu hàm với nhiều tham số kiểu dữ liệu T1, T2,…
template <class T1, class T2,…>
return-type funcname (arguments of type T1,T2)
{
        //thân khuôn mẫu hàm
}
```

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

❖ **Ví dụ 1:**



❖ Thực chất, khi sử dụng template, ta đã định nghĩa **một tập "vô hạn" các hàm chồng nhau** với tên **hoanvi().**

❖ Để gọi một trong các phiên bản này, ta chỉ cần **gọi nó với kiểu dữ liệu tương ứng.**

❖ Dùng function template chúng ta chỉ cần viết ***một hàm duy nhất cho nhiều kiểu dữ liệu khác nhau*** thay vì phải viết nhiều hàm cho từng kiểu dữ liệu cụ thể.

❖ Dùng function template giúp **giảm được kích thước và tăng tính linh động** của chương trình.

❖ *Chuyện gì xảy ra khi ta biên dịch mã?*

- Trước hết, ***sự thay thế "T"*** trong khai báo/định nghĩa hàm **hoanvi()** *không phải thay thế text đơn giản và cũng* ***không được thực hiện bởi trình tiền xử lý.***
- Việc chuyển phiên bản mẫu của **hoanvi()** thành các cài đặt cụ thể cho int và float được thực hiện bởi **trình biên dịch.**

❖ Hãy xem xét hoạt động của trình biên dịch khi gặp lời gọi **hoanvi() thứ nhất** (với hai tham số **int**)

- Trước hết, trình biên dịch tìm xem có một hàm **hoanvi()** được khai báo với 2 tham số kiểu int hay không? →không tìm thấy *nhưng tìm thấy một template có thể dùng được*.

- ❖ Tiếp theo, nó xem xét khai báo của template **hoanvi()** để xem có thể khớp được với lời gọi hàm hay không?
  - *Lời gọi hàm cung cấp **hai đối 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*
- ❖ Khi đã xác định được **template khớp với lời gọi hàm**, trình biên dịch kiểm tra xem đã có một phiên bản của **hoanvi()** 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 **hoanvi()** 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*.
- ❖ Như vậy, đến *cuối quy trình biên dịch* đoạn mã trong ví dụ, **sẽ có 3 phiên bản của hoanvi() được tạo** *(một cho hai tham số kiểu int, một cho hai tham số kiểu float, một cho hai tham số kiểu char)* với các lời gọi hàm của ta được liên kết với phiên bản thích hợp.
  - 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 **hoanvi()** 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*

❖ **Ví dụ 2:** *Sử dụng khuôn mẫu hàm với kiểu dữ liệu lớp*

```
template <class T>
void hoanvi(T &a, T &b)
{
    T temp = a;
    a = b;
    b = temp;
}
int main()
{
    Point p1(2,4), p2(1,5);
    hoanvi(p1,p2);
    cout<<"p1:"<<p1<<", p2:"<<p2<<endl;
    return 0;
}
```

❖ **Ví dụ 3:** *Khuôn mẫu hàm với nhiều tham số kiểu dữ liệu*

```
template<class T1, class T2>
T1 sum(T1 x, T2 y, T1 z)
{
    return x + y + z;
}
int main()
{
    int ia = 5, ib = 4, ic = 3;
    float fa = 1.5, fb = 5.0, fc = 4.0;
    cout<<sum(ia,fa,ib)<<endl;//(int,float,int)
    cout<<sum(fa,ia,fb)<<endl;//(float,int,float)
    cout<<sum(ia,ib,ic)<<endl;//(int, int, int)
    cout<<sum(fa,fb,fc)<<endl;//(float,float,float)
    //cout<<sum(ia,fa,fb)<<endl;//(int,float,float)
    return 0;
}
```

error C2782: 'T1 sum(T1,T2,T1)': template parameter 'T1' is ambiguous.

- ❖ **Hạn chế của khuôn mẫu hàm**
  - Về nguyên tắc, một tham số kiểu có thể tương ứng với kiểu dữ liệu bất kì
  - Tuy nhiên, trong một số trường hợp hàm thể hiện được sinh ra **không thực hiện đúng hoặc báo lỗi dịch**

- **Ví dụ 1:**

```
template<class T>
T find_min(T a, T b)
{
    return (a < b ? a : b);
}
int main()
{
    cout<<find_min(4,5)<<endl; //return 4
    cout<<find_min(1.5,2.0)<<endl;//return 1.5
    cout<<find_min("HELLO","ANNA")<<endl; // return HELLO. Incorrect Result
    return 0;
}
```

Giải thích lỗi:

❖ **Đầu vào kiểu const char***:

- Khi gọi find_min("HELLO", "ANNA"), các chuỗi ký tự như "HELLO" và "ANNA" thực sự được coi là con trỏ kiểu **const char*,** trỏ đến địa chỉ của các chuỗi này trong bộ nhớ.
- Do đó, việc so sánh a < b trong hàm find_min sẽ thực hiện so sánh địa chỉ bộ nhớ của hai con trỏ thay vì so sánh nội dung chuỗi. Kết quả có thể không đúng với mong đợi, vì địa chỉ bộ nhớ không đảm bảo thứ tự từ điển.

"Anna" có địa chỉ lớn hơn "Hello".
Suy ra "Anna" > "Hello"

- *Khi một chuỗi được viết dưới dạng "..." trong C/C++ (từ C++98 trở đi), nó **mặc định** là một hằng chuỗi ký tự (string literal), và kiểu dữ liệu của nó là const char*, để bảo vệ chuỗi hằng khỏi bị thay đổi.*
- ***Lý do:** Trước đây, trong **C (phiên bản cũ)**, các chuỗi hằng ("...") được coi là kiểu char*, và việc sửa đổi chuỗi hằng **không bị ngăn cản**. Tuy nhiên, điều này dẫn đến lỗi chương trình hoặc hành vi không xác định (undefined behavior) khi cố gắng thay đổi nội dung chuỗi, vì chuỗi hằng nằm trong bộ nhớ chỉ đọc.*

❖ **Hành vi không mong muốn**:

- Kết quả "HELLO" được trả về không phải do "HELLO" nhỏ hơn "ANNA" theo từ điển, mà là do **địa chỉ của "HELLO" trong bộ nhớ nhỏ hơn địa chỉ của "ANNA", theo thứ tự chúng được lưu vào bộ nhớ Stack (như hình)**. Điều này dẫn đến kết quả không chính xác.

❖ **Cách khắc phục:**

Thay đổi lời gọi hàm như sau:

```
#include <iostream>
#include <string> // Thêm thư viện để dùng std::string

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

```
int main()
{
    std::cout << find_min(4, 5) << std::endl;          // Trả về 4
    std::cout << find_min(1.5, 2.0) << std::endl;      // Trả về 1.5
    std::cout << find_min(std::string("HELLO"), std::string("ANNA"))
              << std::endl; // Trả về ANNA
    //Hoặc: std::cout << find_min("ANNA","HELLO") << std::endl;
    return 0;
}
```

**** Kiểu const char* là gì?***

- **const char*** là một con trỏ (pointer) trỏ đến một chuỗi ký tự không thay đổi (constant character array). Nó được sử dụng để **lưu trữ địa chỉ của mảng ký tự** (c-string).
- Kiểu này thường được sử dụng trong ngôn ngữ C/C++ để xử lý ***chuỗi kiểu null-terminated***, nghĩa là ***chuỗi kết thúc bằng ký tự '\0'.***

Đặc điểm:

❖ **Hằng chuỗi ký tự ("string")**:
- Một chuỗi được viết trong cặp dấu nháy kép (e.g., "HELLO") mặc định có kiểu const char*.
- Ví dụ: **"HELLO"** được lưu trong bộ nhớ như một mảng ký tự **{ 'H', 'E', 'L', 'L', 'O', '\0' }**.

❖ **Tính chất không thay đổi**:
- Dữ liệu chuỗi được trỏ đến bởi const char* không thể bị sửa đổi, vì nó là hằng (const).

❖ **So sánh**:
- Việc so sánh hai chuỗi kiểu const char* sử dụng toán tử so sánh (<, >, ==) sẽ so sánh địa chỉ bộ nhớ mà con trỏ trỏ tới, **không phải nội dung** của chuỗi.

Ví dụ a: **Khai báo và sử dụng**

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

int main() {
    const char* str1 = "Hello";
    const char* str2 = "World";

    cout << "Chuỗi 1: " << str1 << endl; // In "Hello"
    cout << "Chuỗi 2: " << str2 << endl; // In "World"

    // So sánh địa chỉ bộ nhớ
    if (str1 < str2) {
        cout << "str1 nằm trước str2 trong bộ nhớ." << endl;
    } else {
        cout << "str1 nằm sau str2 trong bộ nhớ." << endl;
    }

    return 0;
}
```

Kết quả:

- Có thể in "Hello" và "World".
- Kết quả của so sánh str1 < str2 phụ thuộc vào địa chỉ bộ nhớ, không liên quan đến thứ tự từ điển.

Ví dụ b: **Sử dụng với hàm strlen và strcmp**

Khi *cần so sánh nội dung chuỗi hoặc làm việc với độ dài*, ta có thể dùng các hàm từ thư viện <cstring>:

```
#include <iostream>
#include <cstring> // Để sử dụng strlen, strcmp

using namespace std;

int main() {
    const char* str1 = "Hello";
    const char* str2 = "World";

    // Độ dài chuỗi
    cout << "Độ dài chuỗi 1: " << strlen(str1) << endl; // 5
    cout << "Độ dài chuỗi 2: " << strlen(str2) << endl; // 5

    // So sánh nội dung chuỗi
    int comparison = strcmp(str1, str2);
    if (comparison < 0) {
        cout << "str1 nhỏ hơn str2 theo thứ tự từ điển." << endl;
    } else if (comparison > 0) {
        cout << "str1 lớn hơn str2 theo thứ tự từ điển." << endl;
    } else {
        cout << "str1 bằng str2." << endl;
    }

    return 0;
}
```

Kết quả:

- strlen trả về độ dài chuỗi (không tính ký tự '\0').
- strcmp:
  - Trả về một số **âm** nếu str1 nhỏ hơn str2 theo thứ tự từ điển.
  - Trả về **0** nếu str1 bằng str2.
  - Trả về một số **dương** nếu str1 lớn hơn str2.

Lưu ý khi sử dụng const char*:

❖ **Không được sửa đổi nội dung chuỗi**:
  - Chuỗi hằng "Hello" được *lưu trong vùng bộ nhớ chỉ đọc*. Nếu cố gắng thay đổi nội dung, chương trình sẽ báo lỗi hoặc dẫn đến hành vi không xác định:
  - const char* str = "Hello";
  - str[0] = 'h'; // Lỗi! Không thể sửa đổi.

❖ **Không so sánh bằng ==, <, > nếu cần kiểm tra <u>nội dung</u> chuỗi**:
  - Chỉ nên sử dụng *strcmp hoặc chuyển sang kiểu std::string* để đảm bảo so sánh nội dung chính xác.

❖ **Thay thế bằng std::string khi cần**:
  - std::string tiện lợi hơn vì hỗ *trợ các toán tử so sánh (<, >, ==)* ***dựa trên nội dung*** chuỗi, đồng thời *cho phép chỉnh sửa nội dung.*

- **Ví dụ 2:**

```
template<class T>
T find_min(T a, T b)
{
    return (a < b ? a : b);
}
int main()
{
    Circle c1(2,3,5), c2(2,3,1);
    cout<<find_min(c1,c2)<<endl;

    return 0;
}
```

error C2676: binary '<' :
'Circle' does not define this operator

7.5 Khuôn mẫu lớp (Class Template)

- 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.
- Khuôn mẫu lớp cho phép tạo ra lớp tổng quát
- Những lớp chỉ làm việc trên **một kiểu dữ liệu** thì không nên tổng quát hóa
- Những lớp chứa (*container class*) như **Stack**, **List**,… nên được tổng quát hóa.
- **Bộ dịch** sẽ sinh ra nhiều phiên bản khác nhau của lớp khi đối tượng được tạo

- ***Khai báo:***

```
//Khuôn mẫu lớp với 1 tham số kiểu dữ liệu T
template <class T>
class  class_name
{
      //thành viên của lớp với kiểu dữ liệu T
}
```

```
//Khuôn mẫu lớp với nhiều tham số kiểu dữ liệu T1, T2,…
template <class T1, class T2,…>
class  class_name
{
   //thành viên của lớp với kiểu dữ liệu T1, T2
}
```

- ***Tạo đối tượng của lớp mẫu***

```
      class_name <T> obj_name;
      class_name <T1,T2> obj_name;
```

- ***Định nghĩa hàm thành viên của lớp mẫu***

```
template <class T>
return-type class-name <T>:: function-name (argument list)
{
   //thân hàm thành viên của lớp mẫu với kiểu dữ liệu T
}
```

Ví dụ:

My_List.h

```
#pragma once
template<class T>
class My_List
{
private:
//so luong phan tu lon nhat danh sach co the chua
    int max_size;
//count:so luong phan tu hien co trong danh sach
    int count;
    T *data;
public:
     My_List(int size);
    ~My_List(void);
    bool IsEmpty() const;
    bool IsFull() const;
    int  getCount() const;
    void Insert(T newElem);
    void Delete(T elem);
    bool IsContain(T elem) const;
    void Print() const;

};
```

My_List.cpp

```
#include "My_List.h"
#include <iostream>
using namespace std;

template<class T>
My_List<T>::My_List(int size)
{
    count = 0;
    max_size = size;
    data = new T[max_size];
}
template<class T>
void My_List<T>::Print() const
{
    for(int i = 0; i < count;i++)
        cout<<data[i]<<"\t";
    cout<<endl;
}
template<class T>
void My_List<T>::Insert(T newElem)
{
    data[count++] = newElem;
}
```

main.cpp

```
#include "My_List.h"
#include "My_List.cpp"
using namespace std;
int main()
{
    My_List<int> ds1(5);
    My_List<float> ds2(5);
    My_List<char> ds3(5);
    cout<<"Danh sach 1 co: "<<ds1.getCount()<<" phan tu"<<endl;
    ds1.Insert(4);
    ds1.Insert(1);
    cout<<"Danh sach 1 co: "<<ds1.getCount()<<" phan tu"<<endl;
    ds2.Insert(4.5);
    ds2.Insert(1.5);
    ds3.Insert('A');
    ds1.Print();
    ds2.Print();
    ds3.Print();
    return 0;
}
```

Thêm dòng **#include "My_List.cpp"** để giải quyết vấn đề lỗi liên kết file .h và .cpp

Nếu không sẽ báo lỗi:

```
error LNK2019: unresolved external
symbol "public:
 __thiscall My_List<int>::~My_List<int>(void)
……
```

7.6 Khai báo bạn bè của khuôn mẫu lớp

(Sử dụng nạp chồng toán tử)

```
#include <iostream>
using namespace std;
template<class T>
class My_List
{
private:
//so luong phan tu lon nhat danh sach co the chua
    int max_size;
//count:so luong phan tu hien co trong danh sach
    int count;
    T *data;
public:
     My_List(int size);
    ~My_List(void);
    bool IsEmpty() const;
    bool IsFull() const;
    int  getCount() const;
    void Insert(T newElem);
    void Delete(T elem);
    bool IsContain(T elem) const;
   // void Print() const;
    template<class T>
    friend ostream& operator<<(ostream& out, const My_List<T> &ds);
};
```

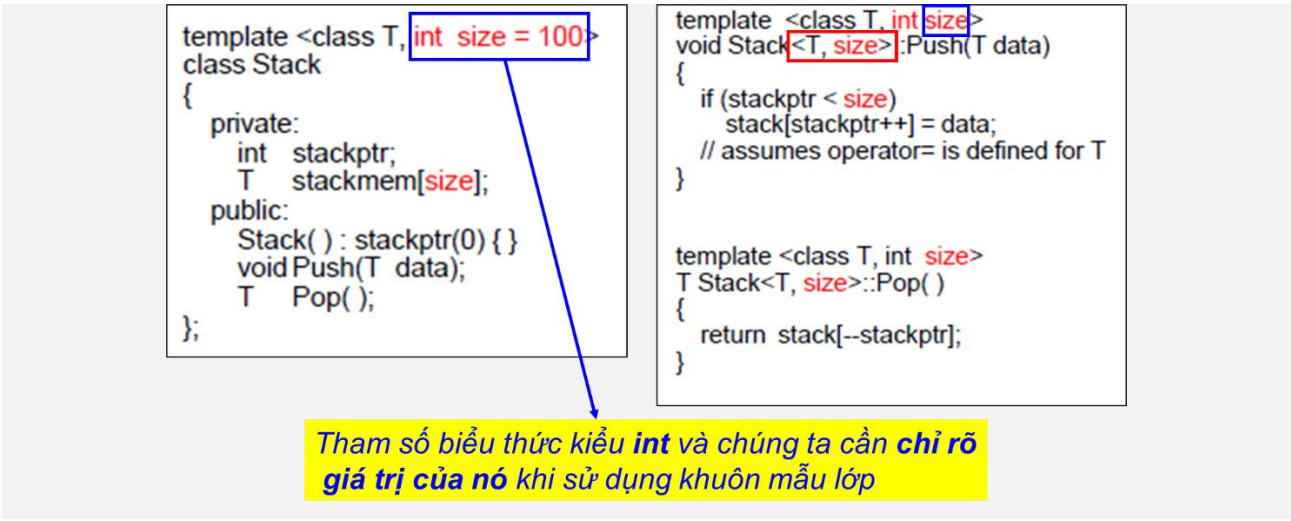
Sử dụng toán tử << để in danh sách

```
#include "My_List.cpp"
#include <iostream>
#include "My_List.h"
using namespace std;
int main()
{
    My_List<int> ds1(5);
    My_List<float> ds2(5);
    My_List<char> ds3(5);
    cout<<"Danh sach 1 co: "<<ds1.getCount()<<" phan tu"<<endl;
    ds1.Insert(4);
    ds1.Insert(1);
    cout<<"Danh sach 1 co: "<<ds1.getCount()<<" phan tu"<<endl;
    ds2.Insert(4.5);
    ds2.Insert(1.5);
    ds3.Insert('A');
    cout<<ds1<<endl;
    cout<<ds2<<endl;
    cout<<ds3<<endl;
    return 0;
}
```

7.7 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

❖ **Ví dụ:**



`void main( )`
`{`
`Stack<int, 10> S;`
`Stack<double> D;`
 	
`S.Push(10);`
`S.Push(20);`
 	
`D.Push(2.7);`
`D.Push(3.141495);`
 	
`cout << S.Pop( ) << endl;`
`cout << D.Pop( ) << endl;`
`}`
	Khai báo **template <class T, int size = 100>** là **tổng quát**, tức là cho phép sử dụng bất kỳ kiểu dữ liệu nào cho T ***miễn là kiểu đó hỗ trợ toán tử gán (=).***
	
	Stack<double>: kiểu T là double, và tham số size trong khuôn mẫu mặc định **100** sẽ được sử dụng, & giá trị này sẽ được sử dụng trong phiên bản cụ thể của lớp Stack được tạo ra cho đối tượng ***D.***
	
	Stack<int, 10>, tham số size trong khuôn mẫu sẽ được gán giá trị **10,** và giá trị này sẽ được sử dụng trong phiên bản cụ thể của lớp Stack được tạo ra cho đối tượng ***S.***

7.7.1 Phép gán giữa các đối tượng của các lớp thể hiện

`Stack<int, 10> S;`
`Stack<double> D;`
`D = S; // error`
	• Stack<int, 10> và Stack<double> là **hai lớp hoàn toàn khác nhau**.
	• Mặc dù cả hai đều sử dụng cùng một khuôn mẫu Stack, nhưng khi cụ thể hóa (instantiation), chúng tạo ra các lớp **riêng biệt** với *kiểu T và kích thước size khác nhau.*
	• Do đó, phép gán D = S; là không hợp lệ vì kiểu của chúng không tương thích.
	• Phép gán giữa hai đối tượng chỉ hợp lệ nếu chúng thuộc **cùng một kiểu** (cùng loại template với các tham số mẫu giống nhau).

`Stack<int, 10> S;`
`Stack<int, 20> D;`
`S = D;  //error`
	• Mặc dù cả hai đều sử dụng cùng kiểu dữ liệu int, nhưng các giá trị của tham số mẫu size (10 và 20) khác nhau.
	• Điều này dẫn đến việc **Stack<int, 10> và Stack<int, 20>** ***vẫn được xem là hai lớp khác nhau.***
	• Trình biên dịch coi Stack<int, 10> và Stack<int, 20> là hai kiểu không tương thích.
	• Phép gán giữa các đối tượng Stack<int, size> chỉ hợp lệ nếu **giá trị của tham số size giống nhau.**

7.8 Standard Template Library (STL)

- STL là thư viện của C++ cung cấp nhiều cấu trúc dữ liệu, và thuật toán để hỗ trợ cho lập trình máy tính
- Mỗi lớp trong STL đều là **khuôn mẫu**
- STL gồm nhiều lớp chứa như: **vector, list, set, map, stack, queue,…**
- Mỗi lớp chứa trong STL có thể sử dụng với kiểu dữ liệu, đối tượng bất kì

Ví dụ 1: Lớp Vector

```
// Instantiate a vector
vector<int> V;
// Insert elements
V.push_back(2);          // v[0] == 2
V.insert(V.begin(), 3);  // V[0] == 3, V[1] == 2
// Random access
V[0] = 5;                // V[0] == 5
// Test the size
int size = V.size();     // size == 2
```

Ví dụ 2: Lớp map

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

int main() {
    map<string, int> freq; // map of words and their frequencies
    string word;           // input buffer for words.

    //--- Read words/tokens from input stream
    while (cin >> word) {
        freq[word]++;
    }

    //--- Write the count and the word.
    map<string, int>::const_iterator iter;
    for (iter=freq.begin(); iter != freq.end(); ++iter) {
        cout << iter->second << " " << iter->first << endl;
    }
    return 0;
}//end main
```

*Chương trình **đếm tần suất xuất hiện của các từ trong đầu** vào (input stream) bằng cách sử dụng **std::map***

❖ **map<string, int> freq;**:
- Tạo một đối tượng std::map tên là freq.
- **Key (string)**: Là từ (word) được nhập vào.
- **Value (int)**: Là số lần (tần suất) từ đó xuất hiện.

❖ **string word;**:
- Tạo một biến word để lưu trữ tạm thời từ được nhập vào.

❖ **while (cin >> word)**:
- Vòng lặp đọc các từ từ **dòng đầu vào chuẩn (std::cin)**.
- **cin >> word**: Đọc từng từ (cách nhau bởi khoảng trắng) vào biến word.
- Khi một từ được đọc thành công:
  - **freq[word]++**:
    - Nếu word đã có trong map freq, giá trị của freq[word] sẽ tăng lên 1.
    - Nếu word chưa có trong map, nó sẽ được *tự động thêm vào với giá trị khởi tạo là 0,* sau đó tăng lên 1 (tương đương với freq[word] = 1).

❖ **map<string, int>::const_iterator iter;**:
- Khai báo một iterator (iter) để duyệt qua các phần tử trong map.
- **const_iterator**: Được sử dụng vì dữ liệu trong map không bị thay đổi khi duyệt qua *(const_iterators don't allow you to change the values that they point to, regular iterators do).*

❖ **for (iter = freq.begin(); iter != freq.end(); ++iter)**:
- Vòng lặp duyệt qua từng cặp **key-value** trong map:
  - **freq.begin()**: Trỏ đến phần tử đầu tiên trong map.
  - **freq.end()**: Trỏ đến **sau phần tử cuối cùng** trong map (không truy cập được).
  - **++iter**: Di chuyển iterator tới phần tử tiếp theo.

❖ **cout << iter->second << " " << iter->first << endl;**:
- **iter->second**: Lấy giá trị (tần suất) của từ.
- **iter->first**: Lấy key (từ).
- In ra tần suất (second) trước, sau đó là từ (first), mỗi từ trên một dòng.

Cách hoạt động của chương trình:

Input:
- Chương trình nhận input từ người dùng thông qua **dòng đầu vào chuẩn (std::cin)** hoặc từ file (nếu input được chuyển hướng từ file).

Output:
- In ra danh sách các từ trong input kèm theo số lần xuất hiện của mỗi từ.

Quá trình:
- Người dùng nhập vào một chuỗi các từ.
- Mỗi từ sẽ được thêm vào map freq hoặc tăng giá trị đếm nếu từ đó đã có.
- Sau khi đọc xong, chương trình duyệt qua map và in ra các từ cùng tần suất của chúng.

Ví dụ chạy chương trình:
- **Input:** `hello world hello map example example hello`
- **Output:**

```
1 map
1 world
2 example
3 hello
```

Giải thích:
- Các từ xuất hiện trong map freq theo thứ tự **sắp xếp từ điển**:
  - "example" xuất hiện 2 lần.
  - "hello" xuất hiện 3 lần.
  - "map" xuất hiện 1 lần.
  - "world" xuất hiện 1 lần.

Lưu ý:

❖ **Map tự động sắp xếp key:**
- std::map tự động sắp xếp các phần tử theo thứ tự tăng dần của key (dựa trên so sánh từ điển với kiểu std::string).

❖ **Phân biệt chữ hoa - chữ thường:**
- Từ "Hello" và "hello" sẽ được coi là **khác nhau**, vì std::string phân biệt chữ hoa và chữ thường.

❖ **Nhập liệu kết thúc:**
- Người dùng phải nhấn **Ctrl+D** (trên Linux/Mac) hoặc **Ctrl+Z** (trên Windows) để kết thúc nhập liệu và cho vòng lặp while thoát.

B. NGOẠI LỆ (EXCEPTION)

7.9 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?**
  - ***Lỗi lúc biên dịch*** (compile-time error-lỗi cú pháp), ***lỗi lúc thực thi*** (run-time error- giải thuật sai, không dự đoán được tình huống).
  - Thí dụ: thực hiện phép chia mà mẫu số là 0
  - Khi 1 **run-time error** xảy ra, chương trình *kết thúc đột ngột và điều khiển được trả lại cho OS* ➔ Cần phải quản lý được các tình huống này.
- **Ngoại lệ (Exception):**
  - Exception là một lỗi đặc biệt. Lỗi này xuất hiện vào lúc thực thi chương trình.
  - Các *trạng thái không bình thường* xảy ra trong khi thi hành chương trình tạo ra các exception. Những trạng thái này *không được biết trước* trong khi ta đang xây dựng chương trình.
  - Nếu bạn không phân phối các trạng thái này thì exception có thể bị kết thúc đột ngột.
  - Ví dụ, việc chia cho 0 sẽ tạo một lỗi trong chương trình.
  - **Ngôn ngữ Java** cung cấp bộ máy dùng để xử lý ngoại lệ rất tuyệt vời. Việc xử lý này làm *hạn chế tối đa trường hợp hệ thống bị phá vỡ (crash) hay hệ thống bị ngắt đột ngột.* Tính năng này làm cho Java là một ngôn ngữ lập trình mạnh.

7.10 Cách xử lý lỗi truyền thống

Xử lý lỗi truyền thống thường là mỗi hàm lại thông báo trạng thái *thành công/thất bại* qua **1 mã lỗi**

- *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

Ví dụ:

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

- **Cài đặt mã xử lý tại nơi phát sinh ra lỗi**:
  - Trong đoạn code, *lỗi chia cho 0 được kiểm tra ngay tại chỗ, bên trong hàm devide*. Nếu denom bằng 0, giá trị error sẽ được gán bằng 1, báo hiệu một lỗi đã xảy ra.
- **Truyền trạng thái lên mức trên**:
  - *Trạng thái lỗi được truyền lên thông qua tham chiếu (int& error).* Điều này phù hợp với cách xử lý lỗi truyền thống, nơi kết quả hoặc trạng thái được báo hiệu qua các biến tham chiếu, tham số, hoặc giá trị trả lại.
- **Khó hiểu và dễ nhầm**:
  - Người gọi hàm *phải kiểm tra giá trị của error sau khi gọi hàm để xác định xem có lỗi xảy ra hay không.* Điều này có thể dẫn đến việc bỏ sót kiểm tra, làm chương trình dễ lỗi.

- **Không phải lúc nào cũng đầy đủ thông tin để xử lý**:
  - Hàm devide chỉ trả về một trạng thái lỗi cơ bản (error = 1). Nó *không cung cấp thêm thông tin chi tiết về bản chất của lỗi, chẳng hạn như thông báo cụ thể, loại lỗi, hay ngữ cảnh*.

7.11 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

7.11.1 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>

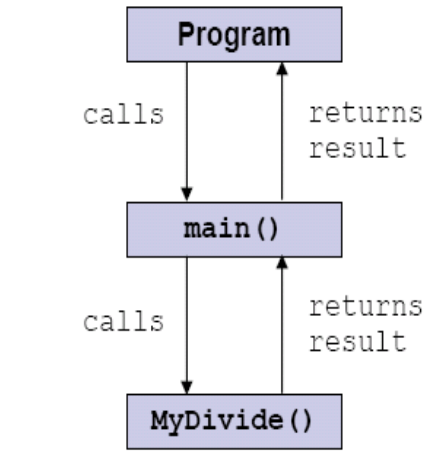
7.11.2 Cơ chế ngoại lệ

- 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*)

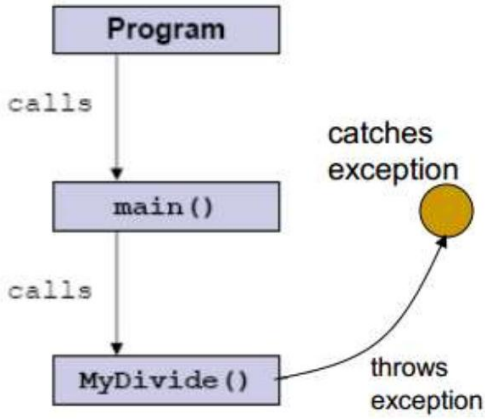
Ví dụ:

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

```
int main() {
      int x, y;
      cout << “Nhập 2 số: ”;
      cin >> x >> y;
      cout << “Kết quả x/y=”;
      cout << MyDivide(x, y) << “\n”;
      return 0;
}
```



- Quy trình **ném và bắt ngoại lệ:**
  - Giả sử người dùng *nhập mẫu số bằng 0*
  - Mã chương trình trong MyDivide() *tạo một ngoại lệ* (bằng cách nào đó) *và ném*
  - Khi một hàm ném một ngoại lệ, nó lập tức *kết thúc thực thi* và gửi ngoại lệ đó cho nơi gọi nó
  - Nếu main() có thể xử lý ngoại lệ, nó sẽ bắt và giải quyết ngoại lệ. Chẳng hạn yêu cầu người dùng nhập lại mẫu số.

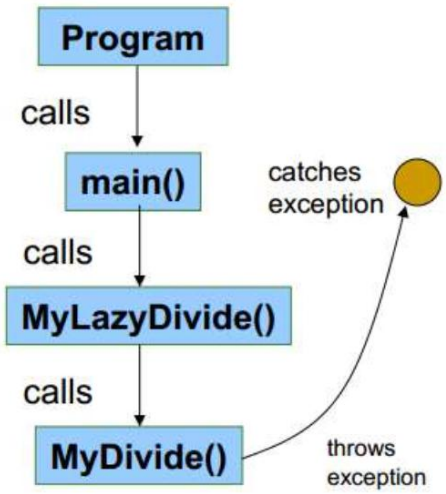


❖ Nếu 1 hàm ***không thể bắt ngoại lệ:*** Giả sử hàm **MyLazyDivide()** không thể bắt ngoại lệ do **MyDivide()** ném.

- Không phải hàm nào bắt được ngoại lệ cũng có thể bắt được **mọi loại** ngoại lệ;
- Chẳng hạn hàm **f()** bắt được các ngoại lệ **E1** nhưng không bắt được các ngoại lệ **E2**;
- Ngoại lệ đó sẽ được *chuyển lên mức trên cho **main()** bắt.*

❖ Nếu ***không có hàm nào bắt được ngoại lệ:***

- Tại mức thực thi cao nhất, chương trình tổng (nơi gọi hàm **main()**) sẽ ***bắt mọi ngoại lệ còn sót lại mà nó nhìn thấy***;
- Khi đó, chương trình **lập tức kết thúc:** Quy trình này *không hoàn toàn giống với các quy trình "bắt" thông thường* & ta **nên tránh không để chúng xảy ra.**



❖ Cơ chế xử lý ngoại lệ của C++ có 3 tính năng chính:

- **Tạo và ném ngoại lệ (sử dụng từ khóa throw):**
  - throw được dùng để ném ra một **ngoại lệ** khi chương trình gặp lỗi hoặc điều kiện bất thường mà nó không thể tiếp tục xử lý. Nó báo hiệu rằng **đã xảy ra sự cố và cần được xử lý ở nơi khác** trong chương trình.
  - **Cách hoạt động:**
    - Khi gặp một tình huống lỗi, chương trình **dừng thực thi tại vị trí throw** và **nhảy đến khối catch phù hợp**.
    - Để ném một ngoại lệ, ta dùng từ khóa throw, kèm theo đối tượng mà ta định ném: **throw <object>;**
    - Ngoại lệ có thể là ***bất kỳ giá trị nào, từ kiểu dữ liệu cơ bản*** (int, char, v.v.) ***đến đối tượng*** (ví dụ: string, hoặc các lớp ngoại lệ tùy chỉnh). **[throw 15; throw MyObj();]**

Ví dụ: MyDivide() ném ngoại lệ là một string

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

- Nếu **denominator == 0**, chương trình ném ra một **đối tượng string** chứa thông báo lỗi.
- Nếu không có khối catch phù hợp, chương trình sẽ kết thúc với ***lỗi***.

  - Trường hợp cần **cung cấp nhiều thông tin hơn cho hàm gọi** *(ví dụ: các giá trị đầu vào, trạng thái hệ thống, v.v. để người sử dụng hoặc chương trình gọi có thể hiểu rõ và xử lý tình huống hiệu quả hơn)*, ta tạo 1 *class dành riêng cho các ngoại lệ.* Ví dụ: ta cần cung cấp cho người dùng 2 số nguyên.

✍ **Ta có thể tạo 1 lớp ngoại lệ:**

```
class MyExceptionClass {
private:
    int a, b;  // Các tham số gây lỗi

public:
    // Hàm khởi tạo để nhận và lưu trữ thông tin
    MyExceptionClass(int a, int b) : a(a), b(b) {}

    // Hàm hủy (không bắt buộc, dùng mặc định nếu không cần đặc biệt hóa)
    ~MyExceptionClass() {}

    // Hàm trả về giá trị a và b
    int getA() { return a; }
    int getB() { return b; }
};
```

- Lớp này chứa hai biến a và b, đại diện cho ***các giá trị có liên quan đến lỗi.***
- Người dùng có thể gọi các hàm ***getA() và getB() để lấy giá trị này trong quá trình xử lý ngoại lệ.***

✍ Sau đó, dùng thể hiện của lớp vừa tạo để làm ngoại lệ.

```
int x = 10, y = 0;

// Kiểm tra điều kiện lỗi
if (y == 0) {
    throw MyExceptionClass(x, y);
     // Ném đối tượng ngoại lệ với thông tin x và y
}
```

✍ ***Đầy đủ:***

```
try {
    int x = 10, y = 0;

    // Thực hiện phép chia
    if (y == 0) {
        throw MyExceptionClass(x, y);  // Ném ngoại lệ
    }

    cout << "Kết quả: " << (x / y) << endl;
}
catch (MyExceptionClass &e) {  // Bắt đối tượng ngoại lệ
    // Xử lý lỗi và lấy thông tin chi tiết
    cout << "Đã xảy ra lỗi với các giá trị: "
         << "a = " << e.getA() << ", b = " << e.getB() << endl;
}
```

Khối catch nhận đối tượng ngoại lệ e và sử dụng các hàm getA() và getB() để lấy thông tin chi tiết về lỗi.

- **Tách logic xử lý ngoại lệ khỏi phần còn lại của hàm (sử dụng từ khóa try)**:
  - try được dùng để **bao bọc** các đoạn mã có khả năng gây ra lỗi. *Nó đóng vai trò như một "khối an toàn"*, trong đó:
    - ✍ Nếu *xảy ra ngoại lệ* (bị ném bằng throw), chương trình sẽ **bỏ qua phần còn lại của khối try** và nhảy đến khối catch phù hợp.
    - ✍ Nếu *không có ngoại lệ nào xảy ra*, chương trình tiếp tục thực thi bình thường.

- **Cách hoạt động:**
  - ***Đặt các đoạn mã có thể gây lỗi*** (ví dụ: chia cho 0, truy cập mảng ngoài phạm vi) ***vào bên trong khối try.***
  - Phải có ít nhất một khối catch đi kèm để xử lý ngoại lệ.
  - Kiểu dữ liệu **catch** phải trùng với kiểu dữ liệu của **throw**.

Ví dụ: **(Sử dụng lại hàm MyDivide ở ví dụ trên)**

```
int main() {
    int x = 10, y = 0;
    try {
        double result = MyDivide(x, y);  // Gọi hàm có khả năng ném ngoại lệ
        cout << "Kết quả: " << result << endl;
    }
    catch (string &err) {  // Bắt ngoại lệ kiểu string
        cout << "Đã xảy ra lỗi: " << err << endl;
    }
    return 0;
}
```

- Khối try chứa đoạn mã divide(x, y).
- Hàm divide ném ngoại lệ vì y == 0 => Chương trình ngay lập tức nhảy đến khối catch phù hợp, truyền đối tượng lỗi (err) cho nó.
- Khối catch xử lý ngoại lệ, hiển thị thông báo lỗi.

- **Bắt và giải quyết ngoại lệ (sử dụng từ khóa catch)**:
  - Ngoại lệ **được ném ra bởi throw** sẽ được **"bắt" trong một khối catch.**
  - Ở đây, bạn có thể xác định cách xử lý lỗi, như *hiển thị thông báo hoặc thực hiện biện pháp khắc phục.*

Ví dụ:

```
catch (string &s) {
    cout << s << endl;  // Xử lý ngoại lệ
}
```

Hình dung đơn giản:

- **try giống như một cái lưới an toàn.**
- **throw giống như một tín hiệu báo lỗi, được "ném" khi chương trình gặp sự cố.**
- Khối **catch** sẽ "bắt" lỗi này và xử lý nó.

7.11.3 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
};
```

- Mã liên quan đến ***thuật toán*** nằm trong khối **try**;
- Mã ***giải quyết lỗi*** đặt trong (các) khối **catch**.

Tách mã → chương trình dễ hiểu, dễ bảo trì hơn.

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

- Chú ý phải có ***dấu chấm phẩy đánh dấu kết thúc*** của toàn khối try-catch

```
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
};
```

❖ Ta viết lại hàm **main()** ban đầu để sử dụng khối **try – catch**:

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

int 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);
return 0;
}
```

✍ Có 2 thay đổi so với phiên bản trước:

- Dùng một biến bool làm cờ báo hiệu thành công hay thất bại và đưa khối try-catch vào trong một vòng do ... while để thực hiện nhiệm vụ cho đến khi thành công
- Ta đã sửa kiểu thành ***tham chiếu đến string*** thay vì **string**. Cách này hiệu quả hơn & tránh được slicing nếu ta làm việc với các ngoại lệ là thể hiện của các lớp dẫn xuất.

✍ Ngoại lệ bắt được được đặt tên ("s") để ta có thể truy nhập nó từ bên trong khối **catch**.

Lý do dùng tham chiếu (&) trong catch:

- **Tránh sao chép không cần thiết:** Khi một ngoại lệ được ném ra (thông qua throw), *việc sao chép đối tượng ngoại lệ vào khối catch có thể tốn tài nguyên*, đặc biệt nếu đối tượng ngoại lệ là một kiểu dữ liệu phức tạp hoặc lớn. Sử dụng tham chiếu giúp tránh việc này.

- **Duy trì tính toàn vẹn của đối tượng:** Khi bạn sử dụng tham chiếu, khối catch *làm việc trực tiếp với đối tượng ngoại lệ đã được ném ra*, đảm bảo rằng mọi thay đổi (nếu có) sẽ tác động trực tiếp lên nó.
    - Chẳng hạn như bộ nhớ **heap**, việc sao chép có thể làm mất tính đồng bộ giữa đối tượng gốc và bản sao.
- **Tuân thủ nguyên tắc hiệu năng tốt:** Trong C++, truyền tham chiếu (đặc biệt với kiểu dữ liệu lớn) là một thực tiễn tốt để giảm chi phí sao chép và cải thiện hiệu năng.

Khi nào không cần tham chiếu trong catch: Nếu đối tượng ngoại lệ là một kiểu dữ liệu cơ bản (như int hoặc char), bạn có thể sử dụng tham trị vì việc sao chép các kiểu dữ liệu nhỏ thường không đáng kể.

7.12 So khớp ngoại lệ (Exception Matching)

❖ 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**.

❖ **Chú ý:** Khi tìm các kiểu dữ liệu khớp với ngoại lệ, trình biên dịch nói chung sẽ **không thực hiện đổi kiểu tự động** *(thường đối với các kiểu nguyên thủy)*.
- 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**
- Do vậy trong đoạn mã sau, 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):

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

- 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) {…};
```

❖ Nếu ta muốn **bắt tất cả các ngoại lệ được ném** (kể cả các ngoại lệ ta không thể giải quyết)?
- ***Ví dụ:*** ta có thể cần chạy 1 số đoạn mã dọn dẹp trước khi hàm kết thúc (chẳng hạn dọn dẹp các vùng bộ nhớ cấp phát động).

❖ Để có một lệnh catch bắt được mọi ngoại lệ, **ta đặt dấu ba chấm (…) bên trong lệnh catch.**

❖ Chỉ nên **sử dụng nó cho lệnh catch cuối cùng** trong một khối try-catch (*nếu không, nó sẽ vô hiệu hóa các lệnh **catch** đứng sau).*

```
try {...}
catch(<exception type1>) {...}
catch(<exception type1>) {...}
//...
catch(<exception typeN>) {...}
catch(...) {...};
```

7.13 Chuyển tiếp ngoại lệ (Re-Throwing Exception)

- Trong 1 số trường hợp, ta có thể muốn *chuyển tiếp 1 ngoại lệ mà ta đã bắt* – thường là **khi ta không có đủ thông tin để giải quyết trọn vẹn ngoại lệ** đó *(hoặc để phần còn lại được xử lý ở mức cao hơn)*.
- This approach becomes particularly valuable when *managing exceptions at multiple levels* or when *additional actions need to be performed before resolving the exception.*
- Thông thường, ta sẽ chuyển tiếp 1 ngoại lệ **sau khi thực hiện 1 số công việc** dọn dẹp ***bên trong 1 lệnh catch "...";***
- Để chuyển tiếp, *ta dùng từ khóa **throw**, không kèm tham số,* từ bên trong lệnh **catch**.

```
catch(...)
{
    cout << "An exception was thrown." << endl;
    //...
    throw;  //re-throw exception
};
```

Ví dụ

```
#include <iostream>
#include <stdexcept>
using namespace std;

// Function to perform division
int divide(int numerator, int denominator)
{
	try {
		if (denominator == 0) {
			// Throw a runtime_error if attempting to divide by zero
			throw runtime_error("Division by zero!");
		}
		// Perform the division and return the result
		return numerator / denominator;
	}
	catch (const exception& e) {
		cout << "Caught exception in divide(): " << e.what() << endl;
		// Rethrow the caught exception to handle it at a higher level
		throw;
	}
}

// Function to calculate the sum of two numbers
int calculateSum(int a, int b)
{
	try {
		if (a < 0 || b < 0) {
			// Throw an invalid_argument exception for negative numbers
			throw invalid_argument("Negative numbers not allowed!");
		}
		// Calculate and return the sum
		return a + b;
	}
```

```
	catch (const exception& e) {
		cout << "Caught exception in calculateSum(): " << e.what() << endl;
		// Rethrow the caught exception to handle it at a higher level
		throw;
	}
}

int main()
{
	try {
		// Calculate the sum of 10 and the result of dividing 20 by 2
		int result = calculateSum(10, divide(20, 2));
		cout << "Result: " << result << endl;

		// Attempt to divide by zero, triggering an exception
		int invalidResult = calculateSum(5, divide(10, 0));
		cout << "Invalid Result: " << invalidResult << endl;
	}
	catch (const exception& e) {
		cout << "Caught exception in main: " << e.what() << endl;
		// Handle the exception at the highest level
	}

	return 0;
}
```

Kết quả:

```
Result: 20
Caught exception in divide(): Division by zero!
Caught exception in main: Division by zero!
```

Giải thích:

- Đầu tiên, chương trình tính tổng của 10 và kết quả của phép chia 20 cho 2 là 20. Kết quả này được in ra và *không có ngoại lệ nào được nêu ra* trong phần này.
- Tiếp theo, chương trình cố gắng chia cho số không khi tính tổng của 5 và kết quả của phép chia 10 cho 0. Điều này kích hoạt ngoại lệ "Chia cho số không!" được bắt trong hàm divide() và được ném lại. Ngoại lệ được ném lại sau đó được bắt trong hàm main() và được in ra là "Chia cho số không!" cùng với các thông báo xử lý ngoại lệ thích hợp.
- **Hàm divide()**: Bắt ngoại lệ ngay tại chỗ để ghi log, sau đó re-throw để cho hàm gọi tiếp tục xử lý.
- **Hàm calculateSum()**: Tương tự divide(), ghi log tại chỗ rồi re-throw để đưa ngoại lệ lên hàm gọi (main()).
- **Hàm main()**:
  - Gọi cả hai hàm divide() và calculateSum().
  - Bắt và xử lý toàn bộ ngoại lệ được ném từ các hàm con ở cấp độ cao nhất.
  - Xử lý tổng quát như thông báo lỗi, kết thúc chương trình an toàn.

Công dụng:

- **Ghi Log và Theo Dõi Lỗi Tại Nhiều Cấp:** Mỗi hàm có thể ghi log lỗi chi tiết trước khi chuyển ngoại lệ đi nơi khác. Điều này giúp việc debug dễ dàng hơn *(Giúp dễ dàng xác định được lỗi xuất phát từ đâu).*
- **Giữ Nguyên Thông Tin Lỗi:** Khi re-throw, thông tin của ngoại lệ ban đầu (ví dụ: runtime_error) không bị thay đổi.
- **Quản Lý Lỗi Ở Mức Tổng Quát:** main() có thể tập trung xử lý ngoại lệ cuối cùng (hiển thị thông báo, kết thúc chương trình, hoặc thực hiện hành động khắc phục).

- **Phân Tách Logic Xử Lý:** Mỗi hàm xử lý một phần của ngoại lệ (ghi log, giải phóng tài nguyên), nhưng *không cần phải chịu trách nhiệm xử lý hoàn toàn.*

Lưu ý khi sử dụng Re-Throw:

- **Cẩn thận khi mất dấu thông tin gốc:** Nếu bạn ném một ngoại lệ mới mà không giữ lại thông tin từ ngoại lệ gốc, bạn có thể làm mất các chi tiết quan trọng của lỗi ban đầu.
- **Không nên lạm dụng re-throw:** Chỉ nên sử dụng re-throw khi thực sự cần thiết. Quá nhiều tầng re-throw có thể làm chương trình khó theo dõi.

- Khi 1 ngoại lệ được chuyển tiếp, **mọi lệnh catch còn lại trong khối sẽ bị bỏ qua**, ngoại lệ được chuyển lên mức thực thi tiếp theo (như **ném 1 ngoại lệ mới**);
- Do ngoại lệ được ném lại mà không bị sửa đổi, ta có thể thấy được tầm quan trọng của việc sử dụng tham chiếu
    - Khi sử dụng tham số **không phải tham chiếu** (truyền bằng giá trị) trong lệnh catch, đối tượng ngoại lệ được sao chép. Nếu đối tượng ngoại lệ là một thực thể của lớp dẫn xuất, việc sao chép chỉ **giữ lại các thuộc tính của lớp cơ sở** (do cơ chế slicing), dẫn đến mất thông tin quan trọng.
        - **Object Slicing (Cắt bớt đối tượng):** Khi ngoại lệ được ném bởi một lớp dẫn xuất, nếu nó bị bắt bởi một catch không sử dụng tham chiếu (e.g., catch (MyException e)), chỉ phần dữ liệu của lớp cơ sở được giữ lại, dữ liệu riêng của lớp dẫn xuất sẽ bị loại bỏ.
    - Ngược lại, khi dùng **tham chiếu** (&), đối tượng gốc được truy cập mà không sao chép hoặc cắt bớt. Toàn bộ đối tượng, bao gồm cả dữ liệu của lớp dẫn xuất, được giữ nguyên, giúp giữ lại đầy đủ thông tin của lớp dẫn xuất.

Ví dụ:

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

class BaseException {
public:
    virtual string getMessage() const {
        return "Lỗi cơ bản.";
    }
};

class DerivedException : public BaseException {
public:
    string getMessage() const override {
        return "Lỗi cụ thể: File không tồn tại.";
    }
};

// Hàm ném ngoại lệ lớp dẫn xuất
void throwException() {
    throw DerivedException();
}

int main() {
    try {
        throwException();
    }
```

```
    catch (BaseException e) { // Không dùng tham chiếu
        cout << "Không dùng tham chiếu: " << e.getMessage() << endl;
    }
    catch (const BaseException& e) { // Dùng tham chiếu
        cout << "Dùng tham chiếu: " << e.getMessage() << endl;
    }
    return 0;
}
```

Kết quả chạy:

```
Không dùng tham chiếu: Lỗi cơ bản.
Dùng tham chiếu: Lỗi cụ thể: File không tồn tại.
```

- Trong trường hợp ***không dùng tham chiếu*** (catch (BaseException e)), khi đối tượng DerivedException được sao chép vào e, **chỉ phần BaseException được giữ lại**, làm mất thông tin đặc thù của lớp dẫn xuất.
- Khi ***dùng tham chiếu*** (catch (const BaseException& e)), **đối tượng gốc DerivedException được giữ nguyên,** và ta vẫn truy cập được dữ liệu đầy đủ của lớp dẫn xuất.

7.14 Quản lý bộ nhớ

- Ta đã biết: khi 1 ngoại lệ được ném, *hàm thực hiện đang chạy sẽ kết thúc*, điều khiển được trả về cho mức thực thi tiếp theo cao hơn cho đến khi gặp điểm bắt ngoại lệ. Stack sẽ được cuốn cho đến khi gặp điểm bắt ngoại lệ;
- Do đó quy trình dọn dẹp tự động xảy ra *như khi hàm kết thúc bình thường*, đó là:
  - Các đối tượng được *cấp phát tự động* bên trong hàm sẽ được thu hồi;
  - Trong các đối tượng trên, *đối với đối tượng bất kỳ mà constructor của nó đã được thực hiện hoàn chỉnh, destructor của nó sẽ được gọi.*
- Các đối tượng còn lại phải được *hủy 1 cách tường minh.*

- Nếu không tìm thấy lệnh **catch** tương ứng, sau khi mọi hàm đã kết thúc, 1 hàm thư viện đặc biệt **terminate()** sẽ được chạy (có thể coi đây là nơi bắt ngoại lệ cuối cùng);
- Trường hợp mặc định, **terminate()** gọi hàm **abort()**, hàm này sẽ lập tức kết thúc chương trình:
  - Trong trường hợp này, quy trình dọn dẹp không xảy ra, như vậy, *destructor của các đối tượng tĩnh & toàn cục sẽ không được gọi;*
  - Lưu ý rằng ***terminate()*** *cũng được gọi ngay khi có 1 ngoại lệ được ném trong quy trình dọn dẹp* (nghĩa là 1 destructor cho 1 đối tượng cấp phát tự động ném ngoại lệ trong quá trình **unwind**).

Ví dụ:

```
#include <iostream>
#include <exception>
using namespace std;

// Một lớp đơn giản với constructor và destructor
class MyObject {
public:
    MyObject(const string& name) : name(name) {
        cout << "Constructor: " << name << endl;
    }
    ~MyObject() {
        cout << "Destructor: " << name << endl;
    }

private:
    string name;
};
```

```
// Một hàm ném ngoại lệ
void throwException() {
    MyObject obj1("Obj1");
    throw runtime_error("Ngoại lệ được ném!");
}

void rethrowFromDestructor() {
    struct RethrowObj {
        ~RethrowObj() {
            throw runtime_error("Ngoại lệ trong destructor!");
        }
    };
    RethrowObj obj;
    throw runtime_error("Ngoại lệ chính!");
}

int main() {
    // Minh họa unwind stack khi gặp ngoại lệ
    try {
        MyObject objMain("ObjMain");
        throwException(); // Ném ngoại lệ và cuốn ngược stack
    } catch (const exception& e) {
        cout << "Caught in main: " << e.what() << endl;
    }

    // Minh họa terminate() khi ném ngoại lệ trong destructor
    try {
        rethrowFromDestructor();
    } catch (const exception& e) {
        cout << "Caught in main (dự kiến không in ra vì terminate() sẽ được gọi): "
             << e.what() << endl;
    }

    return 0;
}
```

Kết quả chạy:

```
Constructor: ObjMain
Constructor: Obj1
Destructor: Obj1
Destructor: ObjMain
Caught in main: Ngoại lệ được ném!
terminate called after throwing an instance of 'std::runtime_error'
  what():  Ngoại lệ trong destructor!
```

Phân tích kết quả:

`Constructor: ObjMain`

- Khi chương trình thực thi, đối tượng objMain được tạo trong khối try đầu tiên.
- Constructor của MyObject được gọi, với tên "ObjMain".
- Dòng "Constructor: ObjMain" được in ra từ constructor.

`Constructor: Obj1`

- Hàm throwException() được gọi từ khối try.
- Trong hàm throwException(), đối tượng obj1 được tạo với tên "Obj1".
- Constructor của MyObject được gọi, in ra dòng "Constructor: Obj1".

`Destructor: Obj1`

- Sau khi ngoại lệ được ném bởi throw runtime_error("Ngoại lệ được ném!"), quá trình **stack unwinding** (cuốn ngược stack) bắt đầu.
  - Cơ chế **cuốn ngược stack** (*stack unwinding*) xảy ra khi một ngoại lệ được ném (*throw*), và chương trình bắt đầu *giải phóng các tài nguyên được cấp phát tự động theo thứ tự ngược lại so với khi chúng được tạo ra (từ trên xuống dưới trong stack).*
- Trước khi thoát khỏi hàm throwException(), đối tượng obj1 (được cấp phát tự động trong hàm) phải được hủy.
- Destructor của obj1 được gọi, in ra dòng "Destructor: Obj1".

```
Destructor: ObjMain
```

- Ngoại lệ tiếp tục lan ra ngoài hàm throwException() và được bắt ở khối catch trong main().
- Trước khi khối catch xử lý ngoại lệ, đối tượng objMain trong phạm vi try cũng bị hủy.
- Destructor của objMain được gọi, in ra dòng "Destructor: ObjMain".

```
Caught in main: Ngoại lệ được ném!
```

- Ngoại lệ "Ngoại lệ được ném!" được xử lý trong khối catch của main().
- Dòng "Caught in main: Ngoại lệ được ném!" được in ra.

Đoạn thứ hai: terminate()

```
terminate called after throwing an instance of 'std::runtime_error'
  what():  Ngoại lệ trong destructor!
```

- Trong khối try thứ hai, hàm rethrowFromDestructor() được gọi.
- Trong hàm này:
  - Một đối tượng RethrowObj được tạo.
  - Ngoại lệ "Ngoại lệ chính!" được ném.
  - Trước khi thoát khỏi hàm (stack unwinding), destructor của RethrowObj được gọi.
  - Destructor của RethrowObj ném thêm ngoại lệ "Ngoại lệ trong destructor!".
- C++ **không thể xử lý hai ngoại lệ đồng thời**:
  - Ngoại lệ thứ nhất: "Ngoại lệ chính!".
  - Ngoại lệ thứ hai: "Ngoại lệ trong destructor!".
- Không có cơ hội để catch xử lý ngoại lệ này vì ném ngoại lệ trong quá trình cuốn ngược stack là vi phạm quy tắc.
- Điều này dẫn đến việc gọi **std::terminate().**
- **Chương trình kết thúc ngay lập tức**, không in thêm bất kỳ dòng nào từ khối catch.

Lưu ý

❖ **Cuốn ngược stack** xảy ra khi:
- Một ngoại lệ được ném.
- Tất cả các đối tượng tự động (cấp phát trong stack) trong phạm vi hiện tại được hủy theo thứ tự ngược lại so với khi chúng được tạo.
- Sau khi tất cả các đối tượng đã bị hủy, chương trình tìm kiếm khối catch phù hợp để xử lý ngoại lệ.

❖ **Destructor được gọi khi ngoại lệ xảy ra**:
- Trong quá trình stack unwinding, các đối tượng được tạo trước ngoại lệ vẫn bị hủy một cách tự động.
- Đây là lý do cả obj1 và objMain đều gọi destructor trước khi khối catch xử lý ngoại lệ.

❖ **Ngoại lệ trong destructor rất nguy hiểm**:
- Nếu có một ngoại lệ đang tồn tại, và destructor ném thêm ngoại lệ, C++ không thể xử lý cả hai cùng lúc.
- Kết quả: Gọi std::terminate(), chương trình kết thúc đột ngột.

❖ Có thể thay đổi hoạt động của **terminate()** bằng cách sử dụng hàm **set_terminate()**.

```
#include <iostream>
#include <exception>
using namespace std;

// Hàm terminate tùy chỉnh
void customTerminate() {
    cout << "Custom terminate handler: Chương trình kết thúc do ngoại lệ không được xử lý!" << endl;
    exit(1); // Kết thúc chương trình một cách có kiểm soát
}

int main() {
    set_terminate(customTerminate); // Đặt terminate tùy chỉnh

    try {
        throw runtime_error("Ngoại lệ không được bắt!");
    } catch (const char* msg) { // Cố tình không bắt runtime_error
        cout << "Caught: " << msg << endl;
    }

    return 0;
}
```

Kết quả: *Custom terminate handler: Chương trình kết thúc do ngoại lệ không được xử lý!*

7.15 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.* *(Xem các ví dụ trên).*
- 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. File này cũng đã **#include <exception>** nên khi dùng *không cần #include cả hai*
- Có hai lớp quan trọng được dẫn xuất trực tiếp từ 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.**
    - Một lý do là cả 2 lớp này đều có constructor nhận tham số là 1 **string** mà nó sẽ là kết quả trả về của hàm **what()**.

  - **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 (chẳng hạn chỉ số không hợp lệ)

❖ 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;
     }
}
```

❖ Ta phải sửa lệnh **catch cũ để bắt được ngoại lệ kiểu invalid_argument** (thay cho kiểu string trong phiên bản trước)

```
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);
}
```

7.16 Khai báo ngoại lệ

❖ Làm thế nào để user biết được 1 hàm/phương thức có thể ném những ngoại lệ nào?

❖ Đọc chú thích, tài liệu?

- Không phải lúc nào cũng có tài liệu & tài liệu đủ thông tin;
- Không tiện nếu phải kiểm tra cho mọi hàm.

❖ C++ cho phép khai báo 1 hàm có thể ném những loại ngoại lệ nào hoặc sẽ không ném ngoại lệ:

- Một phần của giao diện của hàm;
- Ví dụ: hàm **MyDivide()** có thể ném ngoại lệ **invalid_argument**.

❖ Cú pháp: từ khóa throw ở cuối lệnh khai báo hàm, tiếp theo là cặp ngoặc "**()**" chứa 1 hoặc nhiều tên kiểu (tách nhau bằng dấu "**,**"):

```
void MyFunction(...) throw(type1, type2, ..., typeN) {...}
```

❖ Hàm không bao giờ ném ngoại lệ:

```
void MyFunction(...) throw() {...}
```

❖ Cú pháp tương tự đối với phương thức:

```
bool FlightList::contains(Flight *f) const throw(char *) {...}
```

❖ *Nếu không có khai báo throw, hàm/phương thức có thể ném bất kỳ loại ngoại lệ nào.*

❖ Chuyện gì xảy ra nếu ta ***ném 1 ngoại lệ thuộc kiểu không có trong khai báo***?

- Nếu 1 hàm ném 1 ngoại lệ không thuộc các kiểu đã khai báo, hàm **unexcepted()** sẽ được gọi;
- Theo mặc định, **unexcepted()** gọi hàm **terminate()** mà ta đã nói đến;
- Tương tự **terminate()**, *hoạt động của **unexcepted()** cũng có thể được thay đổi bằng cách sử dụng hàm **set_unexpected()**.*

- Ta phải đặc biệt cẩn trọng khi làm việc với các cây thừa kế & các khai báo ngoại lệ;
- Giả sử có lớp cơ sở **B** chứa 1 phương thức ảo **foo()**:
  - Khai báo **foo()** có thể ném 2 loại ngoại lệ **e1** & **e2**:

```
class B
{
    public:
        void foo() throw(e1, e2);
};
```

- Giả sử **D** là lớp dẫn xuất của **B**, **D** định nghĩa lại **foo()**:
  - Cần có những hạn chế nào đối với khả năng ném ngoại lệ của **D**?

- Khai báo 1 phương thức về cốt yếu là để tuyên bố những gì người dùng có thể mong đợi từ phương thức đó:
  - Đưa ngoại lệ vào khai báo hàm/phương thức *hạn chế các loại đối tượng có thể được ném từ hàm/phương thức.*
- Khi có sự có mặt của **thừa kế & đa hình,** điều trên cũng phải áp dụng được;
- Do vậy, nếu 1 lớp dẫn xuất override 1 phương thức của lớp cơ sở, nó **không thể** bổ sung các kiểu ngoại lệ mới vào phương thức:
  - Nếu không, ai đó *truy nhập phương thức qua 1 con trỏ trỏ tới lớp cơ sở* có thể gặp phải 1 ngoại lệ mà họ không mong đợi (do nó không có trong khai báo của lớp cơ sở).
- Tuy nhiên, 1 lớp dẫn xuất được phép giảm bớt số loại ngoại lệ có thể ném.

- Ví dụ: nếu phiên bản **foo()** của lớp **B** có thể ném ngoại lệ thuộc kiểu **e1** & **e2**; phiên bản override của lớp **D** có thể chỉ được ném: Một tập con của **e1** và **e2**; Hoặc không ném ngoại lệ nào (***noexcept***).
  - Không vi phạm quy định của lớp cơ sở **B**.
- Tuy nhiên, **D** sẽ không thể bổ sung kiểu ngoại lệ mới **e3** cho các ngoại lệ mà **foo()** của **D** có thể ném:
  - Do việc đó vi phạm khẳng định rằng thể hiện của **D** "**là**" thể hiện của **B**.

- Các ràng buộc tương tự cũng áp dụng khi loại đối tượng được ném thuộc về ***1 cây thừa kế:***
  - Giả sử ta có cây thừa kế thứ hai gồm các lớp ngoại lệ, trong đó **BE** là lớp cơ sở và **DE** là lớp dẫn xuất.
  - Nếu phiên bản **foo()** của **B** chỉ ném đối tượng thuộc lớp **DE** (lớp dẫn xuất), thì phiên bản **foo()** của lớp **D** không thể ném thể hiện của lớp **BE** (lớp cơ sở).

7.17 Constructor - Destructor & Ngoại Lệ

- Constructor **không có giá trị trả về**, nên cách tốt nhất để thông báo khởi tạo **không** thành công là **ném ngoại lệ**.
- Cần chú ý để đảm bảo **constructor** không bao giờ để 1 đối tượng ở trạng thái khởi tạo dở: Dọn dẹp trước khi ném ngoại lệ.

```cpp
#include <iostream>
#include <stdexcept>
using namespace std;

class ResourceHandler {
public:
    ResourceHandler() {
        cout << "Constructor: Allocating resources" << endl;
        // Giả lập lỗi khởi tạo
        throw runtime_error("Error during initialization!");
    }

    ~ResourceHandler() {
        cout << "Destructor: Cleaning up resources" << endl;
    }
};

int main() {
    try {
        ResourceHandler obj; // Ném ngoại lệ trong constructor
    } catch (const exception& e) {
        cout << "Caught exception: " << e.what() << endl;
    }
    return 0;
}
```

Kết quả:

Constructor: Allocating resources
Caught exception: Error during initialization!

- Khi ngoại lệ được ném từ constructor, destructor sẽ **không được gọi**, vì đối tượng chưa được khởi tạo hoàn chỉnh.
    - Khi ngoại lệ xảy ra, quá trình khởi tạo bị dừng, đối tượng obj không được tạo ra hoàn chỉnh và do đó **không tồn tại hợp lệ**.
    - Destructor chỉ được gọi cho các đối tượng đã được khởi tạo đầy đủ (tức là constructor phải hoàn tất mà không xảy ra lỗi). Trong trường hợp này, constructor không hoàn thành, nên không có đối tượng obj nào hợp lệ để hủy. Vì vậy, destructor ~ResourceHandler() không được gọi.
- Tài nguyên đã cấp phát **cần được dọn dẹp trước khi ném ngoại lệ**.

❖ **Không nên để ngoại lệ được ném từ destructor;**

❖ Nếu destructor trực tiếp hoặc gián tiếp ném ngoại lệ, **chương trình sẽ gọi terminate() & kết thúc.**

- ***Hậu quả:*** chương trình nhiều lỗi có thể kết thúc bất ngờ mà ta không nhìn thấy được nguồn gốc có thể của lỗi.

```
#include <iostream>
#include <stdexcept>
using namespace std;

class MyObject {
public:
    MyObject() {
        cout << "Constructor called" << endl;
    }

    ~MyObject() {
        cout << "Destructor called" << endl;
        // Không nên ném ngoại lệ từ đây
        throw runtime_error("Exception from destructor");
    }
};

int main() {
    try {
        MyObject obj;
        throw runtime_error("Main exception");
    } catch (const exception& e) {
        cout << "Caught exception: " << e.what() << endl;
    }
    return 0;
}
```

Kết quả:

```
Constructor called
Destructor called
terminate called after throwing an instance of 'std::runtime_error'
what():  Exception from destructor
```

Giải thích: Destructor ném ngoại lệ trong khi main() cũng ném ngoại lệ => terminate() được gọi, và chương trình kết thúc mà không xử lý được ngoại lệ gốc.

❖ Vậy, **destructor** *cần bắt tất cả các ngoại lệ có thể được ném từ các hàm được gọi* từ đây để đảm bảo không ném thêm ngoại lệ.

```
#include <iostream>
#include <exception>
using namespace std;

// Lớp mô phỏng việc cấp phát tài nguyên
class ResourceHandler {
public:
    ResourceHandler() {
        cout << "Constructor: Allocating resources" << endl;
    }

    ~ResourceHandler() {
        cout << "Destructor: Cleaning up resources" << endl;
```

```
        try {
            // Giả lập lỗi trong destructor
            throw runtime_error("Error in destructor!");
        } catch (const exception& e) {
            // Bắt ngoại lệ để tránh terminate()
            cout << "Caught exception in destructor: " << e.what() << endl;
        }
    }
};

int main() {
    try {
        ResourceHandler obj; // Tạo đối tượng ResourceHandler
        throw runtime_error("Error during main process!"); // Ném ngoại lệ từ main
    } catch (const exception& e) {
        cout << "Caught exception in main: " << e.what() << endl;
    }
    return 0;
}
```

Kết quả khi chạy mã:

```
Constructor: Allocating resources
Destructor: Cleaning up resources
Caught exception in destructor: Error in destructor!
Caught exception in main: Error during main process!
```

Giải thích chi tiết:

- **Tạo đối tượng ResourceHandler obj trong main:** Constructor của ResourceHandler được gọi, in ra: "Constructor: Allocating resources".
- **Ngoại lệ ném từ main:**
  - Lệnh throw runtime_error("Error during main process!"); ném một ngoại lệ từ hàm main.
  - Quá trình stack unwinding bắt đầu: đối tượng obj cần được hủy.
- **Destructor của ResourceHandler được gọi:**
  - Destructor in ra: "Destructor: Cleaning up resources".
  - Trong destructor, ngoại lệ runtime_error("Error in destructor!") được ném, nhưng nó **được bắt ngay bên trong destructor** bởi khối try-catch.
  - Thông báo: "Caught exception in destructor: Error in destructor!" được in ra.
- **Ngoại lệ từ main được xử lý:**
  - Sau khi destructor xử lý xong và không ném ngoại lệ ra ngoài, quá trình stack unwinding tiếp tục.
  - Ngoại lệ từ main được bắt bởi khối catch trong main.
  - Thông báo: "Caught exception in main: Error during main process!" được in ra.

C. GHI LOG (LOGGING)

Là quá trình ghi lại các thông tin về hoạt động của một chương trình hoặc hệ thống. Các thông tin này thường được lưu trữ vào file, cơ sở dữ liệu, hoặc hiển thị ra màn hình để phục vụ cho mục đích giám sát, gỡ lỗi, và phân tích.

Mục đích của ghi log

- **Theo dõi hoạt động:** Ghi lại các sự kiện xảy ra trong chương trình, ví dụ như người dùng đăng nhập, xử lý dữ liệu, hoặc gửi yêu cầu đến máy chủ.
- **Phát hiện lỗi:** Giúp lập trình viên biết được chương trình gặp lỗi ở đâu và lý do gây lỗi.
- **Phân tích hiệu suất:** Ghi lại thời gian thực hiện của các tác vụ để tối ưu hóa chương trình.

- **Giám sát hệ thống:** Giúp quản trị viên hoặc đội kỹ thuật kiểm tra tình trạng hệ thống và phát hiện các vấn đề tiềm ẩn.

Ví dụ:

Trong ví dụ này, log được ghi ra màn hình console.

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

void processTask() {
    cout << "[INFO]: Bắt đầu xử lý công việc..." << endl;

    try {
    // Gây ra lỗi
        throw string("Đã xảy ra lỗi trong quá trình xử lý!");
    } catch (string &e) {
        cout << "[ERROR]: " << e << endl;   // Ghi log lỗi
    }

    cout << "[INFO]: Kết thúc xử lý công việc." << endl;
}

int main() {
    processTask();
    return 0;
}
```

Kết quả:

```
[INFO]: Bắt đầu xử lý công việc...
[ERROR]: Đã xảy ra lỗi trong quá trình xử lý!
[INFO]: Kết thúc xử lý công việc.
```

Ngoài ra có thể ghi log vào file thay vì hiển thị trên console để lưu trữ lâu dài.

Các mức độ log

- **DEBUG:** Thông tin chi tiết phục vụ cho việc gỡ lỗi (dành cho lập trình viên).
- **INFO:** Thông tin về hoạt động bình thường của chương trình.
- **WARNING:** Cảnh báo về các tình huống không bình thường nhưng chưa gây lỗi.
- **ERROR:** Các lỗi cần được chú ý nhưng *không làm dừng chương trình.*
- **CRITICAL:** Lỗi nghiêm trọng khiến *chương trình không thể tiếp tục* hoạt động.

Lợi ích của ghi log

- **Hỗ trợ gỡ lỗi:** Ghi log cho biết lỗi xảy ra ở đâu và vì sao.
- **Hiểu rõ hơn về chương trình:** Ghi lại luồng hoạt động của ứng dụng.
- **Phân tích lịch sử:** Log lưu trữ các sự kiện để xem xét lại sau này.
- **Tự động hóa giám sát:** Dùng log để phát hiện các sự cố tự động qua công cụ giám sát.

Lưu ý khi ghi log

- **Không ghi thông tin nhạy cảm:** Tránh ghi mật khẩu hoặc dữ liệu nhạy cảm vào log.
- **Quản lý kích thước log:** Log file cần được xoay vòng hoặc xóa khi quá lớn.
- **Không lạm dụng:** Ghi quá nhiều log có thể làm chậm chương trình.