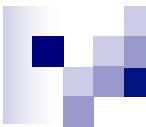


## CHƯƠNG 8:

# Ngoại lệ (Exception)

TS. LÊ THỊ MỸ HẠNH  
Bộ môn Công nghệ Phần mềm  
Khoa Công Nghệ Thông Tin  
Đại học Bách khoa – Đại học Đà Nẵng



# Ngoại lệ

- Xử lý lỗi
- Xử lý lỗi theo kiểu truyền thống
- Ý tưởng về ngoại lệ trong C++
- Giới thiệu về ngoại lệ
- Cú pháp
- Ném ngoại lệ
- **try-catch**
- Khớp ngoại lệ
- Chuyển tiếp ngoại lệ
- Chuyện hậu trường
- Lớp **exception**
- Khai báo ngoại lệ

# Xử lý lỗi

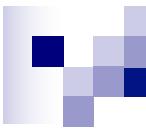
- Chương trình nào cũng có khả năng gặp phải các tình huống không mong muốn
  - người dùng nhập dữ liệu không hợp lệ
  - đĩa cứng bị đầy,
  - file cần mở bị khóa
  - đối số cho hàm không hợp lệ
- Xử lý như thế nào?
  - Một chương trình không quan trọng có thể dừng lại
  - Chương trình điều khiển không lưu? Điều khiển máy bay?

# Xử lý lỗi

```
double MyDivide(double numerator, double denominator) {  
    if (denominator == 0.0) {  
        // Do something to indicate that an error occurred  
    }  
    else {  
        return numerator / denominator;  
    }  
}
```

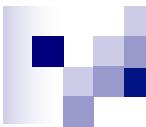
xử lý lỗi  
như thế nào  
bây giờ?

- Mã thư viện (nơi gấp lỗi) thường không có đủ thông tin để xử lý lỗi
  - Cần có cơ chế để mã thư viện báo cho mã ứng dụng rằng nó vì một lý do nào đó không thể tiếp tục chạy được, để mã ứng dụng xử lý tùy theo tình huống



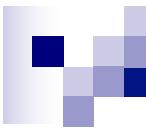
# Xử lý lỗi truyền thông

- Xử lý lỗi truyền thông thường là mỗi hàm lại thông báo trạng thái thành công/thất bại qua một mã lỗi
  - biến toàn cục (chẳng hạn **errno**)
  - giá trị trả về
    - **int remove ( const char \* filename );**
    - tham số phụ là tham chiếu
      - **double MyDivide(double numerator,  
double denominator,  
int &status);**



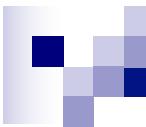
# Xử lý lỗi truyền thông – Hạn chế

- phải có lệnh kiểm tra lỗi sau *mỗi* lời gọi hàm
  - code trông rối rắm, dài, khó đọc
- lập trình viên ứng dụng quên kiểm tra, hoặc cố tình bỏ qua
  - bản chất con người
  - lập trình viên ứng dụng thường không có kinh nghiệm bằng lập trình viên thư viện
- rắc rối khi đầy thông báo lỗi từ hàm được gọi sang hàm gọi vì từ một hàm ta chỉ có thể trả về một kiểu thông báo lỗi



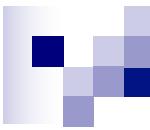
# C++ exception

- Exception – ngoại lệ là cơ chế thông báo và xử lý lỗi giải quyết được các vấn đề kể trên
  - Tách được phần xử lý lỗi ra khỏi phần thuật toán chính
  - cho phép 1 hàm thông báo về nhiều loại ngoại lệ
  - Không phải hàm nào cũng phải xử lý lỗi
    - nếu có một số hàm gọi thành chuỗi, ngoại lệ chỉ cần được xử lý tại một hàm là đủ
  - không thể bỏ qua ngoại lệ, nếu không, chương trình sẽ kết thúc.
- Tóm lại, cơ chế ngoại lệ mềm dẻo hơn kiểu xử lý lỗi truyền thống



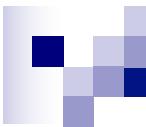
# Các kiểu ngoại lệ

- 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

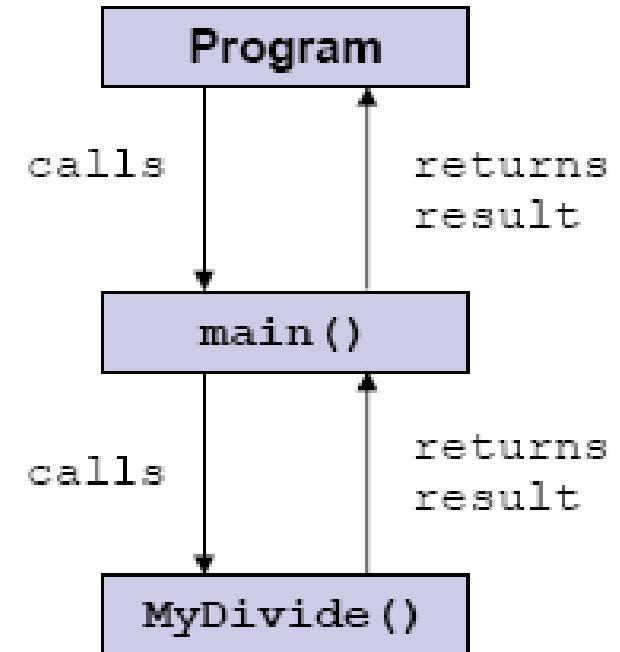
- Quá trình truyền ngoại lệ từ ngữ cảnh thực thi hiện hành tới mức thực thi cao hơn gọi là **ném một ngoại lệ** (throw an exception)
  - vị trí trong mã của hàm nơi ngoại lệ được ném được gọi là **điểm ném** (throw point)
- Khi một ngữ cảnh thực thi tiếp nhận và truy nhập một ngoại lệ, nó được coi là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

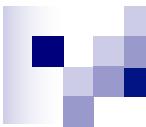


# Cơ chế ngoại lệ

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

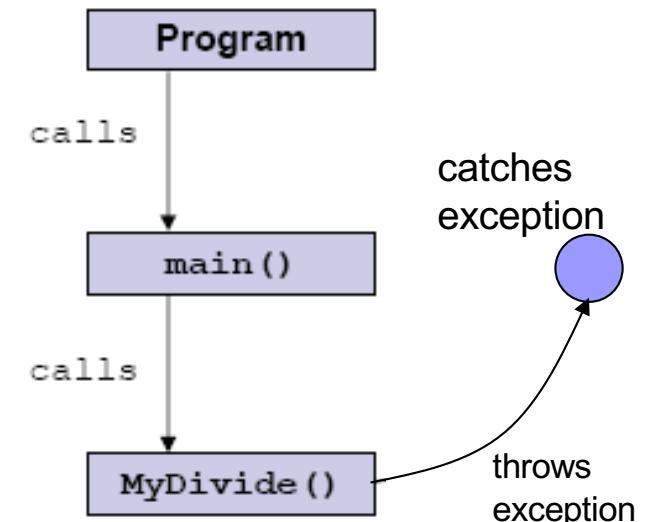
```
...
int main() {
    int x, y;
    // Prompt user for two numbers
    cout << "Enter two integers, separated by a space: ";
    cin >> x >> y;
    cout << "The first number divided by the second is: "
    << MyDivide(x, y) << endl;
}
```

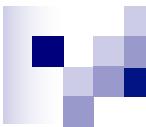




# Cơ chế ngoại lệ

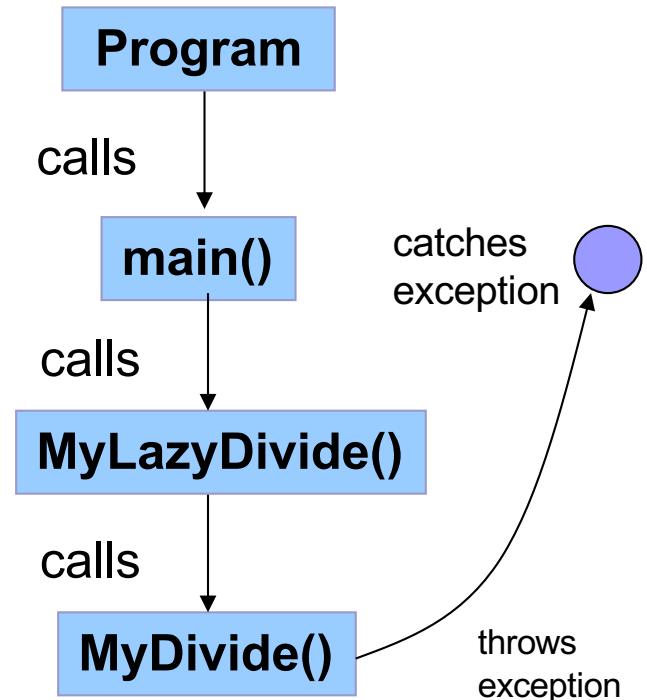
- Quy trình ném và bắt ngoại lệ
  - giả sử người dùng nhập mẫu số bằng 0
  - Mã chương trình trong MyDivide() tạo một ngoại lệ (bằng cách nào đó) và ném
  - Khi một hàm ném một ngoại lệ, nó lập tức kết thúc thực thi và gửi ngoại lệ đó cho nơi gọi nó
  - Nếu main() có thể xử lý ngoại lệ, nó sẽ bắt và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

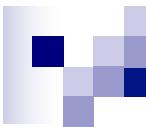




# Cơ chế ngoại lệ

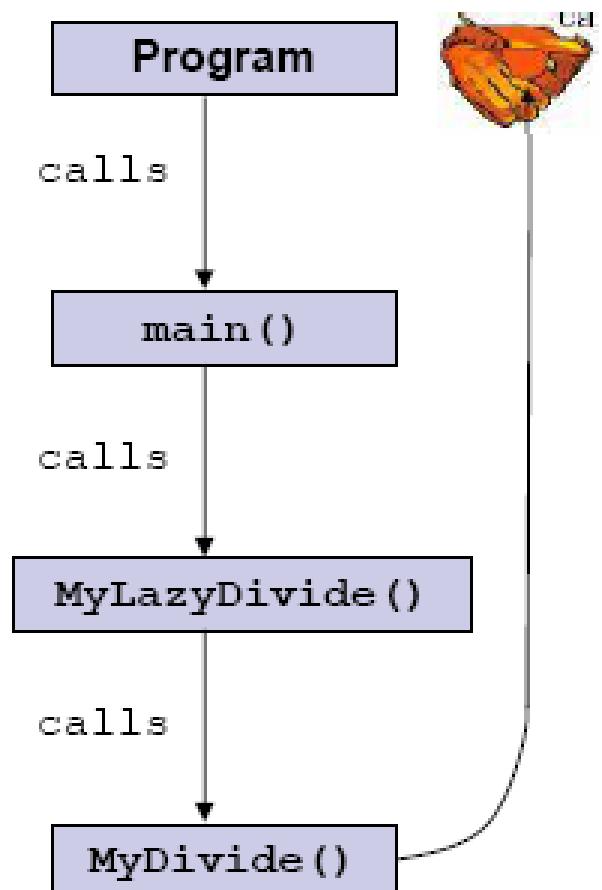
- Nếu một hàm không thể bắt ngoại lệ
  - giả sử hàm **MyLazyDivide()** không thể bắt ngoại lệ do **MyDivide()** ném
- Không phải hàm nào bắt được ngoại lệ cũng có thể bắt được **mọi loại** ngoại lệ
  - Chẳng hạn hàm **f()** bắt được các ngoại lệ loại **E1** nhưng không bắt được các ngoại lệ loại **E2**.
- Ngoại lệ đó sẽ được chuyển lên mức trên cho **main()** bắt

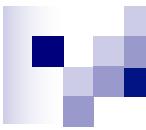




# Cơ chế ngoại lệ

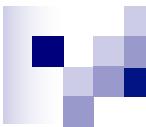
- Nếu không có hàm nào bắt được ngoại lệ?
  - Hiện giờ ví dụ của ta vẫn chưa có đoạn mã bắt ngoại lệ nào, nếu có một ngoại lệ được ném, nó sẽ được chuyển qua tất cả các hàm
- Tại mức thực thi cao nhất, chương trình tổng (nơi gọi hàm **main()**) sẽ bắt mọi ngoại lệ còn sót lại mà nó nhìn thấy
- Khi đó, chương trình lập tức kết thúc
  - Quy trình này không hoàn toàn giống với các quy trình "bắt" thông thường và ta nên tránh không để chúng xảy ra.





# Cú pháp xử lý ngoại lệ

- Ta đã có các khái niệm cơ bản về xử lý ngoại lệ, sử dụng cơ chế đó trong C++ như thế nào?
- 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 lôgic xử lý ngoại lệ trong một hàm ra khỏi phần còn lại của hàm (sử dụng từ khoá **try**)



# throw – Ném ngoại lệ

- Để ném một ngoại lệ, ta dùng từ khoá **throw**, kèm theo đối tượng mà ta định ném

```
throw <object>;  
//Throws an exception (replace "<object>"  
//with whatever type of object is to be thrown)
```

- Ta có thể dùng **mọi thứ** làm ngoại lệ, kể cả giá trị thuộc kiểu có sẵn

```
throw 15; // Throws the int 15 as an exception  
throw MyObj(...); // Throws an instance of MyObj as an exception
```

- 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;  
    }  
}
```

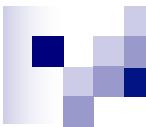
# throw – Ném ngoại lệ

- Trường hợp cần cung cấp nhiều thông tin hơn cho hàm gọi, ta tạo một class dành riêng cho các ngoại lệ
- Ví dụ, ta cần cung cấp cho người dùng 2 số nguyên
  - Ta có thể tạo một lớp ngoại lệ:

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

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

```
int x, y;
if(...) throw MyExceptionClass(x, y);
```

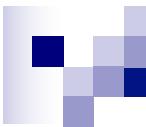


# Khối try – catch

- 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
- Cú pháp chung cho khối **try – catch**:

```
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**
- Việc tách mã làm cho chương trình dễ hiểu, dễ bảo trì hơn

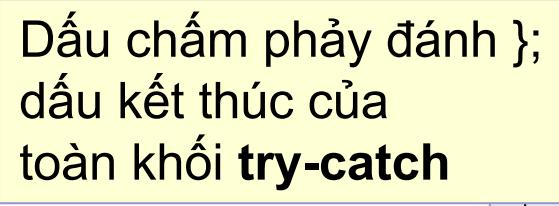


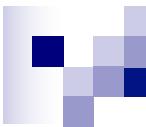
# Khối try – catch

- 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  
};
```

Dấu chấm phẩy đánh };  
dấu kết thúc của  
tòan khối try-catch





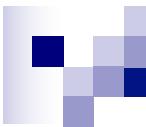
# Khối try – catch

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

```
int main() {  
    int x, y;  
    double result;  
    // Prompt user for two numbers  
    cout << "Enter two integers, separated by a space: ";  
    cin >> x >> y;  
    try {  
        result = MyDivide(x, y); //Use temporary variable to separate  
        // call  
        // to MyDivide() from output code  
    }  
    catch (string) {  
        // resolve error  
    };  
    cout<<"The first number divided by the second is:<< result<<  
    endl;  
}
```

Chú ý: **MyDivide()** ném  
ngoại lệ là một **string**

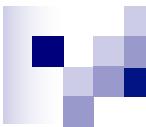
Giải quyết lỗi như thế nào?



# Khối try – catch

```
int main() {  
    int x, y;  
    double result;  
    bool success;  
  
    do {  
        success = true; // Set success to true  
        cout << "Enter two integers, separated by a space: "; cin >> x >> y;  
        try {  
            result = MyDivide(x, y);  
        }  
        catch (string& s) {  
            cout << s << endl;  
            success = false; // An exception occurred - set success to false  
        };  
    } while (success == false); // If success == false, repeat loop  
    cout << "The first number divided by the second is:" << result << endl;  
}
```

Có thể 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



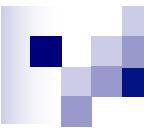
# Khối try – catch

## ■ Nhận xét về slide trước

- Tại lệnh **catch**, có hai thay đổi so với phiên bản trước:
  - Ta đã sửa kiểu thành tham chiếu đến **string** thay vì **string**. Cách này hiệu quả hơn và 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**

# Exception Matching (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 ăn khớp, ngoại lệ xem như là được giải quyết, không cần tiếp tục tìm kiếm.
  - Nếu không tìm thấy, mức thực thi hiện hành bị kết thúc và ngoại lệ sẽ 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
  - Nếu một ngoại lệ kiểu **float** được ném, nó sẽ không khớp với một khối **catch** cho ngoại lệ kiểu **int**
  - Tuy nhiên, 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**



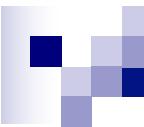
# Exception Matching

- Do vậy, trong đoạn mã sau, mọi ngoại lệ là đối tượng được sinh từ cây **MotorVehicle** sẽ khớp với lệnh **catch** đầu tiên (các lệnh còn lại sẽ không bao giờ chạy)

```
try { ... }  
    // Matches everything from the motor vehicle hierarchy  
    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) {...} ;
```



# Exception Matching

- 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 một 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**

```
catch (...) {...}; // This will catch any exception
```

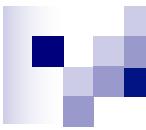
- Do tham số ba chấm bắt được mọi ngoại lệ, ta chỉ nên sử dụng nó cho lệnh **catch** cuối cùng trong một khối **try-catch** (nếu không, nó sẽ vô hiệu hóa các lệnh **catch** đứng sau)

```
try {...}  
catch (<exception type1>) {...}  
catch (<exception type2>) {...}  
...  
catch (<exception typeN>) {...}  
catch (...) {...};
```

# Re-Throwing Exception (chuyển tiếp ngoại lệ)

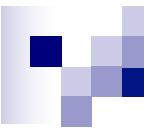
- Trong một số trường hợp, ta có thể muốn chuyển tiếp một ngoại lệ mà ta đã bắt - thường là khi ta không có đủ Thông tin để giải quyết trọn vẹn ngoại lệ đó
  - Thông thường, ta sẽ chuyển tiếp một ngoại lệ sau khi thực hiện một số công việc đơn dẹp bên trong một lệnh **catch** ba chấm
  - Để chuyển tiếp, ta dùng từ khoá **throw**, không kèm tham số, từ bên trong lệnh **catch**.

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



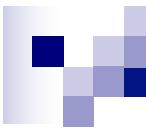
# Chuyển tiếp ngoại lệ

- Khi một 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 khi ném một 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
  - Nếu một lệnh **catch** cho lớp cơ sở với tham số không phải tham chiếu bắt được một ngoại lệ thuộc lớp dẫn xuất, nó sẽ **slice** (thay đổi vĩnh viễn) đối tượng để đối tượng đó chỉ còn chứa các thuộc tính của lớp cơ sở
  - Bằng cách sử dụng tham chiếu, ta có thể đảm bảo rằng ngoại lệ được chuyển tiếp mà không bị thay đổi.



# Quản lý bộ nhớ

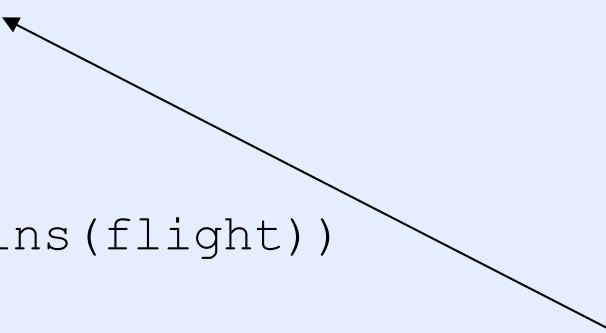
- Ta đã biết: khi một ngoại lệ được ném, hàm hiện đang chạy sẽ kết thúc, điều khiển được trả về cho mức thực thi tiếp theo cao hơn cho đến khi gặp điểm bắt ngoại lệ. stack sẽ được cuốn (unwind) cho đến khi gặp điểm bắt ngoại lệ
- Do đó quy trình dọn dẹp tự động xảy ra giống như khi hàm kết thúc bình thường, đó là:
  - Các đối tượng được cấp phát tự động bên trong hàm sẽ được thu hồi
  - Trong các đối tượng đó, 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 một cách tường minh.



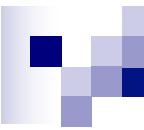
# Quản lý bộ nhớ

## ■ Ví dụ

```
bool FlightList::contains(Flight *f) const throw(char*) {  
    Flight flight;  
    Airport *a = new Airport("SFO", "San Francisco");  
    ...  
    throw "Out of Bounds";  
    ...  
}  
...  
try {  
    if (flightList->contains(flight))  
        ...  
}  
catch (char* str) {  
    // Handle the error  
}
```

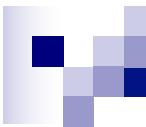


Khi ngoại lệ được ném,  
destructor cho **flight**  
được tự động gọi, nhưng  
destructor của **a** thì không.



# Quản lý bộ nhớ

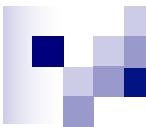
- Nếu không tìm thấy lệnh **catch** tương ứng, sau khi mọi hàm đã kết thúc, một hàm thư viện đặc biệt **terminate()** sẽ được chạy (có thể coi đây là nơi bắt ngoại lệ cuối cùng)
- Trường hợp mặc định, **terminate()** gọi hàm **abort()**, hàm này sẽ lập tức kết thúc chương trình
  - Trong trường hợp này,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



# Lớp exception

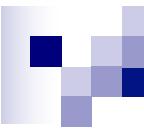
- Để tích hợp hơn nữa các ngoại lệ vào ngôn ngữ C++, lớp **exception** đã được đưa vào thư viện chuẩn
  - sử dụng `#include <exception>` và `namespace std`
- Sử dụng thư viện này, ta có thể ném các thề hiện của **exception** hoặc tạo các lớp dẫn xuất từ đó
- Lớp **exception** có một hàm ảo **what()**,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

```
try { ... }
catch (exception e) {
    cout << e.what();
}
```



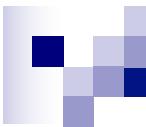
# Lớp exception

- 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**
  - File này cũng đã **#include <exception>** nên khi dùng không cần **#include** cả hai
- Trong đó có hai lớp quan trọng được dẫn xuất trực tiếp từ **exception**:
  - **runtime\_error**
  - **logic\_error**



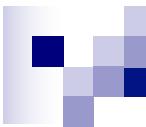
# Lớp exception

- **runtime\_error** dùng để đại diện cho 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** dùng cho các lỗi trong lôgic 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 hai lớp này đều có constructor nhận tham số là một **string** mà nó sẽ là kết quả trả về của hàm **what()**



# Lớp exception

- **runtime\_error** có các lớp dẫn xuất sau:
  - **range\_error** điều kiện sau (post-condition) bị vi phạm xảy ra tràn số học
  - **overflow\_error** không thể cấp phát bộ nhớ
  - **bad\_alloc**
- **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)



# Lớp exception

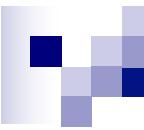
- Ta có thể viết lại hàm **MyDivide()** để sử dụng các ngoại lệ chuẩn tương ứng như sau:

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

- Ta sẽ phải sửa lệnh **catch** cũ để bắt được ngoại lệ kiểu **invalid\_argument** (thay cho kiểu **string** trong phiên bản trước)

# Lớp exception

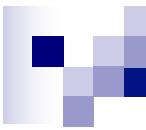
```
int main() {  
    int x, y;  
    double result;  
    do {  
        cout << "Enter two integers, separated by a space: ";  
        cin >> x >> y;  
        try {  
            result = MyDivide(x, y);  
        }  
        catch (invalid_argument& e) {  
            cout << e.what() << endl;  
            continue; // "Restart" the loop  
        };  
    } while (0); // If we reach here, exit loop  
    cout<<"The first number divided by the second is:"<<result<<endl;  
}
```



# Khai báo ngoại lệ

- Vậy, làm thế nào để user biết được một hàm/phương thức có thể ném những loại ngoại lệ nào?
- Đọc chú thích, tài liệu?
  - không phải lúc nào cũng có tài liệu và tài liệu đủ thông tin
  - không tiện nếu phải kiểm tra cho mọi hàm
- C++ cho phép khai báo một 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**

```
double MyDivide(double numerator, double denominator) throw (invalid_argument);
```



# Khai báo ngoại lệ

- Cú pháp: từ khoá **throw** ở cuối lệnh khai báo hàm, tiếp theo là cặp ngoặc đơn chứa một hoặc nhiều tên kiểu (tách nhau bằng dấu phẩy)

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

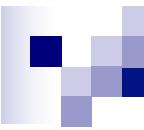
- hàm không bao giờ ném ngoại lệ:

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

- Cú pháp tương tự đối với phương thức

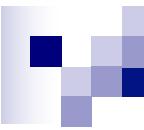
```
bool FlightList::contains(Flight *f) const throw(char*) { ... }
```

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



# Khai báo ngoại lệ

- Chuyện gì xảy ra nếu ta ném một ngoại lệ thuộc kiểu không có trong khai báo?
  - Nếu một hàm ném một ngoại lệ không thuộc các kiểu đã khai báo, hàm **unexpected()** sẽ được gọi
  - Theo mặc định, **unexpected()** gọi hàm **terminate()** mà ta đã nói đến
  - Tương tự **terminate()**, hoạt động của **unexpected()** cũng có thể được thay đổi bằng cách sử dụng hàm **set\_unexpected()**

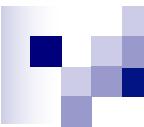


# Khai báo ngoại lệ

- Ta phải đặc biệt thận trọng khi làm việc với các cây thừa kế và các khai báo ngoại lệ
- Giả sử ta có lớp cơ sở **B** chứa một phương thức ảo **foo()**
  - Ta khai báo rằng **foo()** có thể ném hai loại ngoại lệ **e1** và **e2**

```
class B {  
    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**?

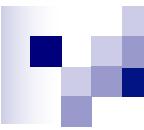


# Khai báo ngoại lệ

- Nhớ lại rằng một khai báo phương thức về cốt yếu là để tuyên bố những gì người dùng có thể mong đợi từ phương thức đó
  - Ta đã nói rằng 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 và đa hình, điều trên cũng phải áp dụng được
- Do vậy, nếu một lớp dẫn xuất override một 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 một con trỏ tới lớp cơ sở có thể gặp phải một ngoại lệ mà họ không mong đợi (do nó không có trong khai báo của lớp cơ sở)
- Tuy nhiên, một lớp dẫn xuất **được phép** giảm bớt số loại ngoại lệ có thể ném

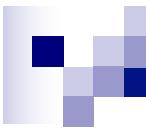
# Khai báo ngoại lệ

- Ví dụ, nếu phiên bản `foo()` của lớp B có thể ném ngoại lệ thuộc kiểu `e1` và `e2`, phiên bản override của lớp D có thể chỉ được ném ngoại lệ thuộc loại `e1`
  - Không vi phạm quy định của lớp cơ sở
- 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



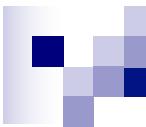
# Khai báo ngoại lệ

- Các ràng buộc tương tự cũng áp dụng khi loại đối tượng được ném thuộc về một 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 **BE** (lớp cơ sở)



# constructor và ngoại lệ

- Cách tốt để thông báo việc khởi tạo không thành công
  - constructor không có giá trị trả về
- Cần chú ý để đảm bảo constructor không bao giờ để một đối tượng ở trạng thái khởi tạo dở
  - gọn dẹp trước khi ném ngoại lệ



# destructor và ngoại lệ

- Không nên để ngoại lệ được ném từ destructor
- Nếu destructor trực tiếp hoặc gián tiếp ném ngoại lệ, chương trình sẽ kết thúc
  - một hậu quả: chương trình nhiều lỗi có thể kết thúc bất ngờ mà ta không nhìn thấy được nguồn gốc có thể của lỗi
- Vậy, destructor cần bắt tất cả các ngoại lệ có thể được ném từ các hàm được gọi từ đây