C++ PROGRAMMING

DR. YIHSIANG LIOW (JULY 10, 2025)

Contents

71. Exceptions and assertions

OBJECTIVES:

- Understand the flow of control when an exception is thrown.
- Write code to throw exception
- Write exception handling code to catch and process exceptions.
- Write assertions.
- Understand the difference between exceptions and assertions.

Exceptions

Exceptions are error control devices. Here's the big picture ...

Suppose you write a piece of code. Say function `f()` calls function `g()`, and something terrible happens in `g()`. `g()` can tell `f()` that something really bad happened in different ways. For instance the return value of `g()` can be an error code. Another way is to get `g()` to set a global variable to some error code; `f()` can check this global variable after execution has returned back to `f()`. In both cases, you control the flow of execution to handle some error.

There's a better way. It involves **exceptions**.

In the case of exceptions, to tell the calling function something has gone wrong, you **throw** an object or value back to the calling function; this object/value is called the **exception**. The calling function can choose to **catch** the exception or not. But the interesting thing is that if `h()` calls `f()` and `f()` calls `g()` and `g()` throws an exception back, if `f()` does not catch it but `h()` does, then the flow of execution goes to the code in h() that catches the exception (or error).

That's the big picture. Now how do you try to catch the error? If `f()` wants to catch an exception, it must wrap the code where an exception might occur in a **try-block**. The try-block is followed by a **catch-block**. If `g()` throws an object or value of some type, say it's an integer, then your catch must specify that it's trying to catch an integer exception:

```
void f()
{
    try
    {
        // code
        g();
        // more code
    }
    catch (int i)
    {
        // code to clean up the mess caused
        // by the error
    }
}
```

What about on `g()`'s side? If something goes wrong `g()` throws an integer back. This is achieved by:

```
void g()
{
    // ... some code ...
    throw 5;
    // ... some code ...
}
```

Your code can catch exceptions of different types, not necessarily an integer. It can even be an object.

```
void g()
{
     std::cout << "g(): about to throw 5...\n";
     throw 5;
     std::cout << "g(): after throw 5 ...\n";

}

void f()
{
     try
     {
           std::cout << "f(): about to call g()...\n";
           g();
           std::cout << "f(): after calling g()...\n";
     }
     catch (int i)
     {
           std::cout << "f(): caught int " << i << '\n';
     }
}

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

Run the following programs/experiments and **trace their execution**. That is in fact the reason for exceptions: they alter the flow of normal program execution.

Program 1. A Simple Example

Run this program. Then uncomment the `throw` statement and run it again.

```
#include <iostream>
#include <exception>
void f()
{
     std::cout << "f ... 1\n";
     throw std::exception();

     //throw std::exception();
     std::cout << "f ... 2\n";
}

int main()
{
    try
    {
        std::cout << "try ... 1\n";
        f();
        std::cout << "try ... 2\n";
    }
    catch (std::exception & e)
    {
        std::cout << "caught exception ... \n";
        std::cout << "end\n";
    }
}
return 0;
```

WARNING: Depending on your compiler, you might need to write `std::exception` instead of `exception`. You might or might not need to `#include` exception.

`exception` is a class that comes with your compiler. In this case I'm throwing a standard `exception` object. You can actually throw any object or any value. If you prefer, you can create your own class for an exception object to throw, or you can just throw any value (`int`, `double`, `bool`, etc.)

This is really handy. Why? Suppose you're writing a program without exceptions and the code looks like the following, i.e., `f()` calls `g()`, `g()` calls `h()`, and `h()` calls `i()`:

```
void i()
{
     ...
}

void h()
{
     ...
     i();
     ...
}

void g()
{
     ...
     h();
     ...
}

void f()
{
     ...
     g();
     ...
}
```

Here's a diagram that describes the function call relationship:

```
f → g → h → i
```

where the → means "calls".

After it's done you analyze your code and realize that some errors might occur in `i()`

```
void i()
{
     ...
     z = x / y; // oops what if y is zero???
     ...
}
```

Suppose you want `f()` to handle this error. Then you might need to pass an error code back to `f()` like this (we return 0 for no error and 1 if there's an error). This means that the error code must pass from

i to h, h must pass the error code to g, and g must pass the error code to f:

```
int i()
{
    ...
    // return 1 if there's an error
    if (y == 0)
       return 1;

    z = x / y; // oops what if y is zero???

    ...

  return 0;  // return 0 if there's no error
}

int h()
{
    ...
    if (i() != 0) return 1;
    ...
    return 0;
}

int g()
{
    ...
    if (h() != 0) return 1;
    ...
  return 0;
}

void f()
{
    ...
    if (g() != 0)
    {
      // ...  do some error
      // handling ...
    }
    ...
}
```

Boy ... what a pain!!! Look at the amount of changes you need to make!!! Not to worry ... exceptions to the rescue ...

```
void i()
{
     ...
     if (y == 0) throw exception();
     z = x / y; // oops what if y is zero???
     ...
}

void h()
{
     ...
     i();
     ...
}

void g()
{
     ...
     h();
     ...
}

void f()
{
     ...
     try
     {
          g();
     }
     catch (exception & e)
```