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();
    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)
    {
       // ... do some error
      //  handling ...
    }
    ...
}
```

Much cleaner!!!

The important thing is this ...

You only need to modify the function code that might run into an error and the function that should handle the error. In other words you only

need to change `i()` and `f()`. We say that `i()` is the thrower and `f()` is the catcher.

Now, just imagine the nightmare if you have a chain of 20 function calls: You would need to modify 20 functions from the thrower all the way back to the catcher.

In a real piece of software, the exception handling code can for instance print a message to the user to let them know that an email was sent to the company and bug fixes will come soon. The program can (for instance) shut down or maybe restart.

Program 3. Rethrowing an exception

```
#include <iostream>
#include <exception>

void f()
{
    std::cout << "f ... 1\n";
    throw exception();
    std::cout << "f ... 2\n";
}

void g()
{
    try
    {
        std::cout << "g ... 1\n";
        f();
        std::cout << "g ... 2\n";
    }
    catch (exception & e)
    {
        std::cout << "g ... caught\n";
        throw;
    }
}


int main()
{
    try
    {
        std::cout << "main ... 1\n";
        g();
        std::cout << "main ... 2\n";
    }
    catch (exception & e)
    {
        std::cout << "main ... caught \n";
    }

    std::cout << "end\n";

    return 0;
}
```

This is helpful when the error handling involves several functions within the function call stack.

Program 4. Writing your own exception

```
#include <iostream>
#include <exception>

class MyException
{
    // dummy class
};


void f()
{
    throw MyException();
}

int main()
{
    try
    {
        f();
    }
    catch (MyException e)
    {
        std::cout << "caught a MyException object\n";
    }

    return 0;
}
```

Here's an example on handling division by zero.

```
#include <iostream>
#include <exception>

class ZeroDivisionError
{};


int f(int x, int y, int z)
{
  if (y == 0) throw ZeroDivisionError();

  return x / y + z;
}

int main()
{
    try
    {
        f(1, 0, 2);
    }
    catch (ZeroDivisionError & e)
    {
        std::cout << "You cannot divide by zero!\n";
    }
    return 0;
}
```

Exercise: Write a `Stack` class to contain integers; suppose the maximum capacity of the stack is 5 (example: you use an array to keep the values). Write a `StackException` class to handle overflow and underflow errors. (Skeleton answer on the next page.)

```
#include <iostream>
#include <cmath>
#include <ctime>

class StackException
{};

class Stack
{
public:
    ...
    void push(int i)
    {
       if (size == 5)
            throw StackException();
       ...
    }
    int pop()
    {
       if (size == 0)
           throw StackException();
       ...
    }

    ...
};


int main()
{
    srand((unsigned) time(NULL));
    Stack stack;

    try
    {
        while (1)
        {
           int option;
           std::cin >> option;
           switch (option)
           {
               case 0: stack.push(rand()); break;
               case 1: stack.pop(); break;
           }
           std::cout << stack << std::endl;
        }
    }
```

```
    catch (StackException & e)
    {
        std::cout << "stack error" << std::endl;
    }
    return 0;
}
```

Program 5. Exception objects with members

Of course you can put anything in your exception object. Here's an example where our own handcrafted exception class, `MyException`, contains an integer error code.

```
#include <iostream>
#include <exception>


class MyException
{
public:
    MyException(int err_code0)
        : err_code(err_code0)
    {}

    int err_code;
};


void f()
{
    throw MyException(5);
}

int main()
{
    try
    {
        f();
    }
    catch (MyException & e)
    {
        std::cout << "caught a MyException object. "
                  << "Error code:" << e.err_code
                  << "\n";
    }
    return 0;
}
```

Program 6. Catching Multiple Exceptions

Of course there's no reason to believe that in a chunk of code (say in a function) there is only one possible error. The following show you that a function, in the example that would be `main()`, can be a catcher to two different types of exceptions.

```
#include <iostream>
#include <exception>


class MyException
{
public:
    MyException(int err_code0) : err_code(err_code0)
{}
    int err_code;
};

class MyException2
{};

void f()
{
    throw MyException(5);       // TRY EACH THROW
    // throw MyException2();
    // throw 42;

}

int main()
{
    try
    {
        f();
    }
    catch (MyException & e)
    {
        std::cout << "caught a MyException object. "
                  << "Error code:" << e.err_code
                  << "\n";
    }
    catch (MyException2 & e)
    {
        std::cout << "caught a "
                  << "MyException2 object.\n";
    }
    catch (int i)
    {
       std::cout << "caught integer " << i << "\n";
    }
    return 0;
}
```

Exercise Recall a previous exercise:

```
#include <iostream>
#include <cmath>
#include <ctime>

class StackException
{};

class Stack
{
public:
    ...
    void push(int i)
    {
       if (size == 5)
         throw StackException();
       ...
    }
    int pop()
    {
       if (size == 0)
         throw StackException();
       ...
    }

    ...
};


int main()
{
    srand((unsigned) time(NULL));
    Stack stack;

    try
    {
        while (1)
        {
            int option;
            std::cin >> option;
            switch (option)
            {
                case 0: stack.push(rand()); break;
                case 1: stack.pop(); break;
            }
            std::cout << stack << std::endl;
        }
     }
```

```
    catch (StackException e)
    {
      std::cout << "stack error" << std::endl;
    }
}
```

Modify the program by replacing StackException with two new classes, `StackOverflowException` and `StackUnderflowException`, to handle the two different errors. The code in `main()` should catch both errors.

By the way it's a good idea to keep exceptions which are thrown only in a class together with the class itself in the same header file. It makes the exception class easier to find.

Program 7. What if the exception is not caught?

First try this and see what happens when an exception is thrown but no one is catching it:

```
#include <iostream>
#include <exception>


void f()
{
    std::cout << "f ... 1\n";
    throw exception();
    std::cout << "f ... 2\n";
}

int main()
{
    std::cout << "try ... 1\n";
    f();
    std::cout << "try ... 2\n";

    return 0;
}
```

Get it?

Memory Allocation Exception

You know that if the system is not able to allocate memory for a pointer,

```
int * p = new int[some_big_number];
```

the program might (1) crash and stop or (2) the pointer value returned is 0. (For the case of g++, when the requested memory cannot be fulfilled, the program crashes.)

In case (2), say you're writing network game and the new operator returns 0 you might want to do this:

```
int * p = new int[some_big_number];
if (p == NULL)
{
    // Ooops ... system ran out of memory ...
    // Print some error message, clean up
    // (release resources like network
    // connections, database connections, etc.),
    // and stop the program
}
else
{
    // continue what you wanted to do
}
```

In case (1) where the system crashes and stop, it actually throws an exception. (For newer C++ compilers (1) is actually the norm.) The reason why the program crashes and stops is because the exception is not caught. The exception object thrown is from the class `std::bad_alloc`. Run this:

```
#include <iostream>
#include <exception>

int main()
{
    try
    {
        int * p = new int[1000000];
        std::cout << "ok" << std::endl;
    }
    catch (std::bad_alloc & e)
    {
        std::cout << "bad!!!\n";
    }
    return 0;
}
```

Your system should be able to provide memory for `p` to point to. Next modify it like so ...

```
...
    try
    {
        while (1)
        {
            int * p = new int[1000000];
            std::cout << "ok" << std::endl;
        }
    }
...
```

... and system is bound to run out of memory for allocation.

So in the case where your C++ compiler throws a `std::bad_alloc` exception when the new operator cannot be satisfied, you can do this:

```
#include <iostream>
#include <exception>

int main()
{
    try
    {
        game(); // run the game
    }
    catch (std::bad_alloc & e)
    {
        // Ooops ... system ran out of memory ...
        // Print some error message,
        // clean up (release resources like
        // network connections, database
        // connections, etc.).
    }

    return 0;
}
```

But ...

What if the compiler you're using is older and does not thrown a `std::bad_alloc` object when there's not enough memory or for some reason you do not want the compiler to throw a `std::bad_alloc` object when the new operator cannot be satisfied?

First of all you can disable throwing. Try this:

```
#include <iostream>
#include <exception>
#include <new>


int main()
{
    try
    {
        while (1)
        {
            int * p = new (std::nothrow) int[1000000];
            std::cout << "ok" << std::endl;
        }
    }
    catch (std::bad_alloc & e)
    {
        std::cout << "bad!!!\n";
    }
    return 0;
}
```

This will cause `p` to be set to `NULL` if there's not enough memory for allocation.

Most modern C++ compilers ***do*** throw an exception when there's not enough memory. But anyway if you have a compiler that does not (or you prefer not to) and the compiler understands `std::nothrow` and you want to throw your own memory exception object, you can do the following:

```
#include <iostream>
#include <exception>
#include <new>

class MemoryAllocationException {};

int main()
{
    try
    {
        while (1)
        {
            int *p = new (std::nothrow) int[1000000];

            if (p == NULL)
                throw MemoryAllocationException();

            std::cout << "ok" << std::endl;
        }
    }
    catch (MemoryAllocationException & e)
    {
        std::cout << "bad!!!\n";
    }

    return 0;
}
```

Other Scenarios

Suppose you have an `LongInt` class for the modeling and computation of integer values that are larger than what the `int` provides. Then one possible place where you want to raise an error (besides memory allocation error) if the division-by-zero error. Therefore your `LongInt` class might look like this:

```
// File: LongInt.h

class DivByZeroException {};

class LongInt
{
...

    LongInt & operator/=(const LongInt & i)
    {
        if (i == 0) throw DivByZeroException();
        ...
        return (*this);
    }

...
};
```

Your main program might then look like this:

```
// File: main.cpp

#include <iostream>
#include <exception>
#include "LongInt.h"

int main()
{
    std::cout << "predict the next-day stock "
              << "price of Google!!!\n"
              << "enter \n"
              << "1. current price\n"
              << "2. your weight\n"
              << "3. today's highest temperature\n"
              << "4. some random number\n";
    try
    {
        LongInt I, J, K, L;
        std::cin >> I >> J >> K >> L;
        std::cout << (I + L * J / K) << std::endl;
    }
    catch (std::bad_alloc & e)
    {
        std::cout << "out of mem!!! buy more"
                  << "ram!!!\n";
    }
    catch (DivByZeroException & e)
    {
        std::cout << "oops ... highest temp "
                  << "cannot be zero\n";
    }
}
```

Another common thing that is done when something severe has occured is to log the error, i.e. write some useful error/debugging message to a file, and possible send the file to the author of the program. You've seen this before when a program unexpectedly crashes and a window pops up, informs you that an error has occured and asks you if you want to send the error to Microsoft (or some other company). That program was probably contains exception handling.

Bad Usage of Exceptions

From previous examples you see that exceptions can be used to alter flow of execution in situations where some type of error has occurs. Of course you can use this to alter the flow of execution in any situation, but that would be a WRONG usage of exceptions. Here's one such bad use of exceptions:

```
#include <iostream>

class A {};
class B {};
class C {};
class D {};
class F {};

int main()
{
    while (1)
    {
        try
        {
            double score;
            std::cin >> score;
            if (score >= 90)
            {
                throw A();
            }
            else if (score >= 80)
            {
                throw B();
            }
            else if (score >= 70)
            {
                throw C();
            }
            else if (score >= 60)
            {
                throw D();
            }
            else
            {
                throw F();
            }
        }
        catch (A e)
        {
            std::cout << "You got A!!!\n";
        }
        catch (B e)
        {
            std::cout << "You got B!!!\n";
```

```
        }
        catch (C e)
        {
            std::cout << "You got C!!!\n";
        }
        catch (D e)
        {
            std::cout << ""You got D!!!\n";
        }
        catch (F e)
        {
            std::cout << ""You got F!!!\n";
        }
    }
    return 0;
}
```

Study the above carefully. Of course you know that the following is better:

```
#include <iostream>

int main()
{
    while (1)
    {
        double score;
        std::cin >> score;
        if (score >= 90)
        {
            std::cout << "You got A!!!\n";
        }
        else if (score >= 80)
        {
            std::cout << "You got B!!!\n";
        }
        else if (score >= 70)
        {
            std::cout << "You got C!!!\n";
        }
        else if (score >= 60)
        {
            std::cout << "You got D!!!\n";
        }
        else
        {
            std::cout << "You got F!!!\n";
        }
    }
    return 0;
}
```

The type of error you want to handle with exceptions are usually those that are pretty severe, in the sense that the program usually has to shutdown completely and possibly it would have to do some form of resource clean up (example: release memory usage, release some exclusive hold of hardware resource, etc.)

Assertions

Compile and run this program. For this example, I'll be using g++.

```
#include <iostream>
#include <cassert>

int q(int x, int y)
{
    assert(y != 0);
    return x / y;
}

int main()
{
    std::cout << q(1, 1) << std::endl;
    std::cout << q(1, 0) << std::endl;

    return 0;
}
```

When I run this I get the following:

```
[student@localhost assert]$ g++ *.cpp
[student@localhost assert]$ ./a.out
1
a.out: main.cpp:6: int q(int, int): Assertion `y != 0' failed.
Aborted (core dumped)
```

Now compile and run it this way:

```
[student@localhost assert]$ g++ *.cpp -DNDEBUG
[student@localhost assert]$ ./a.out
1
Floating point exception (core dumped)
[student@localhost assert]$
```

Assertions are also used to catch errors. You can turn on or off assertion checking. In the second execution above, when you turn off assertion checking, the integer division goes through, causing a division-by-zero exception.

You can also compile with assertions turned off in this manner by writing your code this way:

```
#include <iostream>
#define NDEBUG
#include <cassert>

int q(int x, int y)
{
    assert(y != 0);
    return x / y;
}

int main()
```

```
{
    std::cout << q(1, 1) << std::endl;
    std::cout << q(1, 0) << std::endl;

    return 0;
}
```

and compiling in the usual way:

```
[student@localhost assert]$ g++ *.cpp
[student@localhost assert]$ ./a.out
1
Floating point exception (core dumped)
```

While you can create a try-catch block to catch exceptions, you **cannot catch assertions**. So assertions are meant for catching programming errors. Once caught you will can figure out what's wrong with the code and fix it. In the above example, the function `q()` should never be used with `0` as the second argument.

The boolean condition for an assertion is meant to never ever happen. Here's an example to differentiate between exceptions and assertions.

Suppose you have a program that prompts the user for a filename. It's entirely possible that the user enters the filename wrongly so that opening the file is an error. This can happen and you can't possibly prevent the user from entering wrong data! On the other hand, if you are writing a program that performs some kind of bit computation using an integer for a bit and there's a helper function that accepts this bit (0 or 1), then the input must be 0 or 1. Sending in 42 is an error. So in this case, using an assertion to check that the input is 0 or 1 is the correct type of error checking.

So error checking for an error that can happen should be handled with exceptions while error check for an error that should never happen should be handled with assertions.

Note that the reason why you can easily turn off assertions when you compile is because during software development, you would turn on assertions and when you want to deliver the software product, you would compile with assertions turned off so as not to waste CPU cycles.

By the way, `assert` is not a function. It's a macro. See notes on preprocessor directives.

(There's also a `static_assert()` that checks a boolean condition that can be evaluated during compile time.)