Exception Handling in C++

An exception, in computer science jargon, is an event that is invoked when something "goes wrong" during the execution of a program. Exceptions can be raised by hardware or by software. Dividing by zero[1], running out of memory, and attempting to access protected parts of memory, are examples of events typically caught by hardware and then picked up by the operating system, which then deals with the guilty program, usually summarily and swiftly, sending it to program heaven[2]. Software can also raise exceptions if the language run-time environment supports it. The C++ language provides the means for software to raise exceptions.

Why do you need to know how to handle exceptions? Here is some motivation.

1. A large percentage of the C and C++ standard library functions raise exceptions when some "exceptional condition" (i.e., error) occurs. The default action taken by the system (meaning operating system and runtime environment) is to terminate the program making the offending call. If your program has instead chosen to "handle" the exception, it will not be terminated involuntarily.

2. If you have ever written a robust program that handled all possible "exceptional" circumstances, such as invalid input, devices not being ready, network connections that timed out, and so on, and you did not use exceptions, then you probably resorted to code that either set some global error variables and required that all functions inspect these variables before proceeding, or you doubled the size of the program with code like

```
int  func(…)
{
        /* … stuff here … */
        errcode = do this;
        if  (errcode is an error)
                return to caller passing the error code up;
        continue with normal processing;

        errcode = do the other thing;
        if (errcode is an error)
                return to caller passing the error code up;
        continue with normal processing;
        /* … more stuff …*/
}
```

You probably ended up writing more lines of code to handle errors than code that actually did the work. When an error occurs in a function sitting on top of a deep call stack, the error has to propagate all the way back to the function that can reasonably recover, which is often

---

[1] Although divide-by-zero errors in C++ do not raise exceptions in the math library.
[2] Asynchronous events such as disk interrupts and keyboard interrupts, are not really exceptions, which are by most definitions, synchronous events.

the main program. You might have wished there were a way to just break out of the call stack immediately, sort of like a goto outside of the current scope (which is usually impossible).

3. The C++ exception handling mechanism in its simplest form is relatively easy to learn and use, and doing so will make your programs easier to read, easier to develop and maintain, and smaller. And knowing this feature of the language will make you a better programmer.

If you have reached this far and are not convinced, you can go back to whatever you were doing before you bothered to download these notes. I will assume that anyone who makes it past this point is ready for the nitty-gritty.

Try-Throw-Catch

These are the three operators that make exception handling work. ***throw***, ***try***, and **catch** are keywords in C++. The idea is that you create a try-block, which is a statement block followed by *exception handlers*. An exception handler is introduced by keyword **catch**. The basic form is

> ***try*** {
>> *a statement sequence containing function calls to functions*
>> *that might throw exceptions, or to functions that might(call functions*
>> *that might call functions that might)*$*^3$ *… throw exceptions*
> *}*
> ***catch*** *(exception_declaration_1) {*
>> *statements to handle exceptions of type exception_declaration_1*
> *}*
> ***catch*** *(exception_declaration_2) {*
>> *statements to handle exceptions of type exception_declaration_2*
> *}*
> ***catch*** *(exception_declaration_3) {*
>> *statements to handle exceptions of type exception_declaration_3*
> *}*
> *…*
> ***catch*** *(exception_declaration_N) {*
>> *statements to handle exceptions of type exception_declaration_N*
> *}*

If an exception is thrown by some function that is called directly or indirectly from within the try-block, execution immediately jumps to the first exception handler whose exception declaration matches the raised exception. If there are no handlers that can catch the exception, the try-block behaves like an ordinary block, which means it will probably terminate.

An *exception declaration* is exactly like the formal parameter declaration of a unary function; it is a type-specifier optionally followed by a parameter name, such as

> **catch** (ErrorType1 e)

or

---

[3] I am using the regular expression operator "*" here to mean "0 or more copies of the preceding string".

> **catch** (MyOwnError)

An example follows.

```
struct the_Ball {
        int b;
        the_Ball(int x = 0) : b(x) { }
};

void test ( )
{
        /* … */
        throw the_Ball(6);      // This is a throw-statement. It is throwing an anonymous object
                                // of the_Ball structure, initialized by a call to the constructor
}

int  main ( )
{
        /* … stuff here */
        try {
                test();
        }
        catch (the_Ball c)
        {
                cerr << " test() threw ball " << c.b << "\n";
        }
        …
}
```

A throw statement is, as shown above, of the form

> **throw** expression;

The thrown expression is any expression that could appear on the right hand side of an assignment statement. The most likely candidate is a call to a constructor, such as I showed above. The safest way to handle exceptions is to define unique types for the distinct exceptions. One way to guarantee that the types are unique is to enclose them in a namespace. This is also a good place for an error count variable.

```
namespace Error {
        int count = 0;
        struct the_Towel {  };   // empty struct, default generated constructor
        struct the_Game { };   // empty struct
        struct the_Ball {
                int b;
                the_Ball(int x = 0) : b(x) { }
        };
        void update()  { count++; }
}
```

Having defined the above error types, the program could be

```
void Pitcher ( )
{
        /* … */
        throw Error::the_Ball(6);
}

void Boxer ( )
{
        /* … */
        throw Error::the_Towel();
}

void PoorSport ( )
{
        /* … */
        throw Error::the_Game();
}

int  main ( )
{
        /* … stuff here */
        try {
                Pitcher ( );
                Boxer ( );
                PoorSport ( );
        }
        catch (Error::the_Ball c)
        {
                Error::update();
                cerr <<Error::count << ": threw ball " << c.b << "\n";
        }
        catch (Error::the_Towel )
        {
                Error::update();
                cerr <<Error::count << ": threw in the towel\n";
        }
        catch (Error::the_Game )
        {
                Error::update();
                cerr << Error::count << ": threw the game\n";
        }
        …
}
```

A More Interesting Example (from Stroustrup, pp. 186-187)

```
struct Range_error {
    int i;
    Range_error ( int n) { t = n; }
};

char  to_char( int n )
// if n is an integer in the range of valid character codes, return n as a char,
// otherwise throw a range error.
{
    if ( n < numeric_limits<char>::min() || numeric_limits<char>::max() < n )
       throw Range_error(n);
    return n;
}

void g ( int m)
{
    try {
        char c = to_char(m);
    }
    catch (Range_error x) {
        cerr << "oops: to_char(" << x.t << ")\n";
    }
```

There is a special notation that you can use for the parameter of an exception handler when you want it to catch all exceptions.

   catch *(…)*

means catch every exception. It acts like the default case of a switch statement when placed at the end of the list of handlers.

This is just a superficial overview of the exception handling mechanism. To give you a feeling for why there is much more to learn, consider some of the following complexities.

- Since any type is a valid exception type, suppose it is publicly derived class of a base type. Do the handlers catch it if they catch the base type? What if it is the other way around?

- Can a handler throw an exception? If so, can it throw an exception of the same type as it handles? If so, what happens? Does it call itself? (Fortunately, although the answer is "yes" to the first question, it is "no" to the third.)

- What if the exception is thrown as a reference, but the handler declares it as a const parameter?

- Are there standard exceptions, and if so, what are they? Which library functions can raise them? The answer to this is that there is a class hierarchy of exceptions, with, not surprisingly, a root named **exception**. All exceptions derive from **exception**. There are **logic_error** exceptions, **bad_cast**, **bad_alloc** (thrown by *new*( )), **underflow_error**, and **out_of_range** (thrown by *at( )*) exceptions, to name a few.

Lastly, you may see function declarations that look like

  void f (int a) **throw** *(err1, err2);*

which specifies that *f( )* is only allowed to throw exceptions of types *err1* and *err2*, and any exceptions that may be derived from these types. Why would you want to do this? Because it is a way of telling the reader of your interface that the function promises to throw only those exceptions, and so the caller does not have to handle any exceptions except those. So

  void f(int a) **throw** *( );*

means that *f( )* throws no exceptions. If *f( )* does throw a different exception, the system will turn it into an *std::unexpected()* call, which results in a terminate call.

The Standard Exception Hierarchy

The set of standard exceptions forms a hierarchy in C++. The language defines a base class named *exception* from which other exceptions are derived. Some of these are defined in the *<stdexcept>* header file; others are defined in *<typeinfo>*, *<new>*, *<ios>*, and *<exception>*. The standard exceptions are listed below. The indentation indicates sub-classing. To the right of the exception name is the header file that contains its definition.

exception	*<exception>*
bad_alloc	*<new>*
bad_cast	*<typeinfo>*
bad_exception	*<exception>*
bad_typeid	*<typeinfo>*
ios_base::failure	*<ios>*
logic_error	*<stdexcept>*
length_error	*<stdexcept>*
domain_error	*<stdexcept>*
out_of_range	*<stdexcept>*
invalid_argument	*<stdexcept>*
runtime_error	*<stdexcept>*
overflow_error	*<stdexcept>*
range_error	*<stdexcept>*

| *underflow_error* | `<stdexcept>` |