Handling Errors

Only extremely lucky programmers never have to deal with errors. Almost all of us have to deal with users who do not enter data in the formats we expect, programming errors that corrupt data, computers that do not meet our hardware or software requirements, hardware failures and other exceptional situations. A successful programmer is one who anticipates problems in advance and, whenever possible, writes code that works around those problems.

This article discusses some of the kinds of situations that might occur during the execution of your programs and what strategies can be used to handle them. As game programmers, we generally deal with the inconveniences of a program crash. In the game software business these are annoying and may lead to poor reviews, a bad reputation, and a loss of sales. In a broader sense, not dealing with errors properly has led to major catastrophes such as rockets exploding (like the Arianne V) and people dying (the Therac-25 radiation therapy machine).

Thus, knowing how to deal with unexpected situations is very important in any situation.

Return Values

The simplest method of detecting the unexpected is to return a code indicating success or failure. Suppose we have a function in our game program that checks to see if DirectX is installed and configured. The prototype for such a function might be something like this:

```
bool checkVideoConfiguration();
```

The usage of this function would look something like this:

```
int main()
{
    if(checkVideoConfiguration() == false)
    {
        //Inform the user that video is not
        //properly set up.
        return 1;
    }
    // game code…
    return 0;
}
```

As our program grows, we might have functions that check the sound, network and setup data. We might want to pull all of this into a single function that initializes the computer. This function then calls other functions that check each subsystem:

```
bool init();
bool checkSoundConfiguration();
bool checkVideoConfiguration();
bool checkNetworkConfiguration();
bool checkSetupData();
```

The usage of this function would look something like this:

```
int main()
{
    if(init() == false)
    {
        //Inform the user that video is not
        //properly set up.
        return 1;
    }
    // game code…
    return 0;
}
```

The `init()` function looks like this:

```
bool init()
{
    if(checkSoundConfiguration() == false)
    {
        return false;
    }

    if(checkVideoConfiguration() == false)
    {
        return false;
    }

    if(checkNetworkConfiguration() == false)
    {
        return false;
    }

    if(checkSetupData() == false)
    {
        return false;
    }
    return true;
}
```

There are two major problems with this approach. Firstly, when `init()` returns false, there is no indication of which function inside `init()` caused the return. When an error occurs, we want to find the reason. If we are developing and testing the program this information is necessary to pinpoint any possible bug and if the false return is triggered when the user is starting, we need to give the user a clear message as to what the problem is. Simply stating something like, "Initialization encountered an error", is more than unhelpful. Messages like this to a gamer that has paid good money to play your game will almost certainly lead to a return of the game and a bad reputation when your former customer posts something online about how lame your software is.

The second problem is that code like this is very fragile. It is not unusual to see a function that calls another function, which calls a third function and so on to an arbitrarily deep level. This code demonstrates the problem:

```
bool init();
bool checkConfiguration1();
bool checkConfiguration2();
bool checkConfiguration3();
bool checkConfiguration4();
bool checkConfiguration5();
bool checkConfiguration6();
```

The usage of this function would look something like this:

```
int main()
{
    if(init() == false)
    {
        //Inform the user that video is not
        //properly set up.
        return 1;
    }
    // game code…
    return 0;
}

bool init()
{
    if(checkConfiguration1() == false)
    {
        return false;
    }
    //Other checks…
}

bool checkConfiguration1()
{
    if(checkConfiguration2() == false)
    {
        return false;
    }
    //Other checks…
}
```

```
bool checkConfiguration2()
{
    if(checkConfiguration3() == false)
    {
        return false;
    }
    //Other checks…
}

bool checkConfiguration3()
{
    if(checkConfiguration4() == false)
    {
        return false;
    }
    //Other checks…
}

bool checkConfiguration4()
{
    if(checkConfiguration5() == false)
    {
        return false;
    }
    //Other checks…
}

bool checkConfiguration5()
{
    if(checkConfiguration6() == false)
    {
        return false;
    }
    //Other checks…
}

bool checkConfiguration6()
{
    //This is the code that actually does the checking…
}
```

A change to any of these functions will likely lead to changes in this entire structure. In software engineering, this is called **tight coupling**. Software modules (in this case, the functions) rely on the specifics of other modules. We want **loose coupling**, where we don't have these kinds of dependencies. It leads to much more robust software that easier to use, maintain and extend.

Return codes are useful in certain situations. Let's suppose we are maintaining a library that uses C file input and the code looks like this:

```
#include  <stdio.h>

//In the middle of the code, we open a
//file stream and read each character.

FILE* stream;
stream = fopen(…);  //where fopen opens some file
if(stream != NULL)
{
    int c;
    while((c = fgetc(stream)) != EOF)
    {
        char ch = (char)c;
        //Do something with the character ch.
    }
}
```

The C function `fgetc(…)` will return the defined value, `EOF`, when the end of the file is reached. Note that this is not an error condition. The end of file is a natural attribute of files and the code is simply checking for that attribute. Also note that there is no long chain of functions calling other functions and a return code is not bubbled up through all these calls.

In these situations, a return value is useful provided that the return code does not represent a software or hardware error and the return value is dealt with by the calling code.

Global Error Codes

In some languages, one notable example being C, there is a global variable defined by the system. In error situations this variable is assigned a value representing various errors. Calling code can check the value of this variable. In C, this variable is named `errno` and is defined in `error.h`. To send the user a message about the error, we could use the C function `perror(…)` which interprets `errno` and sends the message on `stderr`. Provided a function writes to this variable when an error occurs, the use of this would be:

```
#include <error.h>
#include <stdio.h>

if(someFunction() == false)
{
    //errno contains a code regarding the error.
    perror("Ooops!")
}
```

We could implement a similar scheme for our initialization code we saw earlier:

```
int configError = 0;
const int SOUND_CONFIG_ERROR = 1;
const int VIDEO_CONFIG_ERROR = 2;
const int NETWORK_CONFIG_ERROR = 3;
const int DATA_CONFIG_ERROR = 4;

bool checkSoundConfiguration()
{
    if(sound config fails)
    {
        configError = SOUND_CONFIG_ERROR;
        return false;
    }
    return true;
}

bool checkVideoConfiguration ()
{
    if(video config fails)
    {
        configError = VIDEO_CONFIG_ERROR;
        return false;
    }
    return true;
}

bool checkNetworkConfiguration ()
{
    if(network config fails)
    {
        configError = NETWORK_CONFIG_ERROR;
        return false;
    }
    return true;
}

bool checkDataConfiguration()
{
    if(data config fails)
    {
        configError = DATA_CONFIG_ERROR;
        return false;
    }
    return true;
}
```

```
bool init()
{
    if(checkSoundConfiguration() == false)
    {
        return false;
    }

    if(checkVideoConfiguration() == false)
    {
        return false;
    }

    if(checkNetworkConfiguration() == false)
    {
        return false;
    }

    if(checkSetupData() == false)
    {
        return false;
    }
    return true;
}
```

Now we can check the value of the error code

```
int main()
{
    if(init() == false)
    {
        switch(configError)
        {
             case SOUND_CONFIG_ERROR:
                 // Tell user there is a
                 // problem with the sound configuration
                 break;
             case VIDEO_CONFIG_ERROR:
                 // Tell user there is a
                 // problem with the video configuration
                 break;
             case NETWORK_CONFIG_ERROR:
                 // Tell user there is a
                 // problem with the network configuration
                 break;
             case DATA_CONFIG_ERROR:
                 // Tell user there is a
                 // problem with the data configuration
                 break;
        }
        return 1;
    }
    // game code…
    return 0;
}
```

While this error mechanism has been successfully used in a lot of good C code (as well as some very poor C), there are several problems associated with it. First of all, successful operations do not necessarily reset the global variable, so there is no guarantee that the value isn't stale and first set by a function that has not been called in the operation you are evaluating. As a user of this variable, you have to ensure that all code you are calling is using this mechanism properly. Furthermore, this code does not really reduce the tightness of the coupling; there still is a good deal of dependency between the modules.

Perhaps the largest issues with this mechanism are the implications of multithreading, which are necessary parts of any new game.   It is possible that an error could be faithfully recorded in the variable, but before that value is read and dealt with, another thread over-writes the variable. Global error codes predate the common use of multithreading in game software and the two do not work well together.

Return Error Codes

This mechanism is widely used in Windows programming. All of this section is valid in Windows programming and does not apply to other platforms.

Many Window API functions return a value of type `HRESULT`, which is a wrapper for an unsigned 32 bit integer. Not only is this value returned, it is written into a global variable that can be read through the API call `GetLastError()`. The value of 0 (`ERROR_SUCCESS`) represents successful execution of the function and non-zero values represent specific errors. This scheme includes the ability to define custom errors, error messages and functions to retrieve a readable description of the error.

Code examples of this are not included here because the handling errors in the Windows API is beyond the scope of this document and is a subject worth several hundred pages on its own. If you are curious, start by looking up "Error Handling" on the MSDN web Site.

Object States

In C++, errors can be detected by examining the state of an object. The most common application of this is input streams. Recall that in the language, the concepts of input and output have been abstracted to the notion of a stream. Think of a stream as a 'data pipe' that reads or writes data from the keyboard, the screen, a file, the network, other processes or an object we create to change data from one data type or class to another. A common application of this principle is `std::cin` when reading data from the keyboard. Consider this code which gets a list if arbitrary length of positive integers from the user (listing 00):

```
int main()
{
    vector<int> numbers;
    //prompt the user
    cout << "Enter positive numbers (-1 to quit): ";
    int input;
    cin >> input;
    while(input != -1)
    {
        if(input >= 0)
        {
            numbers.push_back(input);
        }
        cout << "Enter positive numbers (-1 to quit): ";
        cin >> input;
    }
    // more code…
    return 0;
}
```

This is very optimistic code. It assumes that user is not going to attempt to enter text that is not numeric. We could always hope for users who follow instructions carefully, but almost all of us are not that fortunate. All will be well, if the user enters something like:

```
34 54 66 34 10 -1
```

But things will not go according to plan if the user tries:

```
67 44 534 22 Hello -1
```

This will cause the program to go into an infinite loop. The `cin` object tries to interpret "Hello" as an integer, fails and enters an error state. The text is not removed from the stream, the value of `input` is not set to -1, and the loop is not exited.

There are four states that the `cin` object could take in its attempt to parse input into a variable:

1. **`good`**: The operation was successful.
2. **`eof`**: The operation encountered the end of file. In the case of the keyboard, this happens when the user enters Ctrl-D (Unix, Macintosh and Linux) or Ctrl-Z (Windows).
3. **`fail`**: Something unexpected happened.
4. **`bad`**: Something unexpected and bad happened. Usually this signals a serious hardware, software or configuration error. Hopefully, this is very rare.

We can check to see if there is an error state on the input stream if we apply a conditional that checks for it (listing 01):

```
int main()
{
    vector<int> numbers;
    //prompt the user
    cout << "Enter positive numbers (-1 to quit): ";
    int input;
    cin >> input;

    while(input != -1)
    {
        if(cin.eof() == true)
        {
            cout << "End of file found" << endl;
            break;
        }
        if(cin.fail() == true)
        {
            cout << "Error in input." << endl;
            break;
        }
        if(cin.bad() == true)
        {
            cout << "Serious error!" << endl;
            return 0;
        }


        if(input >= 0)
        {
            numbers.push_back(input);
        }
        cout << "Enter positive numbers (-1 to quit): ";
        cin >> input;
    }
    return 0;
}
```

If we run a number of times this we get:

```
C:\listing_01>cin_test
Enter positive numbers (-1 to quit): 342
Enter positive numbers (-1 to quit): 34
Enter positive numbers (-1 to quit): 232
Enter positive numbers (-1 to quit): ^Z
End of file found

C:\listing_01>cin_test
Enter positive numbers (-1 to quit): 43
Enter positive numbers (-1 to quit): 54
Enter positive numbers (-1 to quit): 66
Enter positive numbers (-1 to quit): 34
Enter positive numbers (-1 to quit): 2
Enter positive numbers (-1 to quit): Hi
Error in input.
```

Ideally, what we want to do is check to see if `cin` has successfully parsed the text. If not, we clear the contents and reset its state to good. `cin`'s class (`std::istream`) has the negation operator defined such that we can easily check whether its state is good or not. Unfortunately, what `clear()` does not do is to remove the text it tries to parse. The result is another infinite loop (listing 02):

```
int main()
{
    vector<int> numbers;
    //prompt the user
    cout << "Enter positive numbers (-1 to quit): ";
    int input;
    cin >> input;

    while(input != -1)
    {
        if(!cin)
        {
            cin.clear();
        }
        else
        {
            if(input >= 0)
            {
                numbers.push_back(input);
            }
        }
        cout << "Enter positive numbers (-1 to quit): ";
        cin >> input;
    }
    // more code...
    return 0;
}
```

What we can do is create an object that behaves much like `cin` does. The C++ class `std::stringstream` is a class that behaves like a stream but can take a `std::string` for input or output. We also need to massage the code a bit for this to work (listing 03):

```
int main()
{
    vector<int> numbers;
    int input = 0;

    while(input != -1)
    {
        //prompt the user
        cout << "Enter positive numbers (-1 to quit): ";
        string str;
        cin >> str; //get the raw text that was entered
        stringstream ss(str); //create a stringstream
                              //object based on that sring
        ss >> input;          // parse the object
        if(!ss) //report an error
        {
            if(cin.eof())
            {
                cout << "End of file found." << endl;
                cin.clear();
            }
            else
            {
                cout << "Error parsing " << ss.str() << endl;
            }
        }
        else
        {
            if(input >= 0)
            {
                numbers.push_back(input);
            }
        }
    }

    // more code...
    return 0;
}
```

For each iteration of the loop, we create a `stringstream` object and populate it with the text that was entered from the user. We then try to interpret each entry in the `stringstream` object as an integer. If the user enters text that cannot be interpreted as an integer, the attempt to convert it to an `int` sets the state of the `stringstream` object to a fail state, but the `cin` object's state remains good. The only error case that might be encountered by the input stream is the end of file. In this case, we need to reset the state by calling `cin.clear()`.

Assertions

Assertions are checks that are used in C and C++ programs to confirm that some major assumption about the code that a programmer is making is true. The assumption is of some program state that, if false, would be fatal to the process execution.

This is a construct that should never make it into the version of a program that is released to users. If users are experiencing assertion failures, it is a signal that not enough effort has been used in the development and testing processes. Assertions are usually not included in release builds, so what would trip an assertion in testing usually ends up being a general protection fault in Windows or a segmentation fault in Macintosh or Unix.

To use an assertion, the header file `cassert` must be included in the file that uses it:

```
#include <cassert>

GameData* pData = 0;

int main()
{
    pData = getGameData();
    if(pData == 0)
    {
        //Some sort of error.  Quit the program
        return 1;
    }
    //some more code

    //Now we want to access the data object -- check it first
    assert(pData != 0);
    //we know that pData is not null
    //…
}
```

If the value of pData is null when it hits the assertion, the program will immediately exit with some sort of error message such as the file and line number where the failing assertion is located. The assumption in this example is that the value of the pointer needs to be valid and that the data it contains is required for further execution of the game process. When this assertion fails, it indicates a serious error has occurred and the programmer’s next task is to look for the code that set the value of the data pointer to null.

Assertions are useful, but the scope of their use is limited. The key points in using them is that they test for unrecoverable errors and that they should never trigger except relatively early in the development process. They are mechanisms that are useful only for programmers and testers.

Exceptions

Very often a program will encounter an error or an unusual situation that can possibly be handled in the software. This is not a normal situation like finding an end-of-file character in a file stream or a fatal state like a null pointer that should point to some valid object. To handle these situations we return from the function in which the error occurs, not by returning, but by **throwing** an **exception**. The calling code then can **catch** the exception and execute code based on the kind of exception it receives. Proper use of exceptions has been shown to improve software quality, speed up the development time and reduce development costs.

Here is a simple example of exceptions (listing 04):

```
#include <exception>

void functionThatCouldThrowAnException();

int main()
{
    try
    {
        functionThatCouldThrowAnException();
    }
    catch(std::exception& e)
    {
        //deal with the exception
    }
}

void functionThatCouldThrowAnException()
{
    if(/*some error condition*/)
    {
        throw std::exception();
    }
}
```

While this code looks a bit more complex than the examples demonstrating error codes and return statements, this does not show the power of this construct. Of the many advantages of exceptions, one of the biggest is that the calling code does not need to be at the next level (listing 05).

```
void init();
void checkConfiguration1();
void checkConfiguration2();
void checkConfiguration3();
void checkConfiguration4();
void checkConfiguration5();
void checkConfiguration6();
```

The usage of exceptions would look something like this:

```
int main()
```

```
{
    try
    {
         init();
    }
    catch(std::exception& e)
    {
    }
    // game code…
    return 0;
}

void init()
{
    checkConfiguration1();
    //Other checks…
}

void checkConfiguration1()
{
    checkConfiguration2();
    //Other checks…
}
void checkConfiguration2()
{
    checkConfiguration3();
    //Other checks…
}

void checkConfiguration3()
{
    checkConfiguration4();
    //Other checks…
}

void checkConfiguration4()
{
    checkConfiguration5();
    //Other checks…
}

void checkConfiguration5()
{
    checkConfiguration6();
    //Other checks…
}

void checkConfiguration6()
{
    //This is the code that actually does the checking…
    if(/*some configuration error*/)
    {
        throw exception();
    }
}
```

When the exception is thrown in the function `checkConfiguration6()`, it propagates all the way up to the `main()` function. The exception does not have to be coded in the other functions. This alone saves a great deal of time because the code that throws the exception can be arbitrarily deep inside the code that catches it.

Programmers often speak of a program's 'happy path' and 'bad path'. The happy path is the line of execution if everything succeeds and goes as expected or hoped. The bad path is the code that handles the cases when unexpected things happen or errors occur. By using error codes or return values, the code that handles both paths are mixed together. By using exceptions, we can separate the happy path from the bad path and this makes our code easier to write, read and maintain. Notice in this code that the bad path only occurs in the functions that throw and catch the exception. The code in the function in between is cleaner and easier to understand because it is concentrating on the happy path instead of handling an error in a deeper function.

In standard C++, any data type or class can be thrown in an exception. A simple approach is to define error codes and throw a code that represents a certain type of error (listing 06):

```
int const SOUND_CONFIG_ERROR = 0;
int const VIDEO_CONFIG_ERROR = 1;
int const NETWORK_CONFIG_ERROR = 2;
int const DATA_CONFIG_ERROR = 3;
int const LOGON_ERROR = 4;

void checkSoundConfiguration();
void checkVideoConfiguration();
void checkNetworkConfiguration();
void checkDataConfiguration();
void logon();

int main()
{
    try
    {
        checkSoundConfiguration();
        checkVideoConfiguration();
        checkNetworkConfiguration();
        checkDataConfiguration();
        logon();
    }
    catch(int errorCode)
    {
        switch(errorCode)
        {
            case SOUND_CONFIG_ERROR:
                cout << "Sound config error." << endl;
                break;
            case VIDEO_CONFIG_ERROR:
                cout << "Video config error." << endl;
                break;
            case NETWORK_CONFIG_ERROR:
                cout << "Network config error." << endl;
                break;
            case DATA_CONFIG_ERROR:
                cout << "Data config error." << endl;
                break;
            case LOGON_ERROR:
                cout << "Logon error." << endl;
                break;
        }
    }
    return 0;
}


void checkSoundConfiguration()
{
    if(/*Sound check*/)
    {
        throw SOUND_CONFIG_ERROR;
    }
}

void checkVideoConfiguration()
{
    if(/*Video check*/)
    {
        throw VIDEO_CONFIG_ERROR;
    }
}

void checkNetworkConfiguration()
{
    if(/*Network check*/)
    {
        throw NETWORK_CONFIG_ERROR;
    }
}

void checkDataConfiguration()
{
    if(/*Data check*/)
    {
        throw DATA_CONFIG_ERROR;
    }
}

void logon()
{
    if(/*Logon rejected*/ true)
    {
        throw LOGON_ERROR;
    }
}
```

The execution of the functions inside the try block proceeds normally until one of the functions or the code that it calls throws an exception. The execution of the path then jumps out of all functions until it finds a catch block that matches the type of what has been thrown – in this case, an int. Any functions in the `try` block after the call that throws the exception are not executed. For example,

```
    try
    {
        checkSoundConfiguration();
        checkVideoConfiguration();
        checkNetworkConfiguration();
        checkDataConfiguration();
        logon();
    }
    catch(int errorCode)
    {
    }
```

…if an exception is thrown in `checkSoundConfiguration(),` the functions `checkVideoConfiguration(), checkNetworkConfiguration(), checkDataConfiguration()` and `logon()` are not called. The program jumps to the exception block.

Distinguishing exceptions by error code is useful, but there are other approaches. Using an error code and handling the exception in a switch statement is not considered good object oriented design. Good mechanisms allow us to catch different kinds of exceptions based on what is thrown.

Suppose we want to send a message from the logon function informing the user if a logon failure was due to a bad password or user name (yes, I realize that this isn't good security policy, but this is just an example). The code looks similar to the previous example (listing 07):

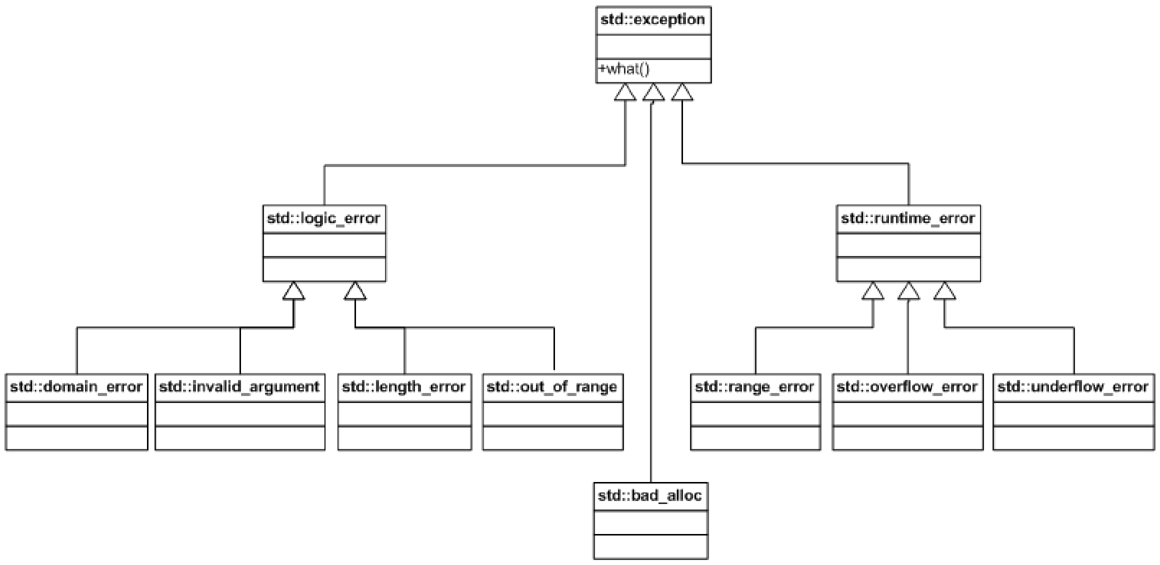
```
int main()
{
    try
    {
        checkSoundConfiguration();
        checkVideoConfiguration();
        checkNetworkConfiguration();
        checkDataConfiguration();
        logon();
    }
    catch(int errorCode)
    {
        switch(errorCode)
        {
            case SOUND_CONFIG_ERROR:
                cout << "Sound config error." << endl;
                break;
            case VIDEO_CONFIG_ERROR:
                cout << "Video config error." << endl;
                break;
            case NETWORK_CONFIG_ERROR:
                cout << "Network config error." << endl;
                break;
            case DATA_CONFIG_ERROR:
                cout << "Data config error." << endl;
                break;
        }
    }
    catch(const char* str)
    {
        cout << str << endl;

    }
    //game code…
    return 0;
}

//Other functions are the same…

void logon()
{
    if(/*bad user name*/)
    {
        throw "Logon Error -- bad user name.";
    }
    if(/*bad password*/)
    {
        throw "Logon Error -- bad password.";
    }
}
```

In case of a logon error a C string is thrown, indicating the kind of error that occurred. When `logon()` throws an exception, the first catch block is not executed because it is written to handle an int. We need a catch block that can handle `const char*` data, just as the second one does. It is this block that handles exceptions from `logon()` while the other block covers the exceptions that throw an `int`.

C++ Exception Classes

This is still not an object oriented approach. Throwing objects is the preferred way of handling exceptions in C++. By using a class structure and virtual functions, you can eliminate many of the switch statements that object oriented programming tries to avoid.

Standard C++ comes with a hierarchy of exception classes that are used by the system; they can also be used in your code.



- `std::exception`: This is the base class for standard exceptions. Because all objects thrown by components of the standard library are derived from this class, all standard exceptions can be caught by catching this type. It contains a virtual function `what()` that returns a C string describing the error that caused the exception.

  - `std::logic_error`: Objects of this report errors in logic in the program. The derived classes of this are:

    - `std::domain_error`: This indicates that the error is a mathematical domain. For example, trying to find the square root of a negative number will throw this kind of exception object.

    - `std::invalid_argument`: An object of this type is thrown by the `std::bitset` class (very rare).

    - `std::length_error`: This exception occurs when `std::string` and `std::vector` attempt to resize to an excessive length.

    - `std::out_of_range`: This exception is thrown when a program tries to access an element of a `std::string`, `std::vector`, `std::deque`, or `std::bitset` that is outside of the range of the container object.

  - `std::runtime_error`: These are errors that can only be determined when the process is running. The derived classes of this are:

    - `std::range_error`: This is a statistical error used in advanced numeric computation (very rare).

    - `std::overflow_error`: Objects of this class are thrown when arithmetic overflow is detected.

    - `std::underflow_error`: Objects of this class are thrown when arithmetic underflow is detected.

- `std::bad_alloc`: This kind of exception is thrown when a call to `new` fails. This is an important class, because instead of manually checking for each allocation of heap memory (as a C programmer does), a C++ programmer needs only to deal with this in a catch block. Fortunately, modern computers and operating systems give game programs gigabytes of memory, so the failure of a call to `new` is very rare.

Using this structure, you can derive your own classes from anywhere in this structure and report on the error with one catch block:

```
int main()
{
    try
    {
        //A bunch of code that could throw exceptions.
    }
    catch(std::exception& e) //Note that this is a reference
    {
        std::cerr << e.what(); //what() is virtual
    }
}
```

You can use any of these classes directly or you can derive your own. Unlike Java or C#, you can even design your own class hierarchy that is not derived from `std::exception`. C++ offers a great deal of flexibility in working with exceptions.

Designing an effective class structure is not a trivial task. In fact, it is often considered just as crucial to a large project as designing the classes on the 'happy path'. Experience is your best guide, but here are several guidelines:

1. Tend to derive your classes from `std::exception` (or classes derived from it). This will allow other programmers to catch it with a reference to the base class.

2. Keep your classes relatively light-weight. Don't throw unreasonably large objects.

3. To keep the exception as light as possible, use references to objects if a catch block is handling a class.

4. To paraphrase Einstein, keep your hierarchy as simple as possible but no simpler. Unnecessary complication in a structure is exactly that: unnecessary.

To handle a wide range of exceptions, structure your catch blocks like this:

```
int main()
{
    try
    {
        //A bunch of code that could throw exceptions.
    }

    catch(int i)
    {
        //decipher what the int value means.
    }

    catch(const char* str)
    {
        //report the contents of the C string.
    }

    catch(MyCustomException1& e) //derived from std::exception
    {
        //deal with the custom exception
    }

    catch(std::bad_alloc& e) //reference
    {
        //deal with the memory allocation error
    }

    catch(std::exception& e) //reference
    {
        //deal with other types of objects derived from
        //std::exception.
        //Objects of type MyCustomException1 and
        // std::bad_alloc were dealt with in their
        //respective catch blocks
    }

    catch(MyCustomException2& e) //Not derived from std::exception
    {
        //deal with the custom exception
    }

    catch(…)
    {
        //This must come last and handles any exception
        // not yet caught.
    }
}
```

An exception will propagate down the list of catch blocks until it finds one of its type. The code inside that catch block will be executed and the execution will jump to beyond the last catch block. If an exception is not handled in any of the catch blocks, it will be caught by a general catch (`catch(…)`) if one is included. The general catch has to be the last catch block if it is included.

Exceptions are very powerful tools that, just like anything else, have their appropriate uses but can be over used and abused if taken to an extreme. The downside of using them is that code inside a try block executes *slightly* slower than code that is not. As a result, it is not an accepted practice to enclose all the code of a program that does a lot of calculation in a try block. Games generally have large blocks of code that calculate physics and graphics. These activities hare very CPU intensive and so they are generally excluded from try blocks. The best guide to learning when to use exceptions comes from experience and mentoring from experienced programmers when working on game code.

Exceptions and Windows

Before the first C++ specification in 1998, Microsoft developed an exception structure for the Win32 API. This is known as Structured Exception Handling (SEH) and it is used widely in many C and C++ projects for Windows. SEH is independent from exceptions in the C++ standard. When planning a project for Win32 or Win64, you must decide whether you want to use them. SEH deals with problems associated with Windows. There is much documentation about using it on the MSDN Web Site so it is not discussed here, but it is an acceptable alternative or adjunct to standard C++. SHE works a bit differently than standard C++ exceptions, so be sure to read the documentation carefully.

This option is set in the Wizards that set up various kinds of projects, but when compiling a Windows program from the command line or Makefile you need to specify which you wish to use:

```
C:>cl /EHsc test.cpp
```

The '`EHsc`' switch tells the compiler that the project is using standard C++ exceptions (same as `/GX`).

```
C:>cl /EHs test.cpp
```

The '`EHs`' switch tells the compiler that the project is using standard C++ exceptions, but to disable SEH.

```
C:>cl /EHa test.cpp
```

The '`EHa`' switch tells the compiler that the project is using standard C++ exceptions and SEH.

Not including one of these options leads to a compiler warning.

Summary

When to use various error handling mechanisms:

Return Values: Use return values when a function can return a value that is a normal part of process execution. An example of this is reading a file in C and watching for the end of file marker.

Global Error Codes: Use this when the system you are programming on has this mechanism in place, like C or the Win32 or Win64 API. It is an older technique that is not thread-safe.

Return Error Codes: Use this when the system you are programming on has this mechanism in place, like the Win32 or Win64 API. This might be thread-safe.

Object States: Use this as an object oriented replacement for return values. The most common application of this technique is the standard I/O classes in C++.

Assertions: Use assertions early in the development process to catch unrecoverable errors that contradict the basic assumptions about the program structure. Assertions should never be used in production code and all errors found by them should be fixed before user testing begins.

Exceptions: Exceptions should used to handle exceptional or unusual situations that can possibly be handled seamlessly (with little or no inconvenience to the user) during process execution.