26 Exceptions

With class based programs, you spend a lot of time developing classes that are intended to be used in many different programs. Your classes must be reliable. After all, no one else is going to use your classes if they crash the program as soon as they run in to data that incorrect, or attempt some input or output transfer that fails.

You should aim to build defences into the code of your classes so that bad data won't cause disasters. So, how might you get given bad data and what kind of defence can you build against such an eventuality?

"Bad data" most frequently take the form of inappropriate arguments passed in a call to a member function that is to be executed by an instance of one of your classes, or, if your objects interact directly with the user, will take the form of inappropriate data directly input by the user.

Simple checks on data

You can do something about these relatively simple problems. Arguments passed to a function can be checked using an `assert()` macro. Thus, you can check that the pointers supposed to reference data structures are not `NULL`, and you can refuse to do things like sort an array with -1234 data elements (e.g. `assert(ptr1 != NULL); assert((n>1) && (n<SHRT_MAX))`). Data input can be handled through loops that only terminate if the values entered are appropriate; if a value is out of range, the loop is again cycled with the user prompted to re-enter the data.

Such basic checks help. Careless errors in the code that invokes your functions are soon caught by `assert()` macros. Loops that validate data are appropriate for interactive input (though they can cause problems if, later, the input is changed to come from a file).

But what happens when things go really wrong? For example, you hit "end of file" when trying to read input, or you are given a valid pointer but the referenced data structure doesn't contain the right information, or you are given a filename but you find that the file does not exist.

Code like:

```
void X::GetParams(const char fname[])
{
        ifstream in(fname, ios::in | ios::nocreate);
        assert(in.good());
```

or

```
    ifstream in(fname, ios::in | ios::nocreate);
    if(!in.good()) {
        cout << "Sorry, that file can't be found.  Quitting."
                      << endl;
        exit(1);
        }
```

catches the error and terminates via the "assertion failed" mechanism or the explicit `exit()` call.

Terminating the program – maybe the only thing to do?

In general, terminating is about all you can do if you have to handle an error at the point that it is detected.

You don't know the context of the call to `X::GetParams()`. It might be an interactive program run directly from the "console" window; in that case you could try to handle the problem by asking for another filename. But you can't assume that you can communicate with the user. Function `X::GetParams()` might be being called from some non-interactive system that has read all its data from an input file and is meant to be left running to generate output in a file that gets examined later.

But other parts of code might have been able to deal with error!

You don't know the context, but the person writing the calling code does. The caller may know that the filename will have just been entered interactively and so prompting for a new name is appropriate when file with a given name cannot be found. Alternatively, the caller may know that a "params" file should be used if it can be opened and read successfully; if not, a set of "default parameters" should be used in subsequent calculations.

Error detection and error handling separated by long call chain

It is a general problem. One part of the code can detect an error. Knowledge of what to do about the error can only exist in some separate part of the code. The only connection between the two parts of the code is the call chain; the code that can detect an error will have been called, directly or indirectly, from the code that knows what to do about errors.

The call chain between the two parts of the code may be quite lengthy:

```
    caller --- (knows what to do if error occurs)
          functionA
                procedure B
                      recursive function X
                            recursive function X (again)
                                  recursive function X (again again)
                                        procedure Y
                                              function Z
                                              Z can detect error
```

While that might be an extreme case, there is often a fairly lengthy chain of calls separating originator and final routine. (For example, when you use menu File/Save in a Mac or PC application, the call chain will be something like `Application::HandleEvent()`, `View::HandleMenu()`, `Document::HandleMenu()`, `Document::DoSaveMenu()`, `Document::DoSave()`, `Document::OpenFile()`; error detection would be in `OpenFile()`, error handling might be part way up the chain at `DoSaveMenu()`.)

Pass an error report back up the call chain

If you don't want the final function in the call chain, e.g. function Z, to just terminate the program, you have to have a scheme whereby this function arranges for a description of the error to be passed back up the call chain to the point where the error can be handled.

The error report has to be considered by each of the calling functions as the call chain is unwound. The error handling code might be at any point in the chain. Intermediate functions, those between the error handler and the error detector, might not be able to resolve the error condition but they might still need to do some work if an error occurs.

Need to tidy up as call chain unwinds.

To see why intermediate functions might need to do work even when they can't resolve an error, consider a slight elaboration of the `X::GetParams()` example. The `X::GetParams()` function now creates two separately allocated, semi-permanent data structures. These are a `ParamBlock` structure that gets created in the heap and whose address is stored in the assumed global pointer `gParamData`, and some form of "file descriptor" structure that gets built in the system's area of memory as part of the process of "constructing" the `ifstream`. Once it has built its structures, `GetParams()` calls `X::LoadParamBlock()` to read the data:

```
void X::GetParams(const char *fname)
{
    ifstream in(fname, ios::in | ios::nocreate);

    some code that deals with cases where can't open the file

    gParamData = new ParamBlock;
    …
    LoadParamBlock();
    …
}

void X::LoadParamBlock(ifstream& in)
{
    for(int i=0;i<kPARAMS;i++) {
           code to try to read next of parameters
           …
           code to deal with problems like
                  unexpected end of file
                  invalid data
           }
```

If `LoadParamBlock()` runs into difficulties, like unexpected end of file (for some reason, the file contains fewer data items than it really should), it will need to pass back an error report.

The function `GetParams()` can't deal with the problem; after all, the only solution will be to use another file that has a complete set of parameters. So `X::GetParams()` has to pass the error report back to the function from where it was called.

However, if `X::GetParams()` simply returns we may be left with two "orphaned" data structures. The file might not get closed if there is an abnormal return from the function. The partly filled `ParamBlock` structure won't be of any use but it will remain in the heap.

The mechanism for passing back error reports has to let intermediate functions get a chance to do special processing (e.g. delete that useless `ParamBlock` structure) and has to make sure that all normal exit code is carried out (like calling destructors for automatic objects – so getting the file closed).

Describing errors that can be handled

A function that hopes to deal with errors reported by a called function has to associate with the call a description of the errors that can be handled. Each kind of error that can be handled may have a different block of handling code.

Code to handle exceptional conditions

Obviously, the creation of an error report, the unwinding of the call chain, and the eventual handling of the error report can't use the normal "return from function" mechanisms. None of the functions has defined its return type as "error report". None of the intermediate functions is terminating normally; some may have to execute special code as part of the unwinding process.

The error handling mechanism is going to need additional code put into the functions; code that only gets executed under exceptional conditions. There has to be code to create an error report, code to note that a function call has terminated abnormally with an error report, code to tidy up and pass the error report back up the call chain, code to check an error report to determine whether it is one that should be dealt with, and code to deal with chosen errors.

The C language always provided a form of "emergency exit" from a function that could let you escape from problems by jumping back up the call chain to a point where you could deal with an error (identified by an integer code). But this `setjmp()`, `longjmp()` mechanism isn't satisfactory as it really is an emergency exit. Intermediate functions between the error detector and the error handler don't get given any opportunity to tidy up (e.g. close files, delete unwanted dynamic structures).

Language extensions needed

A programming language that wants to support this form of error handling properly has to have extensions that allow the programmer to specify the source for all this "exception handling" code. The implementation has to arrange that error reports pass back up the call chain giving each function a chance to tidy up even if they can't actually deal with the error.

C++ has added such "exception handling". (This is a relatively recent addition, early 1990s, and may not be available in older versions of the compilers for some of the IDEs). Exception handling code uses three new constructs, `throw`, `try`, and `catch`:

- `throw`
  Used in the function that detects an error. This function creates an error report and "throws" it back up the call chain.

- `try`
  Used in a function that is setting up a call to code where it can anticipate that errors might occur. The `try` keyword introduces a block of code (maybe a single function call, maybe several statements and function calls). A `try` block has one or more `catch` blocks associated with it.

- `catch`
  The `catch` keyword specifies the type of error report handled and introduces a block of code that has to be executed when such an error report gets "thrown" back at the current function. The code in this block may clear up the problem (e.g. by substituting the "default parameters" for parameters from a bad file). Alternatively, this code may simply do special case tidying up, e.g. deleting a dynamic data structure, before again "throwing" the error further back up the call chain.

Rather than "error report", the usual terminology is "exception". A function may *throw an exception*. Another function may *catch an exception.*

26.1 C++'S EXCEPTION MECHANISM

As just explained, the `throw` construct allows the programmer to create an exception (error report) and send it back up the call chain. The `try` construct sets up a call to code that may fail and result in an exception being generated. The `catch` construct specifies code that handles particular exceptions.

Driver with try {… } and catch { … }

The normal arrangement of code using exceptions is as follows. First, there is the main driver function that is organizing things and calling the code that may fail:

```
void GetSetup()
{
    // Create objects ...
    gX = new X;
    …
    // Load parameters and perform related initialization,
    // if this process fails substitute default params
    try {
           LoadParams();
           …
           }
    catch (Error e) {
           // File not present, or data no good,
           // copy the default params
           gParamData = new ParamBlock(defaultblock);
           …
```

```
                // better log error for reference
                cerr << "Using default params because ";
                if(e.Type() == Error::eBADFILE)
                       cerr << "Bad file" << endl;
                else
                if(e.Type() == Error::eBADCONTENT)
                       …
                }
         …
    }
```

This driver has the `try` clause bracketing the call that may fail and the following `catch` clause(s). This one is set to catch all exceptions of type `Error`. Details of the exception can be found in the variable `e` and can be used in the code that deals with the problem.

Intermediate functions

There may be intermediate functions:

```
void LoadParams()
{
    NameStuff     n;
    n.GetFileName();
    …
    gX->GetParams(n.Name());
    …
    return;
}
```

As this `LoadParams()` function has no `try catch` clauses, all exceptions thrown by functions that it calls get passed back to the calling `GetSetup()` function.

As always, automatics like `NameStuff` are initialized by their constructors on entry and their destructors, if any, are executed on exit from the function. If `NameStuff` is a resource manager, e.g. it allocates space on the heap for a name, it will have a real destructor that gets rid of that heap space. The compiler will insert a call to the destructor just before the final return. If a compiler is generating code that allows for exceptions (most do), then in addition there will be code that calls the destructor if an exception passes back through this intermediate `LoadParams()` function.

Functions that throw exceptions

The `X::GetParams()` function may itself throw exceptions, e.g. if it cannot open its file:

```
void X::GetParams(const char *fname)
{
    ifstream in(fname, ios::in | ios::nocreate);

    if(!in.good())
           throw(Error(Error::eBADFILE));
```

It may have to handle exceptions thrown by functions it calls. If it cannot fully resolve the problem, it re-throws the same exception:

```
        gParamData = new ParamBlock;

        // File open OK, load the params
        try {
                LoadParamBlock(in);
                }
        catch(Error e) {
                // Rats!  Param file must be bad.
                // Get rid of param block and inform caller
                delete gParamData;
                gParamData = NULL;
                // Pass the same error back to caller
                throw;
                }

    }
```

The function `X::LoadParamBlock()` could throw an exception if it ran into problems reading the file:

```
    void X::LoadParamBlock(ifstream& in)
    {
        for(int i=0;i<kPARAMS;i++) {
                code to try to read next of parameters
                if(!in.good())
                        throw(Error(Error::eBADCONTENT));
                }
```

Things to throw

But what are these "exceptions" like the `Errors` in these examples?

Essentially, they can be any kind of data element from a simple `int` to a complex struct. Often it is worthwhile defining a very simple class:

```
    class Error{
    public:
        enum ErrorType { eBADFILE, eBADCONTENT };
        Error(ErrorType whatsbad) { this->fWhy = whatsbad; }
        ErrorType     Type() { return this->fWhy; }
    private:
        ErrorType     fWhy;
    };
```

(You might also want a `Print()` function that could be used to output details of errors.)

Suppose there are no catchers

You might implement your code for class `X` using exceptions but your colleagues using your code might not set up their code to catch exceptions. Your code finds a file that want open and so throws an `eBADFILE Error`. What now?

The exception will eventually get back to `main()`. At that point, it causes the program to terminate. It is a bit like an assertion failing. The program finishes with a system generated error message describing an uncaught exception.

26.2 EXAMPLE: EXCEPTIONS FOR CLASS NUMBER

Class `Number`, presented in Section 19.3, had to deal with errors like bad inputs for its string constructor (e.g. "`14l9O70`"), divide by zero, and other operations that could lead to overflow. Such errors were handled by printing an error message and then terminating the program by a call to `exit()`. The class could have used exceptions instead. A program using `Numbers` might not be able to continue if overflow occurred, but an error like a bad input might be capable of resolution.

We could define a simple class whose instances can be "thrown" if a problem is found when executing a member function of class `Number`. This `NumProblem` class could be defined as follows:

```
class NumProblem {
public:
    enum   NProblemType
           { eOVERFLOW, eBADINPUT, eZERODIVIDE };
    NumProblem(NProblemType whatsbad);
    void PrintOn(ostream& out);
private:
    NProblemType    fWhy;
};
```

Its implementation would be simply:

```
NumProblem::NumProblem(NProblemType whatsbad)
    { fWhy = whatsbad; }

void NumProblem::PrintOn(ostream& out)
{
    switch(fWhy) {
case eOVERFLOW:   cout << "Overflow" << endl; break;
case eBADINPUT:   cout << "Bad input" << endl; break;
case eZERODIVIDE: cout << "Divide by zero" << endl; break;
    }
}
```

Some of the previously defined member functions of class `Number` would need to be modified. For example, the constructor that creates a number from a string would now become:

```
Number::Number(char numstr[])
{
        for(int i=0;i<=kMAXDIGITS;i++) fDigits[i] = 0;
        fPosNeg = 0; /* Positive */
        …
        …
        while(numstr[i] != '\0') {
                if(!isdigit(numstr[i]))
                        throw(NumProblem(NumProblem::eBADINPUT));
                fDigits[pos] = numstr[i] - '0';
                i++;
                pos--;
                }
        …
    }
```

(The version in Section 19.3 had a call to `exit()` where this code throws the "bad input" `NumProblem` exception.)

Similarly, the `Number::DoAdd()` function would now throw an exception if the sum of two numbers became too large:

```
    void Number::DoAdd(const Number& other)
    {
        int     lim;
        int     Carry = 0;
        lim = (fLength >= other.fLength) ? fLength :
                        other.fLength;
        for(int i=0;i<lim;i++) {
                …
                }
        fLength = lim;
        if(Carry) {
                if(fLength == kMAXDIGITS)
                        throw(NumProblem(NumProblem::eOVERFLOW));
                fDigits[fLength] = 1;
                fLength++;
                }
    }
```

and the `Number::Divide()` function could throw a "zero divide" exception:

```
    Number Number::Divide(const Number& other) const
    {
        if(other.Zero_p())
                throw(NumProblem(NumProblem::eZERODIVIDE));
        Number Result; // Initialized to zero

        …

        return Result;
    }
```

Code using instances of class `Number` can take advantage of these extensions. If the programmer expects that problems might occur, and knows what to do when things do go wrong, then the code can be written using `try { ... }` and `catch (...) { ... }`. So, for example, one can have an input routine that handles cases where a `Number` can't be successfully constructed from a given string:

```
Number  GetInput()
{
    int    OK = 0;
    Number  n1;
    while(!OK) {
           try {
                  char buff[120];
                  cout << "Enter number : ";
                  cin.getline(buff,119,'\n');
                  Number n2(buff);
                  OK = 1;
                  n1 = n2;
                  }
           catch (NumProblem p) {
                  cout << "Exception caught" << endl;
                  p.PrintOn(cout);
                  }
           }
    return n1;
}
```

The following test program exercises this exception handling code:

```
int main()
{
    Number n1;
    Number n2;
    n1 = GetInput();
    n2 = GetInput();

    Number n3;
    n3 = n1.Divide(n2);

    cout << "n1 "; n1.PrintOn(cout); cout << endl;

    cout << "n2 "; n2.PrintOn(cout); cout << endl;

    cout << "n3 "; n3.PrintOn(cout); out << endl;

    eturn 0;
}
```

The following recordings illustrate test runs:

```
$ Numbers
Enter number : 71O
Exception caught
Bad input
Enter number : 7l0
Exception caught
Bad input
Enter number : 710
Enter number : 5
n1 710
n2 5
n3 142
```

The erroneous data, characters 'l' (ell) and 'O' (oh), cause exceptions to be thrown. These are handled in the code in `GetNumber()`.

The main code doesn't have any `try {} catch { }` constructs around the calculations. If a calculation error results in an exception, this will cause the program to terminate:

```
$ Numbers
Enter number : 69
Enter number : 0
Run-time exception error; current exception: NumProblem
    No handler for exception.
Abort - core dumped
```

The input values of course cause a "zero divide" `NumProblem` exception to be thrown. (These examples were run on a Unix system. The available versions of the Symantec compiler did not support exceptions.)

26.3 IDENTIFYING THE EXCEPTIONS THAT YOU INTEND TO THROW

If you intend that your function or class member function should throw exceptions, then you had better warn those who will use your code.

A function (member function) declaration can include "exception specifications" as part of its prototype. These specifications should be repeated in the function definition. An exception specification lists the types of exception that the function will throw.

Exception specification

Thus, for class `Number`, we could warn potential users that some member functions could result in `NumProblem` exceptions being thrown:

```
class Number {
public:
    Number();
    Number(long);
```

Specification in a function declaration

```
        Number(char numstr[]) throw(NumProblem);

        …

        Number  Add(const Number& other) const throw(NumProblem);
        Number  Subtract(const Number& other) const
                       throw(NumProblem);
        Number  Multiply(const Number& other) const
        throw(NumProblem);
        Number  Divide(const Number& other) const
                throw(NumProblem);

        int     Zero_p() const;
        void    MakeZero();

    …
    };
```

The constructor that uses the string throws exceptions if it gets bad input. Additions, subtraction, and multiplication operations throw exceptions if they get overflow. Division throws an exception if the divisor is zero.

The function definitions repeat the specifications:

Specification in a function definition

```
    Number::Number(char numstr[]) throw(NumProblem)
    {
        for(int i=0;i<=kMAXDIGITS;i++) fDigits[i] = 0;
        …
    }
```

Exception specifications are appropriate for functions that don't involve calls to other parts of the system. All the functions of class `Number` are "closed"; there are no calls to functions from other classes. Consequently, the author of class `Number` knows exactly what exceptions may get thrown and can provide the specifications.

Don't always try to specify

You can not always provide a complete specification. For example, consider class `BTree` that stores instances of any concrete class derived from some abstract `KeyedStorable` class. The author of class `BTree` may foresee some difficulties (e.g. disk transfer failures) and so may define some `DiskProblem` exceptions and specify these in the class declaration:

```
    void BTree::Insert(const KeyedStorable& data)
                        throw(DiskProblem);
```

However, problems with the disk are not the only ones that may occur. The code for `Insert()` will ask the `KeyedStorable` data item to return its key or maybe compare its key with that of some other `KeyedStorable` item. Now the specific concrete subclass derived from `KeyedStorable` might throw a `BadKey` exception if there is something wrong with the key of a record.

This `BadKey` exception will come rumbling back up the call chain until it tries to terminate execution of `BTree::Insert`. The implementation of exceptions is supposed to check what is going on. It knows that `BTree::Insert()` promises to throw only `DiskProblem` exceptions. Now it finds itself confronted with something that claims to be a `BadKey` exception.

unexpected()

In this case, the exception handling mechanism calls `unexpected()` (a global run-time function that terminates the program).

If you don't specify the exceptions thrown by a function, any exception can pass back through that function to be handled at some higher level. An exception specification is a filter. Only exceptions of the specified types are allowed to be passed on; others result in program termination.

In general, if you are implementing something concrete and closed like class `Number` then you should use exception specification. If you are implementing something like class `BTree`, which depends indirectly on other code, then you should not use exception specifications.

Remember the documentation

Of course, the documentation that you provide for your function or class should describe any related exceptions irrespective of whether the (member) function declarations contain explicit exception specifications.