Exception Handling Alternatives (Part 1)

First published in Overload 30

Introduction

After having seen a lot of C++-exception discussions in the previous issues of Overload, I will introduce some alternate methods of handling exceptional events.
Exceptional events occur. But since they are exceptional, they occur very rarely. And this exactly is the problem with them. Though in programming courses you learn to always handle any ever possible event, in practice most programmers just ignore them in most cases. If you look at this problem in detail, you see that these events are not ignored where they actually occur, but at some higher level. E.g. a "Disk full" error is detected in the output routine (`printf` or `operator<<`) and is flagged, but the calling routine ignores the error. This is exactly, why the exception mechanism was introduced into C++. With this, you cannot ignore an exceptional event after it has been detected (and thrown). But exceptions have other problems, as has been discussed in previous articles and as I will show in a future article again. As I wrote earlier, the exception mechanism of C++ is necessary as a substitution for the C `longjmp` mechanism. But I still think it a not so good idea to promote this as the standard mechanism for working with exceptional events. So, I will suggest you some alternative mechanisms that can handle at least some of the events with fewer problems.

Example

As a running example, I use a typical framework system that implements the Builder pattern of the GoF ("Gang of Four", nickname for the authors of [1]).
The application is a CASE graphical editor, and the framework provides classes to read in a design diagram from a file in a specific format (e.g. CDIF) and to build an internal representation of the diagram. The framework provides (among others) an abstract base class `Input`, which has a pure virtual function `readChar()`, which must be implemented by the application developer in a derived class. This function might encounter a problem on reading a character, and the framework designer must decide how this problem shall be reported to the framework and the application. Of course, end-of-input is not an exceptional event and therefore must be handled by normal processing, probably by a special read character returned.
(BTW, this design is a non-template implementation of the Iterator pattern. But there are not many differences if you decide to use a templated approach).

```
class Input
{
public:
      Input();  // The derived class might open a file in the
constructor, or a network connection, ...
      virtual ~Input();  // No (empty) exception specification here,
because you never know what the destructor of the derived class might
do here.  But I will not discuss this issue further in this article.

      virtual SomeReturn readChar() = 0; // About the exact
declaration see the text.
};
```

Another part of the framework provides the classes to build the internal representation. An abstract base class `DiagramBuilder` is provided that declares the virtual functions for building the parts of the diagram. The application must subclass `DiagramBuilder` and implement these functions.

```
class DiagramBuilder
{
public:
      // ...
      virtual void buildEntity(EntityInfo&) = 0;
      virtual void buildRelation(RelationInfo&) = 0;
      // ...
};
```

Return Codes

The first alternative to C++ exceptions in this discussion are return codes, which were extensively discussed by Ken Hagan and The Harpist in the last issue.
If you must solve the example problem with return codes, you might find a special character to return in case of a problem, but a better solution would be to declare `readChar` like this:

```
ReturnStatus readChar(char &c);
```

Here, `readChar` returns the read character in `c` and reports a problem as a return value of type `ReturnStatus`, which might be a simple enum or a complete object containing all interesting information for the problem. In the latter case, the return value should be returned by reference, as the returned object might be of some class derived from `ReturnStatus`.
For `DiagramBuilder`, the functions must be declared to return a status (probably again a derivation of `ReturnStatus` by reference).
The problem with this approach is that the framework does not know what to do with the event, and must now unwind the stack manually until it can return the error status to the application. This is the classic problem of error handling in frameworks, where error detection and error handling are absolutely not local.
But, where you could perhaps use return codes quite well is inside of the framework. In this aspect, I perhaps agree with Ken Hagan. Where error handling is local, and you are in control of both the source of the exceptional event and the handling of it (and, of course, your source is not inside of a constructor or an overloaded operator), then return codes are probably the best technique for handling exceptional events. But, you must be sure that you are really in complete control of both the source and handling points, i.e. neither of them is inside a virtual function that may be overwritten in a derived class. So, on second thoughts, for frameworks where most classes may be specialized by the user of the framework, return codes are not a good idea for handling exceptional events.

The main problems with return codes for error handling still are:
(1) They are unusable with constructors, destructors and overloaded operators.
(2) They are difficult to propagate to the place, where the event can be handled consistently for the whole application. In our example, the problem occurs inside an application function, and it must be handled by the application, but the framework, which doesn't know what kind of problems might occur, must itself propagate the error.
(3) They can be ignored.

Deferred Error Handling

Another possibility for handling a problem is "deferred error handling" (as I have seen first proposed by James Kanze in [3]). This technique is used by the "classic" iostream library. Here, an object internal error flag is set, if a problem occurs. This flag can be checked by clients using a special member function (e.g. overloaded !-operator). All member functions of the class check the flag and act respectively (probably by doing nothing).
In our example, `readChar()` could set an internal flag, and return end-of-input. This would cause the framework either to report an error ("premature end of file") or just to finish with an incomplete representation. In both cases the application has to check for the flagged error and to handle it appropriately. To be able to identify the error correctly, the flag must not only be a binary value but probably a full exception object.
If a function of `DiagramBuilder` detects a problem, again a flag is set, but the function will return normally. So, the framework will not notice that an exceptional event occurred and will just continue to create the diagram. But, all subsequent calls to one of the member functions of `DiagramBuilder` will see the set error flag and just do nothing. After the framework has completed the diagram, it will return to the application, which will check the error flag and handle it.
In all cases, the framework has nothing to do with the handling of any of the exceptional events; it won't even notice that one has occurred.
One problem arises with the location of the flag: the objects for `Input` or `DiagramBuilder` may no longer exist when the diagram is completed. So, the flag must be in an object that still exists after the completion of the building process. For `DiagramBuilder`, this would obviously be the created diagram representation, but for `Input` it is difficult to find a good location, as the input object itself is created by the framework and only visible from there.

The advantage of deferred error handling is the invisibility to the non-interested parts of the application. In our example, inside the framework the errors are completely transparent.
A disadvantage of deferred error handling is the long time, which might pass between the detection and the handling of an error. And it must be carefully checked that no endless loops are created. E.g. if the frameworks needs some information from the diagram representation to complete its task, this information must be given.
Deferred error handling is best used for classes whose responsibilities are non-critical, e.g. output for purely informational logging. In this case, the application is run until successful completion, and than the error flag is checked and some notification message might be issued.

Error Stack

David Vandevoorde proposed in [4] the usage of a global error stack. This is effectively a variant of the standard C++ exception mechanism, but without hidden control flows and everything (including stack unwinding) must be done explicitly. When a problem is detected, instead of throwing an exception, or to flag it internally (as in the deferred error handling mechanism), it is pushed onto a global stack (in a multi-threaded environment, you have to decide whether you want one stack per thread or one per process). Then, each critical section (or the member functions of critical objects) has to check the error stack for consistency of the manipulated object. If a problem is pending (there is some exception on the stack), nothing is done. Insofar this is similar to the deferred error handling mechanism. But, any function can decide to handle the exception and pop it from the stack. Insofar it is similar to standard exception handling. Where you would have a catch clause, you now just check for an entry on the stack. In our example, both the framework and the application can use this mechanism. If only the application were to use an error stack then `readChar` and the functions of `DiagramBuilder` would put an exception on the stack when a problem occors, and they would check for an exception always on entry and act as in the deferred error handling mechanism. When the framework eventually returns control to the application, the application would check for any pending exceptions on the stack.
If the framework also implemented the error stack mechanism, then the application would have to use the same stack, and `readChar` and the `buildX` functions would have to push their exceptions on this same stack. The framework then typically checks the stack for exceptions directly after each call to one of the functions provided by the application. If an exception is pending it has to unwind the stack manually (i.e. return from all its functions) and return control to the application, which then has to handle the exception. In this case, the framework reports its own problems to the application also via the exception stack.
The error stack mechanism is essentially the same mechanism as standard C++ exception handling. You typically cannot do anything useful as long as there are unhandled exceptions. Your only actions when you have hanging exceptions are in most cases clean-up actions and manual stack unwinding (returning from the function). But you have no hidden stack unwinding without your control, and you can have more than one exception at the same time on the stack, so you don't have artificial (i.e. coming from the C++ environment and not from the problem domain) problems with exceptional events in destructors. But, since the error stack is not a standard mechanism of C++, you cannot assume that any 3rd party code use this technique. And of course, you loose the capabilities of the built-in mechanisms. So, if the constructor of one element of an array fails, but this failure is only reported by an exception on the stack, all other elements are constructed as well, and you have to destruct them again manually. But this is more a problem of efficiency and not of the principle. Another problem is to find the best match for the error handling action. This is a typical multiple dispatch problem, as you have two criteria (the place of the handling and the type of the exception) for which you must find the best matching action. With C++ exceptions, the C++ environment helps you to define this match. With an error stack, you have to do it yourself, and you must use some technique as described in the GoF Visitor pattern (or one of its many variants). Yet another problem is the possibility of ignoring the exception and performing unsafe actions on inconsistent objects. But this can (at least partially) be avoided by the mixing with the deferred error handling mechanism.
An error stack is effectively a mix of all the other techniques (is has elements of C++ exceptions, return codes and deferred error handling). But as a mix, it has its own mix of pros and cons: No hidden control flow, but it can be ignored, no problems with exceptions from destructors, but no compiler supported best match of handling action. And as it is a rather unknown technique I would not use it as the standard exception handling mechanism for the framework, but it might be a good choice for the application part of our example (i.e. the transfer of problems from `readChar` and `buildX` through the framework back to the application.

Safe Termination

Often, a safe termination of the program is the best approach.
James Kanze provided in [5] the following categorization of applications:
(1) Critical: if this fails, a backup system will step in
(2) Small standalone programs like compilers: if they crash they will just be restarted by the user
(3) Other; e.g. this might be a server in a client/server application: it should not crash, but there is no special fail-over environment.
While at first sight a lot of applications seem to belong to category (3), on closer look a lot of applications can be seen as category (1) or (2) applications. E.g. a server might spawn a new process for each client, and if that process crashes then either the client must be restarted (then the server belongs to category (2) because it only serves this client), or the server is restarted automatically (by a process monitor) and the client connection is automatically recreated; in this case the server belongs to category (1), even if there is no special hardware backup system.
Instead of a safe termination (which is the normal solution for categories (2) and (3)), you might terminate at once and produce a post mortem dump to debug the problem. This is the best approach for detecting programming errors. (BTW, unfulfilled pre-conditions (such as "index out of range" or "pop from empty stack") must generally be treated as programming errors – and what can you usefully do with a programming error other than to terminate the program and debug?)
Our example probably falls into category (2); so, termination might be a useful solution. So, `readChar` and the functions of `DiagramBuilder` would just terminate the program on detection of a problem. One way to terminate the program is the EndProgram exception, which enables all functions on the stack to correctly clean up all held resources. For applications with resources that are not freed by the operating system this is one way to avoid resource leaks. But, this again introduces C++ exceptions into the system. A solution to avoid C++ exceptions is to register all external resources at a global place and to free them from the termination routine. Other duties of the termination routine are to inform other threads and/or processes about the termination. When the process is restarted by a process monitor, this monitor could be informed of open client connections etc. so that the restarted process could seamlessly fill in the place of the terminated one.

Review So Far

For the design of the example framework error handling, there are two different decisions to make:
(1) how to report errors of the framework to the application.
(2) how to let the application's derivations of the framework classes report problems.
In both cases, the application has to handle the problem, and not the framework. All the above mechanisms and standard C++ exception handling could be used as design solution. And, as we have seen, it might make sense to use different mechanisms for (1) and (2). And even for `readChar` and `buildX`, which are both of type (2), it could be useful to use different mechanisms. So, the designer of a truly generic framework should not setup a fixed error handling mechanism, but leave these decisions open to the application designer. How to do this easily, and how to solve other problems of all the above mechanisms as well, I will show you in the next Overload issue.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns, Addison-Wesley 1994, ISBN 0-201-63361-2
[2] In spring 96, there was a very interesting Usenet discussion "Safe use of exceptions: possible? worth the trouble?" in comp.lang.c++.moderated. The whole discussion is still very recommended to read. The thread started with Message-ID <4ljuch$aap@netlab.cs.rpi.edu>
[3] James Kanze, above discussion, Message-ID <4ncc5ua@netlab.cs.rpi.edu>
[4] David Vandevoorde, above discussion, Message-IDs <4mlfh0$g@netlab.cs.rpi.edu > and <4mr6fr$qqv@netlab.cs.rpi.edu >
[5] James Kanze, above discussion, Message-ID <4n07r5$mkg@netlab.cs.rpi.edu >