Exception Handling Alternatives (Part 2)

First published in Overload 31

Resume

In part 1, several alternative mechanisms for handling exceptional events were presented. One of the conclusions was that a framework (or library) should not impose a fix error handling mechanism on the application design. Also, all of the proposed mechanisms in part 1 (incl. standard exception handling) have one disadvantage in common: they all loose the context of the error event.
As I have stated several times, C++ standard exception handling is the necessary substitution for the C `longjmp` mechanism. But the `longjmp` error handling mechanism in C consists of two parts (there are other error handling mechanisms in C, e.g. `errno`). While the handling/recovery mechanism is (at least sometimes) the `longjmp` mechanism, the other part is the notification of an error event. This is done through the signal mechanism.
The C signal mechanism is split into three steps:

1. registration of a signal handler
   this is the way to say "I'm prepared to handle this specific problem" and can be roughly compared to `try`
2. raising of a signal
   this is the way to say "this specific problem occurred" and can be compared to `throw`
3. handling the problem
   here, the problem is solved or recovery is tried (often using `longjmp`)

(Note: There are synchronous and asynchronous signals. While step 1 is identical for both, step 2 and 3 are quite different; asynchronous signals are raised by the operating system and for the handling only very limited functionality is available, `longjmp` not among them.)
Though the above description of C signal handling looks quite similar to C++ exception handling, there is one big difference: the signal handler is put on top of the stack, instead of unwinding it. So, you can handle and solve the problem and resume the program flow where the problem was detected by just returning from the signal handler function. Of course, if you can not solve the problem, you have to try to recover from some save point, for which `longjmp` is used, which then unwinds the stack (actually, it is not unwound but just cut off). In C++, a similar mechanism was introduced with the `new_handler()`, but in general, C++ promotes exception handling as the standard mechanism for error events.

Callbacks

C signal handling is just an implementation of the general callback mechanism. And a general callback mechanism can help solving the design problems with error handling. The callbacks just defer the decision which mechanism to use to handle an exceptional event to a very late design time. You can leave the decision open until the complete application is assembled. So you have not to decide about the mechanism when you write the framework, not even when you write the classes and functions the framework uses, but not before you put everything together and actually use the framework. And at that time, you still have all the options to use one of the mechanisms I showed you in the first part. Well, not really all of them. You couldn't provide return codes, and deferred error handling must at least partially be supported by the classes you added to the framework.
But, you have the three main options to terminate the program, to push the error event on a global stack, or to throw a standard exception (for this, the framework and application must be exception safe). And you have another important option with callbacks you wouldn't have otherwise: you can try to solve the encountered problem and then continue with the normal flow of the program! Do you remember the infamous "A: not ready. Abort, Retry, Ignore?" when you forgot to insert the floppy disk into the drive? You just inserted the disk and then the program could continue. Callbacks are a means to do things just like this.
In our example, `readChar()` could call a callback function in case of a problem. An interactive application could register a callback that asks the user what to do: solve the problem (retry), terminate the program (abort) or just continue (ignore), in which case an end-of-file marker could be returned (with the framework possibly returning a "Premature end of file" error). A batch application (e.g. a standalone program that loads some CASE data and checks it for consistency) could register a different callback that just terminates (and perhaps sends some error message to `cerr`).

Object-Oriented Callbacks

So far, the callback mechanism was just used like the simple C signal mechanism. But C++ provides more. An object-oriented callback mechanism uses two objects for the handling of an error event. One object for the event itself (an exception object) and another object for the handler. This handler object is just a function object as is usually used in C++ for callbacks. So, you can add data to the individual callbacks. And you have inheritance. A read exception could be handled by a more general IO handler. This is exactly what you do if you throw a derived exception and catch a base class of that exception. With C++ standard exceptions, the compiler provides you that mechanism. With callbacks, you have to implement your own.

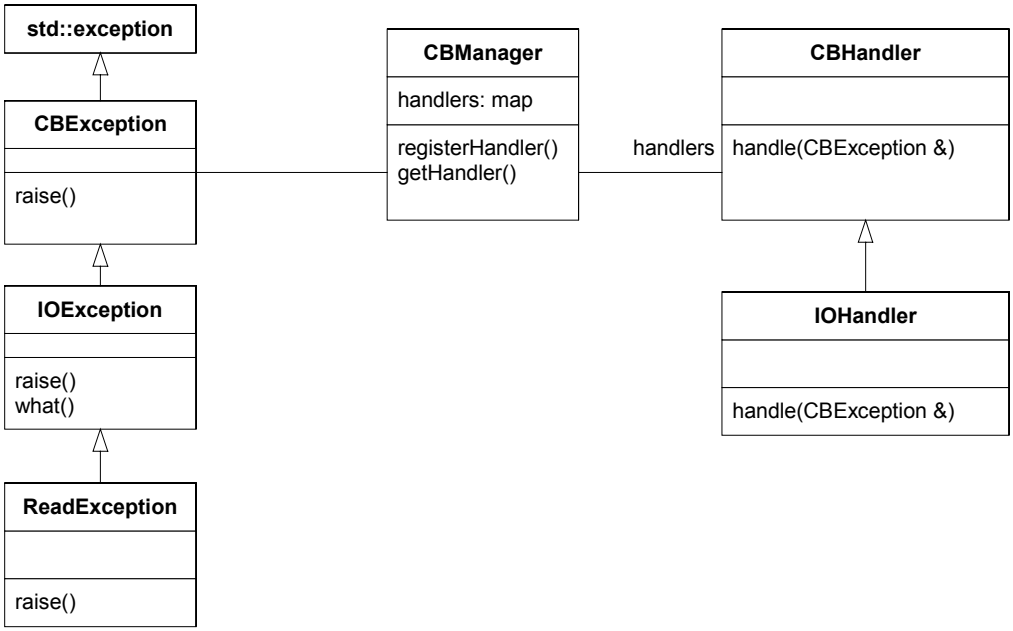
The problem here is partly the typical double dispatch problem, since you must find the best match for the exception and the handler. A common solution for this kind of problem is the Visitor pattern. But most implementations of Visitor that I've seen depend on compile-time resolution. Since handler callbacks are registered at run-time, that approach cannot be used here. Martin E. Nordberg proposed in [1] for his Extrinsic Visitor the usage of a `map<>` based on type IDs, and this is the approach chosen here. This allows a dynamic lookup to find the registered handler for an exception.

But one problem remains: if for the exception to handle no handler was registered, but only for a base class, then the lookup will not find it. To solve this, another pattern from the GoF book [2] is used: Chain of Responsibility. This is used to look for a handler for the base class, if none was found for the original exception. This is continued until the top of the exception hierarchy is reached. If no appropriate handler was found, `terminate()` will be called (before stack unwinding!).

Sample Implementation

The implementation shown here consists of the two class hierarchies for exceptions and handlers and of the manager that registers the handlers and dispatches the exception.

(N.B. At any single point in time, we don't really have a double dispatch, as you have only one set of handlers that are currently registered and of which you must find the best suited one. So, we have only one `map<>`, and not one for each handler hierarchy as we would have in the original double dispatch problem.)



Class Diagram for Callback Example

```
class CBHandler;
template <typename Base, typename Current>
void raise(Current &self)
{
      const CBHandler *h = 0;
      if (h = currentCBManager()->getHandler(typeid(Current), self))
      { // a handler exists for this exception and is called now
```

```
            h->handle(self);
      }
      else
      { // no handler for current class, look for base class
        // this implements the Chain of Responsibility
            self.Base::raise();
      }
}

class CBException : public std::exception
{ public:
      virtual ~CBException() throw() {}
      virtual void raise() = 0 { terminate(), abort(); }
};

class IOException : public CBException
{ public:
      IOException(const char *reason) throw() { text = reason; }
      virtual void raise()
      { ::raise< CBException>(*this); }
      virtual const char *what() const throw()
      { return text; }
private:
      const char *text;
};

class ReadException : public IOException
{ public:
      ReadException(const char *reason) throw(): IOException(reason)
{}
      virtual void raise()
      { ::raise<IOException>(*this); }
};
```

`CBException` is an abstract base class, but nevertheless must provide an implementation for its pure virtual `raise()` function. Here, `CBException` is derived from `std::exception`, but this wouldn't be really necessary in the general case.

In the presented solution, I definitely don't like the `raise()` functions in the concrete exception classes. They are all identical except the argument of the template `raise()` function. The problem is the argument that gives the base class. As they are, the functions are typical candidates for copy-and-paste, with a big potential to introduce programming errors by accidentally giving the wrong argument to the template.

While I understand that it is difficult with multiple inheritance to provide some keyword `base` or `super` to access the base class, RTTI with `type_info` could provide some mechanism. This is just one example for the shortages of the standard RTTI. While it wouldn't have put much burden on the compiler implementers to provide a full reflection functionality like `java.lang.reflect`, it would have been a great relief for programmers who need some meta-class facilities.

```
class CBHandler
{ public:
      virtual ~CBHandler() {}
      virtual void handle(CBException &) const = 0;
};

class IOHandler : public CBHandler
{ public:
      virtual void handle(CBException &e) const
      {
            IOException *ioe = dynamic_cast<IOException *>(&e);
            if (ioe == 0) { terminate(), abort(); }
            // handle the exception -- here just throw it
```

```
            throw(*ioe);
      }
};
```

Just one concrete handler is given here as an example. This one throws the exception to handle (hence the derivation of `CBException` from `std::exception`). Another handler could do anything, for which it might need some data. This would be given by the constructor. Note that handlers are created and destroyed at "safe" times, i.e. not while an exception is waiting for handling. So, no empty exception specifications are necessary for constructors and destructors.

In `handle()`, if the exception is no `IOException` (the only we are prepared to handle), something went definitely wrong and the program is terminated, either by `terminate()`, or if that returns, by `abort()`.

`void` is definitely not a good return type for `handle()`, as it cannot tell the raising function whether the problem was successfully handled or not. But any status type would probably do (depending on the actual environment).

```
class CBManager
{ public:
      const CBHandler *getHandler(const type_info &et, const
CBException &e) const throw()
      {
            std::map<const char*, const CBHandler*>::iterator hp =
handlers.find(et.name());
            if (hp != handlers.end()) { return hp->second; }
            else { return 0; }
      }
      const CBHandler *registerHandler(const type_info &et, const
CBHandler *newH)
      {
            const CBHandler *oldH = 0;
            std::map<const char*, const CBHandler*>::iterator hp =
handlers.find(et.name());
            if (hp != handlers.end())
            {     // handler already registered
                  oldH = hp->second;
                  if (newH) { hp->second = newH; }
                  else { handlers.erase(hp); }
            }
            else { handlers[et.name()] = newH; }
            return oldH;
      }
private:
      std::map<const char*, const CBHandler *> handlers;
};

CBManager *currentCBManager() throw();
```

With already using two patterns in this small system, you might think that `CBManager` would be a good candidate for yet another pattern, namely Singleton. But this is not a good idea. Though at any one time only one manager is valid (which is returned by `currentCBManager()`), there could be other instances of `CBManager` around so you could easily replace the whole set of handlers at once by just swapping complete manager objects. And for multi-threaded applications you definitely need at least one separate instance for each thread.

`currentCBManger()` is declared here, but it is the responsibility of the application to implement this function.

```
CBManager globalCBManager;

CBManager *currentCBManager() throw()
{
      return &globalCBManager;
```

```
}

void f2()
{
      ReadException("Something went wrong").raise();
}

void f1()
{
      try
      { f2(); }
      catch (const IOException &ioe)
      { std::cerr << ioe.what() << std::endl; }
}

int main()
{
      IOHandler ioh;
      const CBHandler *oldH =
globalCBManager.registerHandler(typeid(IOException), &ioh);

      f1();

      globalCBManager.registerHandler(typeid(IOException), oldH);
      return 0;
}
```

`currentCBManager()` returns in this simple example just the address of a global instance. `main()` creates a local object of `IOHandler` and registers it. It is therefore also responsible to unregister the handler and reinstall the previous one. Though the handler is registered in `main()`, the exception thrown by this handler is caught in `f1()`.

Summary

C++ standard exceptions are not the only way to handle exceptional events. There are several other viable mechanisms for that purpose. I personally prefer the callback mechanism as it provides greatest flexibility. But, though the mechanism for finding a registered callback for a given exception it quite similar to finding a matching catch for a thrown exception, there is no compiler support for it and so you have to implement your own. Therefore the performance for the callback mechanism is probably worse than throwing an exception, and it goes even worse, if the handler itself just decides to throw. But as this only happens if really an exception occurs, which should be really exceptional, this doesn't really matter.

[1] Martin E. Nordberg: Default and Extrinsic Visitor, in: Robert C. Martin, Dirk Riehle, Frank Buschmann (eds.): Pattern Languages of Program Design 3, Addison-Wesley 1998, ISBN 0-201-31011-2
[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns, Addison-Wesley 1994, ISBN 0-201-63361-2