Document Number: WG21/N0842=X3J16/96-0024
Date: 23 Jan 1996
Project: C++ Standard Library
Reply to: Nathan Myers
<ncm@cantrip.org>

Exception Safety for Iostream

The iostream library xalloc()/pword() mechanism provides an essential tool for runtime extension of iostream semantics to support user types. While this mechanism works at a very low level, involving casts from void* pointers, its use is mostly easily encapsulated, and because it is simple, it is mostly quite reliable and safe, if used correctly.

The Problem

I say "mostly", above, because there is one area in which it is not safe, and not encapsulable. Imagine you have implemented a type named "Date", for which you have implemented operators << and >>. Imagine further that as an optimization in operator<<, you would like to cache some data in the istream argument in storage provided by pword().

There are two problems in this scenario. First, if operator<< stores anything via pword(), somebody needs to delete that storage when the istream itself goes away. Requiring the owner of the istream itself to clean up breaks encapsulation in Date. Second, the owner of the istream may get no opportunity to clean up if an exception occurs.

Discussion

In practice, these problems mean that library components cannot use the iostream xalloc()/pword() facilities safely. Clearly we need some way for Date to get control during events that require this kind of cleanup. The traditional solution for this kind of problem at runtime is to use callbacks. We need to provide a callback mechanism for important iostream events.

It is not immediately obvious in which class the registry belongs. More particularly, in which destructor do the callbacks occur? They could be called from ~ios_base, ~basic_ios<>, ~basic_istream<>(), or even (e.g.) ~basic_istringstream() -- or all of the above. Of course the more-derived class's destructors have access to more of the stream's resources; by the time the ~ios_base() destructor is reached most such resources have already been released.

Because the purpose for this is simply cleanup, the simplest alternative seems best: place the registry in ios_base.

The next question is, what are the interesting events? It would be foolish to add a callback mechanism and fail to hand over control when it's needed. The only other event of significance (for cleanup) in an ios_base is change of locale.

Proposed Resolution

Add to the definition of class ios_base the declarations:

```
  enum event { imbue_event, destruct_event };
```

```
  typedef void (*event_callback)(event, ios_base&, int index);
  void register_callback(event_callback fn, int index);

and define register_callback():

  Effect: Registers the pair (fn, index) such that during calls to
   imbue() or ~ios_base(), the function *fn* is called with argument
   *index*.  Functions registered are called when an event occurs,
   in opposite order of registration.  Functions registered while a
   callback function is active are not called until the next event.
  Notes: No attempt is made to merge identical pairs; a function
   registered twice is called twice per event.

Add to the description of destructor ~ios_base():

  Calls each registered callback pair (fn, index) as (*fn)(destruct_event,
  *this, index) at such a time that any ios_base member function called
  from within fn has well-defined results.

Add to the description of ios_base member imbue(const locale& loc):

  Calls each registered callback pair (fn, index) as (*fn)(imbue_event,
  *this, index) at such a time that a call to ios_base::getloc() from
  within *fn* returns the new locale value *loc*.
```