Exception Safe Exceptions

Gregory Colvin
Information Management Research
gregor@netcom.com

The class exception

As presently specified the class exception may itself throw exceptions. If using an exception can throw an exception the result may be infinite loops, unbounded recursion, or worse. I propose to seal this hole by specifying an interface which will not throw exceptions.

Interface

class exception {
public:
  exception(const char*) throw();
  exception(const string&) throw(alloc);
  exception(const exception&) throw();
  virtual ~exception() throw();
  exception& operator=(const exception&) throw();
  virtual const char* what() throw();
};

Semantics

An exception is a container for a null-terminated-multi-byte-string (NTMBS). The member function what() returns a pointer to a NTMBS which compares equal (in the sense of strcmp()) to the NTMBS so contained. Note that only the constructor exception(const string&) throw(alloc) may throw an exception. A conforming implementation will not call unexpected(). The Library and Language will not use the constructor exception(const string&).

Expression | Precondition | Postcondition
---|---|---
exception x(p) | p points to a NTMBS with storage duration exceeding that of x or any copy to be made of x. | strcmp(x.what(),p) == 0
exception x(s) | s is a string. | strcmp(x.what(),s.c_str()) == 0
exception x(e) | e is an exception. | strcmp(x.what(),e.what()) == 0
x = e | x and e are exceptions. | strcmp(x.what(),e.what()) == 0

Discussion

Some representatives object to the use of char* in the exception interface, in part because it is easy to violate the precondition for exception(const char*). There is however an easy way to meet the precondition: use a NTMBS with static storage duration, e.g. exception("bad"). Also, having what() return a const char* is a change from the current string return value. Since the string class provides a const char* constructor I believe no important functionality is lost. An alternative would be to specify a string-only interface which disallows exceptions, but this might unduly constrain implementations of the string class. Note also that since the Language and the Library do not use exception(const string&) no string code need be linked in, which meets the strong objections of some representatives to having the language depend on string.
The template *do_throw*

We recently removed from the Library a means of preventing exceptions from being thrown by the Library or the Language. The objection to this facility seemed not to be that it was unnecessary, but that the machinery was needlessly complex. I hope that the following interface is simple enough to prove acceptable.

**Interface**

```c
void (*set_throw_handler(void(*)(const exception&)))(const exception&);
void throw_handler(const exception&);
template<class X> void do_throw(const X &x) throw(X);
```

**Semantics**

The function
```c
void (*set_throw_handler(void(*)(const exception&)))(const exception&);
```
installs the function pointer *pf* as the current throw-handler and returns the previous throw-handler if any, or 0.

The function
```c
void throw_handler(const exception& x);
```
passes a reference to the *exception x* to the current throw-handler, if any.

The function template
```c
template<class X> void do_throw(const X &x) throw(X);
```
passes a reference to the *exception x* to the current throw-handler, if any, and then executes the expression *throw x*.

Library exceptions and the Language exceptions *bad_cast* and *bad_typeid* are thrown (as if) by *do_throw()*.

**Discussion**

Since *do_throw()* is used to throw Library exceptions, as well as the Language exceptions *bad_cast* and *bad_typeid*, users can prevent exceptions from being thrown by installing a throw-handler which does not return. In addition, users can specialize the *do_throw* template to provide custom handling of their own exception classes.

An alternative, but one which requires more language support, would be a single "magic" template function:
```c
template<class X> void (*set_throw_handler(void(*)(const X&)))(const X&);
```
The language would establish a throw-handler for every type of object thrown and call it, if set, as the first part of each throw expression. I believe this alternative was first suggested by Nathan Meyers.