Exception Prevention

Gregory Colvin
Information Management Research
gregor@netcom.com

In San Diego we removed from our Library a facility for preventing exceptions from being thrown by the Library or the Language. I review herein three alternative means of providing this facility and recommend that we adopt the *do_throw* template of 94-0166=N0553.

Alternative 1: The template *do_throw*

At Valley Forge I proposed the following interface.

Interface

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

Semantics

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

The template functions
```
template<class X> void do_throw(const X &x) throw(X);
```
pass a reference to the *exception* referred to by *x* to the current throw-handler, if any, and then execute the expression *throw x*.

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

Alternative 2: A templatized *set_throw_handler*

An alternative which requires language support would be a "magic" function template. I suggested this alternative at Valley Forge.

Interface

```
template<class X> void (*set_throw_handler(void(*)(const X&)))(const X&);
```

Semantics

The language would establish a throw-handler for every type of object thrown and call it, if set, in every *throw-expression* before transferring control to a *handler*, with a reference to the object being passed to the handler.

Alternative 3: A templatized *operator throw*

Another alternative which requires language support would be a "magic" operator template. Although discussed in San Diego, and since then on the reflectors, this alternative has not been proposed formally.

Interface

```
template<class X> void operator throw (const X&);
```

Semantics

This template specifies a family of potentially replaceable functions. The default version of each such function transfers control to a *handler*, passing the object referred to by its argument.

Discussion and Recommended Action

The *do_throw* template requires no language support to allow users to intercept exceptions thrown by the Standard Library, and only minimal support (the insertion of a function call) for the Language exceptions *bad_typeid* and *bad_cast*. The user must make a function call to intercept exceptions, which means that exceptions thrown during static initialization of the Standard Library are impossible to intercept. (Note that users cannot catch such exceptions, as they will cause *terminate()* to be called before any user code can be executed). Also, exceptions not thrown by *do_throw* cannot be intercepted, which fact may discourage the use of raw *throw-expression*s.

The *set-throw-handler* template requires substantial language support: the language processor must magically provide a *throw-handler* for every type of object thrown. I am unsure of the implementation difficulty, but suspect that it could range from trivial to very difficult. The trivial approach is to generate a definition of each required *throw-handler* in each translation unit requiring it and sort out the multiple definitions "at link time". The *set-throw-handler* template cannot handle exceptions thrown by the Library during static initialization, but it does allow all other *throw-expression*s to be intercepted.

The *operator throw* template also requires substantial language support: the language processor must magically replace the default operator "at link time". This might turn out to be a trivial reuse of mechanisms required for template specialization, but I doubt it. Since the replacement is static, all *throw-expression*s can be intercepted.

Which solution to choose depends in part on what problem we need to solve.

The simplest problem is to allow users who are not prepared to handle exceptions to use C++ and its Standard Library. The *do_throw* template is adequate to solve this problem, so long as users are prepared to install a *throw-handler* and avoid all use of raw *throw-expressions*.

A harder problem is to allow users to intercept exceptions arising from any *throw-expression*. A templatized *set-throw-handler* is just adequate to solve this problem, unless we insist on intercepting exceptions thrown during static initialization of the Library.

An even harder problem is to allow users to exceptions arising from any *throw-expression* whatsoever. A templatized *operator throw* is adequate for this purpose.

Given the late date, and the likely implementation difficulties of more radical alternatives, I recommend we adopt the *do_throw* template.