Doc No: X3J16/94-0179 WG21/N0566 R1
Date: 28 Feb 95
Project: Programming Language C++
Ref. Doc: X3J16/94-0179 WG21/N0566
Reply to: Michael J. Vilot
ObjectCraft, Inc.
7 Colby Ct., suite #4–321
Bedford NH USA 03110–6427
mjv@objects.mv.com

Language Support Exceptions

1. Introduction

This proposal was discussed at the last meeting, but did not generate sufficient interest to recommend it to X3J16/WG21. However, several comments received during the CD registration ballot indicate a need to reconsider it.

This proposal recommends changes to the specification for the exceptions `bad_alloc`, `bad_cast`, `bad_typeid`, and `bad_exception`,[1] described in Clause 18, Language support library.

2. Discussion

The essential change is to separate these three exceptions from the class hierarchy described in Clause 19, Diagnostics library. One of the main concerns, discussed on the `c++std-lib` reflector last year, is the dependency on other portions of the C++ Standard Library, and the resulting overhead associated with each exception instance. A secondary concern was the apparent need for `string`-related dynamic memory allocation, even in low-memory (i.e. `bad_alloc`-provoking) situations.

CD Registration Ballot comments echoed these concerns:

Australia: "The exception class hierarchy should use virtual inheritance." (R-13)[2]
France: "The general structure of the libraries shall be revised. In particular, it is necessary to DECOUPLE libraries. In the current proposal, there are unacceptable cross and forward references between libraries (and sections)." (R-26)
Germany: "As specified in the document the class `exception` may itself throw exceptions. Thus exceptions can be thrown while using an exception, which might result in infinite loops or unbounded recursions. This problem has to be resolved." (R-40)
Netherlands: [The C++ Standard] "should contain the definition of the language itself and a minimal set of libraries (effectively only the language support libraries …)." (R-44)
New Zealand: "have a library section which contains only the minimum needed for conformance testing." "Once again, it is necessary to ask 'What about the implementation for a 4-bit micro-controller?'" (R-53)

The first comment increases the complexity and overhead of exceptions derived from class `exception`. The other four comments argue for less complexity and overhead, especially in the language support exceptions.

This proposal addresses these concerns by eliminating the dependencies on both the `string` component and the `exception`/`runtime_error` exception classes. It provides the simplest, least-overhead exception instances. These exceptions can be treated as fixed-size, value-oriented classes that do not even require virtual tables. The `bad_alloc` exception can be implemented for the special case of low-memory conditions.

---

[1] Formerly `Xunexpected`.
[2] These reference numbers correspond to those used by Sam Harbison in Message `c++std-admin-87`.

3. Proposal

Affects Clause: 18.

Change the specifications of `bad_alloc`, `bad_cast`, `bad_typeid` and `bad_exception` to the following:

18.4.2.1 Class `bad_alloc` [lib.bad.alloc]

```
namespace std {
  class bad_alloc {
  public:
    bad_alloc();
   ~bad_alloc();
    const char* const what() const;
  private:
  // unspecified
  };
}
```

The class `bad_alloc` defines the type of objects thrown as exceptions by the implementation to report a failure to allocate storage (5.3.4, 18.4.2.2).

18.4.2.1.1 `bad_alloc` constructor

```
bad_alloc();
```

Effects: Constructs an object of class `bad_alloc`.
Notes: Shall not require the use of dynamic storage allocation (3.7.3).[3]

18.4.2.1.2 `bad_alloc::what`

```
const char* const what() const;
```

Returns: An implementation-defined value.[4]
Notes: Shall not require the use of dynamic storage allocation (3.7.3).

18.5.2.1 Class `bad_cast` [lib.bad.cast]

```
namespace std {
  class bad_cast {
  public:
    bad_cast();
   ~bad_cast();
    const char* const what() const;
  private:
  //  unspecified
  };
}
```

The class `bad_cast` defines the type of objects thrown as exceptions by the implementation to report the execution of an invalid *dynamic-cast* expression (5.2.6).

---

[3] A plausible implementation would be a statically allocated string literal (2.9.4).
[4] The value could be a multi-byte encoding that can be converted to a wide-character string (21.2).

18.5.2.1.1 bad_cast constructor

```
bad_cast();
```

Effects: Constructs an object of class `bad_cast`.

18.5.2.1.2 `bad_cast::what`

```
const char* const what() const;
```

Returns: An implementation-defined value.[5]

18.5.2.2 Class `bad_typeid` [lib.bad.typeid]

```
namespace std {
  class bad_typeid {
  public:
    bad_typeid();
   ~bad_typeid();
    const char* const what() const;
  private:
  //  unspecified
  };
}
```

The class `bad_typeid` defines the type of objects thrown as exceptions by the implementation to report a null pointer in a type identification expression (5.2.7).

18.5.2.2.1 `bad_typeid` constructor

```
bad_typeid();
```

Effects: Constructs an object of class `bad_typeid`.

18.5.2.2.2 `bad_typeid::what`

```
const char* const what() const;
```

Returns: An implementation-defined value.[6]

18.6.2.2 Class `bad_exception` [lib.bad.exception]

```
namespace std {
  class bad_exception {
  public:
    bad_exception();
   ~bad_exception();
    const char* const what() const;
  private:
  //  unspecified
  };
}
```

The class `bad_exception` defines the type of objects thrown as exceptions by the implementation to report a violation of an *exception-specification* (15.5.2).

[5] The value could be a multi-byte encoding that can be converted to a wide-character string (21.2).
[6] The value could be a multi-byte encoding that can be converted to a wide-character string (21.2).

18.6.2.2.1 `bad_exception` constructor

`bad_exception();`

Effects: Constructs an object of class `bad_exception`.

18.6.2.2.2 `bad_exception::what`

`const char* const what() const;`

Returns: An implementation-defined value.[7]

4. Other Issues

One of the justifications for the `exception`-based hierarchy in Clause 19 was to allow a C++ program to catch all exceptions:

```
int main()
{
  try {
    // do some work
  } catch (std::exception& err) {
    // ...
    cerr << err.what() << endl;
    return 1;
  }
  return 0;
}
```

The idea here is to avoid the difficulties of §15.3, ¶7 and §18.6.1. Together, these subclauses define the behavior of a C++ program that fails to catch an exception.[8]

Unfortunately, this does not work. In general, there is no guarantee that all exceptions will be derived from a common base class. Therefore, a conscientious C++ program will have to use a "catch-all" handler:

```
int main()
{
  try {
    // do some work
  } catch (...) {
    cerr << "exception!" << endl;
    return 1;
  }
  return 0;
}
```

It's unfortunate that the stack unwinding in §15.3 is left implementation-defined. If a call to `terminate()` always unwound the stack, the program could catch all exceptions another way:

```
int main()
{
  set_terminate(my_function);
  // do some work
  return 0;
}
```

[7] The value could be a multi-byte encoding that can be converted to a wide-character string (21.2).
[8] §15.3 says an uncaught exception results in a call to `terminate()`. §18.6.1 says the default behavior of `terminate()` is `exit()`. §15.3 also says that whether the stack is unwound (and automatic objects destroyed) is implementation-defined.