Boost.System

Acknowledgment: Chris Kohlhoff designed key aspects of the library

May 8, 2008

Beman Dawes

Boost.System

- Error handling (aka Diagnostics) in Boost now, and accepted for C++0x.
- Other fairly simple functionality that is close to the operating system functionality.

Error Handling Use Cases

- Libraries typically implemented by calling operating system or third-party libraries that report errors by an error code (typically, int)

- Libraries that don't want to define their own exception types for each and every kind of error they encounter

- Libraries that report errors by exceptions.
- Libraries that report errors by error codes.
- Libraries that report errors by both ways.

Error Handling Use Cases (cont)

- Users who need to write portable code
- Users who need to write system specific code
- Users who need to do some of both

Error Reporting Design Cases

- Throwing an exception is the only reasonable way to report an error. Example:

Error Reporting Design Cases

- Throwing an exception is the only reasonable way to report an error. Example: constructor fails, no way to leave object in consistent state.

Error Reporting Design Cases

- Throwing an exception is the only reasonable way to report an error. Example: constructor fails, no way to leave object in consistent state.
- Throwing an exception is always an acceptable way to report an error. Example: Errors are truly exceptional and are usually not dealt with at the point of occurrence

Error Reporting Design Cases

- Returning an error code is the only reasonable way to report an error.
- Example:

Error Reporting Design Cases

- Returning an error code is the only reasonable way to report an error.

  Example: A function whose sole job is to report exactly why an error occurred.

Error Reporting Design Cases

- Throwing an exception or returning an error code are both perfectly reasonable.

  Example:

Error Reporting Design Cases

- Throwing an exception or returning an error code are both perfectly reasonable.

  Example: A function used in some contexts where errors are exceptional and other contexts where errors happen regularly and/or must be dealt with at the point of occurrence.

  Symptom: “My code is so littered with try/catch blocks it is unreadable and unmaintainable.”

Error Reporting Design Cases

- Libraries that need their own unique error codes in addition system error codes.

Boost.System Error Reporting Goals

- Support all of the forgoing design cases.
- Relatively light weight; design target was an object containing only and int and a pointer.
- Portable code should require no more user effort than system-specific code, and visa versa.

Class system_error

```
class system_error : public runtime_error {
public:
  system_error(error_code ec, const string& what_arg);
  system_error(error_code ec);
  system_error(int ev, const error_category& ecat, const string& what_arg);
  system_error(int ev, const error_category& ecat);

  const char* what() const throw();

  const error_code& code() const throw();
};
```

Class error_code

```
class error_code {
public:
  // 19.4.2.2 constructors:
  error_code();
  error_code(int val, const error_category& cat);

// 19.4.2.3 modifiers:
  void assign(int val, const error_category& cat);
  void clear();

  // 19.4.2.4 observers:

  int value() const;
  const error_category& category() const;
  error_condition default_error_condition() const;

  string message() const;
  explicit operator bool() const;

private:
  int m_val;                    // exposition only
  const error_category* m_cat; // exposition only
};
```

Class error_condition

```
class error_condition {
public:
  // 19.4.2.2 constructors:
  error_condition();
  error_condition(int val, const error_category& cat);

// 19.4.2.3 modifiers:
  void assign(int val, const error_category& cat);
  void clear();

  // 19.4.2.4 observers:

  int value() const;
  const error_category& category() const;


  string message() const;
  explicit operator bool() const;

private:
  int m_val;                    // exposition only
  const error_category* m_cat; // exposition only
};
```

Class error_category

```
class error_category {
public:
  virtual ~error_category();
  error_category(const error_category&) = delete;
  error_category& operator=(const error_category&) = delete;
  virtual const char* name() const = 0;
  virtual error_condition default_error_condition(int ev) const;
  virtual bool equivalent(int code, const error_condition& condition) const;
  virtual bool equivalent(const error_code& code, int condition) const;
  virtual string message(int ev) const = 0;
  bool operator==(const error_category& rhs) const;
  bool operator!=(const error_category& rhs) const;
  bool operator<(const error_category& rhs) const;
};

extern const error_category generic_category;  // aka posix_category
extern const error_category system_category;
```

Generic error values

```
namespace posix_error  // in C++0x, enum class generic_error
{
  enum posix_errno
  {
    success = 0,
    address_family_not_supported = EAFNOSUPPORT,
    address_in_use = EADDRINUSE,
    address_not_available = EADDRNOTAVAIL,
    already_connected = EISCONN,
    argument_list_too_long = E2BIG,
    argument_out_of_domain = EDOM,
    bad_address = EFAULT,
    bad_file_descriptor = EBADF,
    ...
  };
}
```

Windows error values

```
namespace windows_error
  {
    enum windows_error_code
    {
      // These names and values are based on Windows winerror.h
      invalid_function = ERROR_INVALID_FUNCTION,
      file_not_found = ERROR_FILE_NOT_FOUND,
      path_not_found = ERROR_PATH_NOT_FOUND,
      too_many_open_files = ERROR_TOO_MANY_OPEN_FILES,
      access_denied = ERROR_ACCESS_DENIED,
      invalid_handle = ERROR_INVALID_HANDLE,
      arena_trashed = ERROR_ARENA_TRASHED,
      not_enough_memory = ERROR_NOT_ENOUGH_MEMORY,
    ...
  };
}
```

Have it your way

```
somewhere.hpp
-------------

extern error_code* throw_on_error;        // statically initialized tag
extern thread_local error_code& errcode;  // initialized before use

filesystem.hpp
--------------

class filesystem_error: public system_error {...};

bool create_directory(const path& dp, error_code& ec = *throw_on_error);

error_code create_hard_link(const path& to,
                            const path& from,
                            error_code& ec = *throw_on_error);
```

Have it your way; throw on error

```
create_hard_link(“foo”, “bar”);
```

Have it your way; detect error

```
if (create_hard_link(“foo”, “bar”, errcode)) // if fails
{
  copy_file(“foo”, “bar”);
}
```

Have it your way; system specific

```
if (create_hard_link(“foo”, “bar”, errcode)) // if fails
{
  if (errcode == windows_error::access_denied)
  {
    enable_windows_workaround();
  }
  else
  {
    copy_file(“foo”, “bar”);
  }
}
```

Have it your way; library specific

```
if (create_hard_link(“foo”, “bar”, errcode)) // if fails
{
  if (errcode == boost_error::developers_asleep)
  {
    send_email_to_boost_list();
  }
  else
  {
    copy_file(“foo”, “bar”);
  }
}
```

Have it your way; portable

```
if (create_hard_link(“foo”, “bar”, errcode)) // if fails
{
  if (errcode == generic_error::permission_denied)
  {
    enable_permission_workaround();
  }
  else
  {
    copy_file(“foo”, “bar”);
  }
}
```

Have it your way; catch exceptions derived from system_error

```
try
{
 some_function();  // uses Boost.Filesystem
}

catch (const filesystem_error& ex)
{
  if (ex.code() == windows_error::access_denied)
  {
    enable_windows_workaround();
  }
  else if (ex.code() == generic_error::permission_denied)
  {
    enable_general_workaround();
  }
  else throw;
}
```

----The End ----

User-defined error codes

- User defines their own category, by deriving from boost::system::error_category.
- User defines their own error value enum.
- User writes a couple of very simple helper functions.