C++11
An error code of your very own

Ralph McArdell

ACCU London

26th September 2017

The Plan

- Our own error code values
- Review of C++11 error code support: std::error_code and friends
- Integrating our own error values with std::error_code

Before we begin...

- The topic was taken from parts of an article on C++11+ exception support under review for publication in the ACCU Overload magazine
- I added this use case to the article after reading about it in Andrzej Krzemieński's 'Your own error code' blog post: https://akrzemi1.wordpress.com/2017/07/12/your-own-error-code/
- Full example code for this talk and the other topics covered in the article can be found at: https://github.com/ralph-mcardell/article-cxx11-exception-support-examples

Our own error codes:
What are we talking about here?

- Error values expressed as integer values
- Problems with multiple sets of error values which have overlapping values for different errors
- For the purposes used here they should be enumerated types:
  - can convert #define macro values or groups of const / constexpr integer values to enum / enum class types.

Our own error codes: Example

- In file appengine_error.h:

```
namespace the_game
{ enum class appengine_error
  { no_object_index   = 100
  , no_renderer
  , null_draw_action   = 200
  , bad_draw_context = 300
  , bad_game_object
  , null_player            = 400
  };
}
```

- In file renderer_error.h:

```
namespace the_game
{ enum class renderer_error
  { game_dimension_too_small = 100
  , game_dimension_bad_range
  , board_too_small                  = 200
  , board_bad_range
  , game_dimension_bad
  , board_not_square                 = 300
  , bad_region                       = 400
  , cell_coordinate_bad              = 500
  , new_state_invalid
  , prev_state_invalid
  };
}
```

C++11 error code support: references

- ‘The C++ Standard Library, second edition’ by Nicolai M. Josuttis
- cppreference.com, http://en.cppreference.com
- n3337, post C++11 Working Draft, Standard for Programming Language
  C++

C++11 error code support: of interest for this talk

- <system_error>
- std::error_code
- std::error_category
- std::make_error_code
- std::is_error_code_enum

C++11 error code support: additional

- std::system_error
- std::error_condition
- std::make_error_condition
- std::is_error_condition_enum

C++11 error code support: provided error categories

- const error_category& std::generic_category() noexcept
  - portable POSIX errno error conditions
- const error_category& std::system_category() noexcept
  - errors reported by the operating system
- const error_category& std::iostream_category()
  - IOStream error codes reported via std::ios_base::failure (which since C++11 is derived from std::system_error)
- const error_category& std::future_category() noexcept
  - future & promise errors provided by std::future_error

C++11 error code support:
provided error value enums

- std::errc
  - portable error condition values
    corresponding to POSIX error codes

- std::io_errc
  - error codes reported by IOStreams
    via std::ios_base::failure

- std::future_errc
  - error codes reported by std::future_error

Integrating our own error values with std::error_code:

error values, categories and codes

- Error value – an integer value representing an error in a specific domain: member of a set of error values
- Error category – a set of error values for a specific domain or sub-system
- Error code – an {error value, error category} pair

  [note: similarly an error condition is a
   {portable error value, error category} pair]

Integrating our own error values with std::error_code:

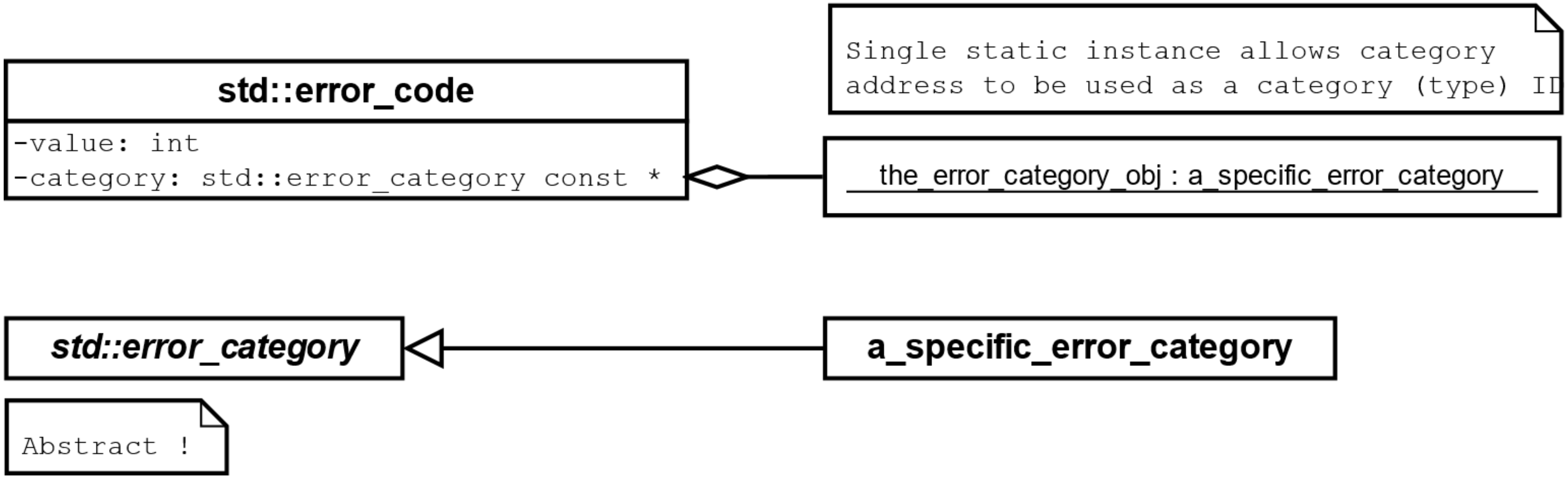
requirements on our error value enum type

- All underlying values will correctly convert to int
- 0 (zero) must be reserved to mean 'OK, no error'

  - Even if our error value types do not provide an OK enumeration value of 0 explicitly so long as a value of 0 is not reserved for an error value then we can create a zero valued instance of the error enum, for example:

```
the_game::appengine_error ok_code_zero_value{};
```

Integrating our own error values with std::error_code:
std::error_code std::error_category relationship


Single static instance allows category address to be used as a category (type) ID
std::error_code
-value: int
-category: std::error_category const *
the_error_category_obj : a_specific_error_category
std::error_category
a_specific_error_category
Abstract !

Integrating our own error values with std::error_code: std::error_category details

```
class error_category
{
public:
        virtual ~error_category() noexcept;

        error_category(const error_category&) = delete;
        error_category& operator=(const error_category&) = delete;

        virtual const char* name() const noexcept = 0;
        virtual string message(int ev) const = 0;

        virtual error_condition default_error_condition(int ev) const noexcept;
        virtual bool equivalent(int code, const error_condition& condition) const noexcept;
        virtual bool equivalent(const error_code& code, int condition) const noexcept;

        bool operator==(const error_category& rhs) const noexcept;
        bool operator!=(const error_category& rhs) const noexcept;
        bool operator<(const error_category& rhs) const noexcept;
};
```

Integrating our own error values with std::error_code:

std::error_code details

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

        template <class ErrorCodeEnum>
        error_code(ErrorCodeEnum e) noexcept;

        template <class ErrorCodeEnum>
        error_code& operator=(ErrorCodeEnum e) noexcept;

        int value() const noexcept;
        const error_category& category() const noexcept;
        string message() const;
        explicit operator bool() const noexcept;

        void assign(int val, const error_category& cat) noexcept;
        void clear() noexcept;
        error_condition default_error_condition() const noexcept;
private:
        int val_; // exposition only
        const error_category* cat_; // exposition only
};
```

Integrating our own error values with std::error_code:

std::error_code details – non-member functions

```
error_code make_error_code(errc e) noexcept;

template <class charT, class traits>
basic_ostream<charT,traits>& operator<<
  ( basic_ostream<charT,traits>& os, const error_code& ec );


bool operator<(const error_code& lhs, const error_code& rhs)  noexcept;
```

Integrating our own error values with std::error_code:

Implementing a custom std::error_category specialisation

- Create sub-class specialisation of std::error_category
- Override the name pure virtual function to return a literal name for the category (const char * to C-string)
- Override the message pure virtual function to return a std::string message for each error value in the category set – don’t forget a default for unrecognised values
- The whole thing can be placed in a single implementation (.cpp) file together with the appropriate make_error_code overload definition as this function is the only point at which the custom error category will be used

Integrating our own error values with std::error_code:

custom std::error_category specialisation example

- In file appengine_error.cpp:

```
namespace
{
  struct appengine_error_category
  : std::error_category
  {
    const char* name()
      const noexcept override;

    std::string message(int ev)
      const override;
  };

  const char*
  appengine_error_category::name()
    const noexcept
  {
    return "app-engine";
  }
```

```
 std::string
 appengine_error_category::message(int ev)  const
 {
   using the_game::appengine_error;
   switch(static_cast<appengine_error>(ev))
   {
   case appengine_error::no_object_index:
      return "No object index";

   case appengine_error::no_renderer:
      return "No renderer currently set";

                 ...

   default:
      return "?? unrecognised error ??";
   }
 }

}
```

Integrating our own error values with std::error_code:

Adding an overload for make_error_code

- Overloads of make_error_code are used to convert a passed error enum value of specific error value enum types to std::error_code values
- They should be placed in the same namespace as the error enum type whose values they convert
- Their *declarations* need to be available whenever an error enum value is to be converted to a std::error_code value – so place in same header as the error enum type definition
- Their *definitions* need access to the custom error category type – so place in same implementation file

Integrating our own error values with std::error_code:
Adding an overload for make_error_code example

- In appengine_error.h after appengine_error definition:

```
std::error_code make_error_code(appengine_error e);
```

- In file appengine_error.cpp after appengine_error_category definition:

```
namespace the_game
{ std::error_code make_error_code(appengine_error e)
  {
    static const appengine_error_category the_err_cat_obj;
    return { static_cast<int>(e), the_err_cat_obj };
  }
}
```

Integrating our own error values with std::error_code:
Adding a specialisation for std::is_error_code_enum

- For types that are eligible for automatic conversion to std::error_code the std::is_error_code_enum struct template should be specialised to provide a true value for the value member
- Specialisations should be placed in namespace std – one of the few occasions application code can add to std
- Like the declaration of the make_error_code overload the specialisation is required where ever enum error values need to be automatically converted to std::error_code objects so should also be placed in the same header file as the error value enum definition
- The definition can simply inherit from std::true_type and have an empty body

Integrating our own error values with std::error_code: Adding a specialisation for std::is_error_code_enum example

- In appengine_error.h after closing of the_game namespace:

```
namespace std
{
    using the_game::appengine_error;

    template <>
    struct is_error_code_enum<appengine_error> : true_type
    {};
}
```

Integrating our own error values with std::error_code: Producing std::error_code objects from custom error values

- Ideally the API interface should deal only in std::error_code values and not mention any domain specific error values at all

- If so then the API header needs to include the <system_error> header for std::error_code but no domain specific error value defining headers

- The API implementation will need to include both <system_error> and headers for any domain specific error value types that are used

- std::system_error exception objects can be created directly from domain specific error enum values

Integrating our own error values with std::error_code:

Producing std::error_code objects from custom error values: example (interface - appengine)

- In file the_game_api.h:

```
# include <system_error>
# include <new> // for std::nothrow

namespace the_game
{
  class appengine
  {
    std::unique_ptr<renderer> rp_;
public:
  std::error_code take_renderer
    (std::unique_ptr<renderer> && rp) noexcept;
  std::error_code update_game_board
                (std::nothrow_t) noexcept;
  void update_game_board();
};

appengine & get_appengine();
}
```

Integrating our own error values with std::error_code: Producing std::error_code objects from custom error values: example (interface - renderers)

- Also in file the_game_api.h:

```
namespace the_game
{
  struct renderer
  {
    virtual int min_dimension() const = 0;
    virtual int max_dimension() const = 0;
  };
  struct oops_renderer : renderer
  {
    int min_dimension() const override;
    int max_dimension() const override;
  };
```

```
struct fine_renderer : renderer
  {
    int min_dimension() const override;
    int max_dimension() const override;
  };
}
```

Integrating our own error values with std::error_code:

Producing std::error_code objects from custom error values: example (implementation - appengine)

- In file the_game_api.cpp:

```
#include "the_game_api.h"
#include "renderer_error.h"
#include "appengine_error.h"
#include <system_error>
#include <memory>
namespace the_game
{ std::error_code appengine::take_renderer
    ( std::unique_ptr<renderer> && rp ) noexcept
  {  auto ec
           { check_dimensions
             ( rp->min_dimension()
             , rp->max_dimension(
             ) };
    if ( !ec )
      rp_ = std::move(rp);
    return ec;
  }
```

```
  std::error_code appengine::update_game_board
    ( std::nothrow_t ) noexcept
   { return rp_ ? appengine_error{}
                : appengine_error::no_renderer;
   }

  void appengine::update_game_board()
   {
     if ( !rp_ )
       throw std::system_error
             ( appengine_error::no_renderer );
   }
}
```

Integrating our own error values with std::error_code:

Producing std::error_code objects from custom error values: example (implementation - functions)

- Also in file the_game_api.cpp:

```
namespace the_game
{
  appengine & get_appengine()
  {
    static appengine the_appengine;
    return the_appengine;
  }

  std::error_code check_dimensions
    ( int dim_min, int dim_max ) noexcept
  {
    if ( dim_min < 3 )
    {  return renderer_error::game_dimension_too_small;
    }
    if ( dim_max < dim_min )
    {  return renderer_error::game_dimension_bad_range;
    }
    return {};
  }
}
```

Integrating our own error values with std::error_code:

Producing std::error_code objects from custom error values: example (implementation - renderers)

- Also in file the_game_api.cpp:

```
namespace the_game
{
 int oops_renderer::min_dimension()
   const
 {
   return 5;
 }

 int oops_renderer::max_dimension()
   const
 {
   return 3;
 }

 int fine_renderer::min_dimension()
   const
 {
   return 3;
 }

 int fine_renderer::max_dimension()
   const
 {
   return 5;
 }
}
```

Integrating our own error values with std::error_code: Consuming std::error_code objects

- Assuming an API interface only deals in std::error_code values then code using the API only need include the API interface header and <system_error> (which should be included by the API interface header!)

- In particular using code need know nothing about the underlying specific error value enumeration types

- Returned std::error_code values allow access to the specific error value, category name and error value's message

- std::system_error exceptions can be caught explicitly to gain access to their contained std::error_code value for access to greater detail than may be present in the std::exception::what() message string

Integrating our own error values with std::error_code: Consuming std::error_code objects: example part #1

- In file cxx11_custom_error_code_example.cpp :

```
#include "custom_error_code_bits/the_game_api.h"
#include <system_error>
#include <iostream>
#include <string>
#include <new>

// Helper to log bad error code return values:
void log_bad_status_codes( std::error_code ec )
{
  if ( ec )
  {
    std::clog << ec << " " << ec.message() << "\n";
  }
}
```

Integrating our own error values with std::error_code:
Consuming std::error_code objects: example part #2

- Also in file cxx11_custom_error_code_example.cpp :

```
int main()
{
  auto & engine{ the_game::get_appengine() };
  // Should fail as setting renderer supporting invalid dimension range
  std::unique_ptr<the_game::renderer>
    rend{new the_game::oops_renderer};
  log_bad_status_codes( engine.take_renderer(std::move(rend)) );
  // Should fail as no renderer successfully set to draw board
  // a) non-throwing overload:
  log_bad_status_codes( engine.update_game_board(std::nothrow) );
```

Integrating our own error values with std::error_code: Consuming std::error_code objects: example part #3

- Also in file cxx11_custom_error_code_example.cpp :

```
// b) throwing overload:
  try
  {
    engine.update_game_board();
  }
  catch ( std::exception & e )
  {
     std::cerr << "Caught exception: " << e.what() << "\n";
  }

  // OK - nothing to report, this renderer is fine and dandy
  rend.reset( new the_game::fine_renderer );
  log_bad_status_codes( engine.take_renderer( std::move(rend)) );

  // OK - now have renderer to render board updates
  log_bad_status_codes( engine.update_game_board(std::nothrow) );
} // end of main
```

Integrating our own error values with std::error_code:

Consuming std::error_code objects: example part #4 – execution output

- When build and run the output should look like this:

```
renderer:101 Reported max. supported game grid less than the min.
app-engine:101 No renderer currently set
Caught exception: No renderer currently set
```