Document number:	DXXXXR0
Date:	2015-12-19
Revises:	N4109/N4015
Project:	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience:	Library Evolution Working Group
Reply-to:	Vicente J. Botet Escriba <vicente.botet@wanadoo.fr>

Refactoring FileSystem TS using `expected` or `status_value`

This paper describes some alternatives to the design of the FileSystem TS N4099 making use of the proposed `std::experimental::expected` N4109 or `std::experimental::status_value` N4233.

FileSystem TS adaptation alternatives

If `std::experimental::expected` is adopted, it seems natural to make use of it in the standard library, and in particular in a new version of the FileSystem TS N4099. This section present some alternative design, but in no way this is intended as a real proposal. We can see this paper as an appendix to N4099 as an study of application of `std::experimental::expected` to a standard library.

The two major concerns of this study are

- the replacement of the out parameter `error_code` and
- how to manage the conflicting names after the first refactoring

Replace out parameter `error_code`

As described in P0157R0 the File System TS make use of an out of bound error reporting mechanism. This section try to see how the use of `expected` can be used to refactor the current interface. The idea is to return an `expected<T, error_code>` or a `status_value<error_code, T>` where there was an `error_code` out parameter.

Function returning a value type

These are the kind of functions that are best adapted to the semantic of `expected` as the current function return not-a-value when `ec` is present.

Current

```
path canonical(const path& p, error_code& ec);
path canonical(const path& p, const path& base, error_code& ec);
```

Expected

```
expected<path, error_code> canonical(const path& p, const path& base = current_path(
```

Function returning void

As these functions don't return nothing, we could already make use of the result type to return the `error_code`. For coherency purposes we can also use `expected<void, error_code>`.

Current

```
void copy(const path& from, const path& to, error_code& ec) noexcept;
void copy(const path& from, const path& to, copy_options options, error_code& ec) nc
```

Expected

```
expecteds<void,error_code> copy(const path& from, const path& to,
                                copy_options options =  copy_options::none) nc
```

Alternatively

```
error_code copy(const path& from, const path& to,
                                copy_options options =  copy_options::none) nc
```

Note that removing the `error_code` parameter allows to have a single function. The same could be achieved if instead of been the last parameter, the FS was decided to put in at fist parameter.

Function returning a `bool`

These functions returns already a status. When the result is false, it can be because an error occurred or because the condition is really false. This makes these kind of functions almost a tri-state function. makes is redundant with the error*code as the error*code is in reality a status, as we can check if it is ok. We could already make use of the result type to return the `error_code` . For coherency purposes we can also use `expected<void, error_code>` .

Current

```
bool equivalent(const path& p1, const path& p2, error_code& ec) noexcept;
```

Usage

```
//...
error_code ec;
if ( equivalent(p1, p2, ec) ) {
    // equivalent case ...
} else if (! ec)
    // not equivalent case ...
} else {
    // error case - make use of ec
}
```

Expected

```
expected<bool, error_code> equivalent(const path& p1, const path& p2) noexcept;
```

Usage

```
//...
auto ex = equivalent(p1, p2);
if ( ! ex )
{
    // error case - make use of ex.error()
}
else if (*ex)
{
    // equivalent case ...
}
else
{
    // not equivalent case ...
}
```

Status and Optional Value

```
status_value<error_code, bool> equivalent(const path& p1, const path& p2) noexcept;
```

Usage

```
//...
auto res = equivalent(p1, p2);
if ( res.status() ) {
 // success case ...
} else {
    // make use of ex.error()
}
```

Function returning a `file_status`

Functions returning the `file_status` are a specific and a more complex case as these functions return already a status. The wording do a mapping from the `error_code` s to the `file_status::file_type` and consider that only some `error_code` s represent an error and associate it to `file_type::none`. Only when the status is `file_type::none` the `error_code` is really significant.

`expected<T, Error>` is designed to help working with error cases.

It would be interesting in separating what is the file status information from what is the file status error.

Two approaches that depend on whether it is important or not to know the concrete reasons error codes

that mapped to `file_type::unknown`, `file_type::not_found`.

- It is not important:
  - The error codes are needed only to refine the error case so we can add it only.
  - The result type can be `expected<file_status, error_code>`.
  - The result type can be `status_value<error_code, file_status>`.
- It is important:
  - The error codes are needed for 3 specific `file_type` values, so we need both.
  - The result type can be `status_value<error_code, file_status>`.
  - The result type can be a struct that group both `file_status_error = pair<file_status, error_code>`.

Note that as the status overload that throws loss the error codes for `file_type::unknown` and `file_type::not_found`, the non-local interface transport less information.

Current

```
file_status status(const path& p, error_code& ec) noexcept;
```

Usage

- Local

```
//...
error_code ec;
auto fst = fs::status(p, ec);
if ( fs::status_known(fst) ) {
 // success case ...
} else {
    // do something
    // make use of ec
   throw filesystem_error("message", p, ec);
}
```

- Non-local

```
//...
auto fst = fs::status(p);
// success case
// ...
```

- Error propagated

```
//...
auto fst = fs::status(p);
// success case
// ...
```

Expected

```
expected<file_status, error_code> status(const path& p) noexcept;
```

Usage

- Local

```
//...
auto efst = fs::status(p);
if ( efst ) {
 // success case ...
} else {
    // do something
    // make use of efst.error()
   throw filesystem_error("message", p, efst.error());
}
```

- Non-local

```
//...
auto fst =  fs::status(p).value();
// success case
// ...
```

- Error propagated

```
//...
return fs::status(p).bind([](file_status const& fs) {
 // success case ...
});
```

Status and Optional Value

```
status_value<error_code, file_status> status(const path& p) noexcept;
```

Usage

```
//...
auto efst = fs::status(p);
if ( fs::status_known(efst.value()) ) {
 // success case ...
 // we can also make use of efst.status()

} else {
    // do something
    // make use of efst.error()
   throw filesystem_error("message", p, efst.error());
}
```

Refactoring `file_status` and `expected` using sum types and pattern matching

`file_status` has two attributes of type `file_type` and `perms`. The `file_type` is the one that contains at the same level file type information and file status information.

Once we will have sum types on the language, we could define a `file_status` as a sum type of the success cases

```
using file_status = not_found | unknown | known(file_type);

expected<file_status, error_code> status(path const&);
```

Note that we are mixing variants and symbolic enums.

Given the adapted definition of `expected`

```
template <class T, class E>
using expected = success(T) | failure(E) {};
```

`expected<_,EC>` is an Error Monad.

Then we could also use pattern matching to get

```
inspect (fs::status(p)) {
    when success(known(ft)): ...
    when success(unknown): ...
    when success(not_found): ...
    when failure(ec): ...
}
```

Waiting for a language solution, we could use the library solution using variant

```
namesapce file_status {
    struct not_found_t{};
    struct unknown_t{};
    template <class T>
    struct known_t {...};
    using type = variant<not_found_t, unknown_t, known_t<file_stats>>;
}
```

Functions returning a reference / operators

These kind of functions are not well adapted to `expected` until we accept references as template parameter for `expected` . We need yet another monadic abstraction to represent the side effect on advancing the iterator.

If we admit that the function can be non-const, we don't need to return anything if we don't want to support chaining.

Current

```
class directory_iterator
{
public:
        directory_iterator&    increment(error_code& ec) noexcept;
```

Usage

```
directory_iterator di;
//...
error_code ec;
di.increment(ec); // return value ignored as it is di&
if (!ec) // success case ...
```

Expected approach

```
class directory_iterator
{
public:
        expected<void, error_code>    increment() noexcept;
```

Usage

```
directory_iterator di;
//...
auto x = di.increment();
if (x) {
    // success case ...
    // use *x;
```

Constuctors

While there is no problem doing this replacement, we cannot return `expected<T, error_code>` from a constructor. We need to introduce factories instead.

Current

```
class directory_iterator
{
public:
        directory_iterator(const path& p, error_code& ec)
```

Usage

```
    error_code ec;
    directory_iterator di{p, ec};
    if (!ec) // success case ...
    else // :( di partialy constructed
```

Expected

```
class directory_iterator
{
    directory_iterator(const path& p)
public:

    friend expected<directory_iterator, error_code> make_directory_iterator(const pa
```

Usage

```
    auto edi = make_directory_iterator di(p);
    if (edi) // success case ...
```

Function name conflict

However, replacing the out `error_code` parameter by a return `expected<T, error_code>` introduces a naming conflict, as the signature of the two functions is ambiguous. For example,

```
class directory_entry
{
public:
    file_status  status() const;
    expected<file_status, error_code> status() const noexcept;
```

There are several alternatives:

- Make use of a suffix `_no_throw` .
- Add an additional tag parameter to disambiguate (e.g. `no_throw`).
- Use a nested namespace for the `expected` based functions
- Make use of the template parameter `Error` to disambiguate both overloads.
- Replace the throw overload by a function returning `expected<T, file_system_error>` .

Add a tag to disambiguate (e.g. `no_throw`)

We can use an additional tag parameter to select the specific behavior.

```
class directory_entry
{
public:
  file_status  status() const;
  expected<file_status, error_code> status(no_throw_t) const noexcept;
  //...
```

The usage of the throwing function doesn't change. For the expected one

```
    auto x = de.status(no_throw);
```

Make use of a suffix (e.g. `_no_throw`)

We can name the functions using a suffix to select the specific behavior.

```
class directory_entry
{
public:
  file_status  status() const;
  expected<file_status, error_code> status_no_throw(no_throw_t) const noexcept;
  //...
```

The usage of the throwing function doesn't change. For the expected one

```
    auto x = de.status_no_throw();
```

Use a nested namespace

This approach can be seen as well adapted for non-member functions

Non-member functions

For non-member function we could also have a nested namespace, e.g. `err`

```
void copy(const path& from, const path& to);
void copy(const path& from, const path& to, error_code& ec) noexcept;
```

```
void copy(const path& from, const path& to);
namespace err {
    expected<void,error_code> copy(const path& from, const path& to) noexcept;
}
```

Classes

However, the member function cannot be nested in an additional namespace. We could duplicate the classes and have one that throws and the other that return expected.

```
namespace std { namespace experimental { namespace filesystem { inline namespace v1

      class directory_iterator
      {
      public:
        typedef directory_entry        value_type;
        typedef ptrdiff_t              difference_type;
        typedef const directory_entry* pointer;
        typedef const directory_entry& reference;
        typedef input_iterator_tag     iterator_category;

        // member functions
        directory_iterator() noexcept;
        explicit directory_iterator(const path& p);
        directory_iterator(const path& p, directory_options options);
        directory_iterator(const directory_iterator& rhs);
        directory_iterator(directory_iterator&& rhs) noexcept;
        ~directory_iterator();

        directory_iterator& operator=(const directory_iterator& rhs);
        directory_iterator& operator=(directory_iterator&& rhs) noexcept;

        const directory_entry& operator*() const;
        const directory_entry* operator->() const;
        directory_iterator&    operator++();

        // other members as required by C++14 §24.1.1 Input iterators
      };

namespace err {

      class directory_iterator
      {
      public:
        typedef directory_entry        value_type;
        typedef ptrdiff_t              difference_type;
```

```
        typedef const directory_entry* pointer;
        typedef const directory_entry& reference;
        typedef input_iterator_tag     iterator_category;

        // factories
        friend expected<directory_iterator, error_code>
            make_directory_iterator(const path& p="", directory_options options=dire

        // constructors/assignments/destructors
        directory_iterator(const directory_iterator& rhs);
        directory_iterator(directory_iterator&& rhs) noexcept;
        ~directory_iterator();

        directory_iterator& operator=(const directory_iterator& rhs);
        directory_iterator& operator=(directory_iterator&& rhs) noexcept;

        // observers
        const directory_entry& operator*() const;
        const directory_entry* operator->() const;

        directory_iterator&    operator++();
        expected<directory_iterator, error_code>    increment(error_code& ec) noexce

        // other members as required by C++14 §24.1.1 Input iterators
      };
}

} } } }  // namespaces std::experimental::filesystem::v1
```

Liabilities

- The classes are duplicated

Make use of the template parameter `Error` to disambiguate both overloads.

```
class directory_entry
{
public:
  template <class Error=filesystem_error>
  file_status  status() const;
  template <>
  file_status  status<filesystem_error>() const;
  template <>
  expected<file_status, error_code> status<error_code>() const noexcept;
  //...
```

The usage of the throwing function doesn't change, but can be made more explicit. For the expected one

```
    auto x = de.status<error_code>(); // return expected<file_status, error_code>
```

```
    auto x = de.status<filesystem_error>(); // throw filesystem_error if a filesyste
    auto x = de.status(); // throw filesystem_error if a filesystem error detected
    // success case
```

```
template <class Error=filesystem_error>
class directory_entry
{
public:
  file_status  status() const;
  //...
};

template <>
class directory_entry<error_code>
{
public:
  expected<file_status, error_code> status() const noexcept;
  //...
};
```

This is not applicable to function template as we cannot partially specialize a function.

Replace the throw overload by `expected<T, filesystem_error>`

```
class directory_entry
{
public:
  template <class Error=filesystem_error>
    requires Same<Error, error_code> or Same<Error, filesystem_error>
  expected<file_status, Error>  status() const;
  //...
```

```
    auto x = de.status(); // returns expected<file_status, filesystem_error>
    if (x) // success case
```

```
    auto x = de.status<error_code>(); // return s expected<file_status, error_code>
    if (x) // success case
```

```
    auto x = de.status<EC>(); // returns expected<file_status, EC>
    if (x) // success case
```

```
namespace common {
    class directory_entry
    {
    public:
      //...
    };
}
namespace err {
    template <class Error=error_code>
        requires Same<Error, error_code> or Same<Error, filesystem_error>
    class directory_entry: public common::directory_entry
    {
    public:
      expected<file_status, Error> status() const noexcept;
      //...
    };
}
using directory_entry = err::directory_entry<filesystem_error>;
```

Acknowledgements

References

- N4099 Beman Dawes, N4099 - Working Draft, Technical Specification — File System

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4099.html

- N4109 Pierre talbot Vicente J. Botet Escriba. N4109, a proposal to add a utility class to represent expected monad (Revision 1), 2014.

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4109.pdf

- N4233 L. Crowl, C. Mysen. N4233, A Class for Status and Optional Value, 2014.

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4233.html

- P0157R0 L. Crowl. P0157R0, Handling Disappointment in C++, 2015.

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0157r0.html