EXTENDED STL

Collections and Iterators

Volume 1

MATTHEW WILSON

Praise for *Extended STL, Volume 1*

"Wilson's menu of STL treatments will no doubt be good eating for generic programming adherents, ardent C programmers just now taking on STL and C++, Java programmers taking a second look at C++, and authors of libraries targeting multiple platforms and languages. Bon appetit!"

—George Frazier, Cadence Design Systems, Inc.

"A thorough treatment of the details and caveats of STL extension."

—Pablo Aguilar, C++ Software Engineer

"This book is not just about extending STL, it's also about extending my thinking in C++."

—Serge Krynine, C++ Software Engineer, RailCorp Australia

"You might not agree 100% with everything Wilson has to say, but as a whole his book is the most valuable, in-depth study of practical STL-like programming."

—Thorsten Ottosen, M.C.S., Boost Contributor

"Wilson is a master lion tamer, persuading multifarious third-party library beasts to jump through STL hoops. He carefully guides the reader through the design considerations, pointing out the pitfalls and making sure you don't get your head bitten off."

—Adi Shavit, Chief Software Architect, EyeTech Co. Ltd

"Wilson's book provides more than enough information to change the angst/uncertainty level of extending STL from 'daunting' to 'doable.' "

—Garth Lancaster, EDI/Automation Manager, Business Systems Group, MBF Australia

"This book will open up your eyes and uncover just how powerful STL's abstractions really are."

—Nevin ":-)" Liber, 19-year veteran of C++

"In the canon of C++ there are very few books that extend the craft. Wilson's work consistently pushes the limits, showing what can and cannot be done, and the tradeoffs involved."

—John O'Halloran, Head of Software Development, Mediaproxy

"Essential concepts and practices to take the working programmer beyond the standard library."

—Greg Peet

"*Extended STL* is not just a book about adapting the STL to fit in with your everyday work, it's also an odyssey through software design and concepts, C++ power techniques, and the perils of real-world software development—in other words, it's a Matthew Wilson book. If you're serious about C++, I think you should read it."

—Björn Karlsson, Principle Architect, ReadSoft; author of *Beyond the C++ Standard Library: An Introduction to Boost*

Table 19.1 Performance of the Raw API and the STL-Adapted Class

Operating System	Using Raw API	Using `readdir_sequence`
Linux (700MHz, 512MB Ubuntu 6.06; GCC 4.0)	2,487 ms	2,091 ms
Win32 (2GHz, 1GB, Windows XP; VC++ 7.1)	16,040 ms	15,790 ms

Another all-around win. Let's look at how to achieve this.

19.1.2 The opendir/readdir API

The **opendir**/**readdir** API consists of four standard and two nonstandard functions and a structure (which has one mandatory field), as shown in Listing 19.3.

Listing 19.3 Types and Functions in the opendir/readdir API

```
struct dirent
{
  char  d_name[]; // Name of the enumerated entry
  . . .           // Other, nonstandard, fields
};
struct DIR;       // Opaque type, representing open directory search
DIR*           opendir(const char* dir);  // Starts search of dir
int            closedir(DIR*);            // Close search
struct dirent* readdir(DIR*);             // Read next entry
void           rewinddir(DIR*);           // Restart search
long int       telldir(DIR*);             // Get current search pos
void           seekdir(DIR*, long int);   // Reposition search
```

The first four functions—`opendir()`, `closedir()`, `readdir()`, and `rewinddir()`—are prescribed by the POSIX standard; `telldir()` and `seekdir()` are extensions. We will focus on the use of the first three functions for the remainder of this chapter. `opendir()` opens a directory search for a given path and, if successful, returns a non-*`NULL`* value of the opaque type `DIR*`, which then identifies the search instance. `readdir()` retrieves each successive entry in the search, returning *`NULL`* when all entries have been enumerated or on error. `closedir()` closes the search and releases any resources allocated by `opendir()` and/or `readdir()`. The value returned by `readdir()` is a pointer to `struct dirent`, which must have at least the field `d_name`, which is either a character buffer or a pointer to a character buffer, containing the name of the entry in the search directory.

19.2 Decomposition of the Longhand Version

Let's now look again at the longhand version (Listing 19.1) and mark the pertinent aspects and problems in code order.

Line **3**: We take a mutable copy of the workplace directory so that we may, if necessary, append a trailing slash in lines 13–16, so that when we need to concatenate it with each entry for passing the full path to `stat()` (line 30), it will be validly formed.

Line **4**: Declare an instance of `std::vector<std::string>`, to which we will add the directory entries as they're enumerated.

Lines **5–9**: Initiate a search, using `opendir()`, and throw a suitable exception if it fails.

Line **13**: If `getWorkplaceDirectory()` returns an empty string, this line will access an element at `size_type(0)−1`, which is likely to be *`0xFFFFFFFF`* or some equally huge value, depending on the size of `size_type`. Whatever its actual value, it will most certainly result in an access violation and a crash. To make this line safe, we'd have to also test `searchDir.size()` against *`0`*. I don't know about you, but this is the kind of thing that gives me a toothache when writing application code.

Lines **19–25**: Some versions of `readdir()` include the dots directories—`"."` and `".."`—in their enumeration results, so these lines test for their presence, while being careful not to elide entries that merely begin with one or two dots. (It's not pretty, is it?)

Line **29**: Concatenate the search directory and the current enumerated entry name to form a full path for the entry. Note that this is a *new* instance of `std::string` for every enumerated entry, including the allocation (and release) of memory to store the result.

Lines **30–32**: Use the `stat()` system call to test whether the given entry is a directory. Passing the entry name alone to `stat()` would work only when the enumeration is conducted in the current directory, which is clearly not the case here.

Line **34**: Add the full path name to the container.

Line **39**: Close the search, which releases any resources associated with it.

Line **41**: Return the filtered results to the caller.

As with the **glob** API in the previous chapter, the longhand form is a combination of verbosity, inconvenience, inefficiency, and subtle bugs. Ensuring a trailing slash on `searchDir`, manually testing for dots directories, and having to call `c_str()` on string instances are inconvenient and lead to verbose code. Having to call `stat()` itself to filter out directories is also something you would ideally leave to a library. The most obvious inefficiency is that a new instance of `entryPath` means at least one visit to the heap for each enumerated entry, but there's also more subtle inefficiency in the fact that the explicit declaration of `dirNames` means that the function return is available only for the named return value optimization (NRVO) and not the return value optimization (RVO). (When compared with the runtime costs of enumerating file system contents, the failure to apply an RVO is likely to bite not a lot. But this situation can crop up in enumerations whose relative runtime costs are considerably different, so I thought I'd bring it to your attention.)

Even if we could endure the other issues, the fact is that this code is not exception safe. Lines 15, 29, and 34 all have the potential to throw an exception, in which case the release of `dir` at line 39 will never happen.

Tip: Mixing C++ (particularly STL) coding with C APIs invariably means exposing many holes through which resources can leak, usually, but not exclusively, as a result of exceptions. Prefer to use ***Façade*** (or ***Wrapper***) classes where available and suitably functionality-rich and efficient.

Absent such classes, consider writing your own. Even if all you gain is RAII and a few defaulted parameters, this is a big improvement in robustness (not to mention that the experience will gain you a concomitant leap in quality of practice).

19.3 `unixstl::readdir_sequence`

Before we define the `readdir_sequence` class, we should look at what the **opendir**/**readdir** API implies about the characteristics of an STL extension. Here are the important features.

- The **opendir**/**readdir** API provides indirect access to the elements of the collection, indicating that an iterator class, rather than a pointer, is required.
- The current directory entry enumeration position may be advanced one element at a time, but only wholly rewound. Neither *bidirectional* nor *random access* iteration (Section 1.3) requirements are fulfilled by a facility for wholesale rewind, so the API will support, at best, the *forward iterator* category.
- Each call to `readdir()` advances the position of the underlying search object. Hence, a given search, initiated by `opendir()`, can support only single-pass iteration, so the iterator category will be *input iterator*.
- The API provides no facility for changing the contents of a directory, so it supports only non-mutating access.
- `readdir()` returns `struct dirent*`, whose only standard-prescribed (and, therefore, portable) member is `d_name`, holding a nul-terminated string of the given entry. This suggests that the collection value type should be `char const*`.
- There is no guarantee that successive calls to `readdir()` return a pointer to the same `struct dirent` instance with overwritten contents or to separate instances. This suggests that the iterator instance should hold a pointer to `struct dirent`, rather than a pointer to its `d_name` member.
- Each call to `opendir()` produces a separate search. Thus, the invocation of `opendir()` should be associated with a call to the `readdir_sequence::begin()` method. It is not yet clear (and actually turns out not to matter) whether the collection calls `opendir()` and passes the resultant `DIR*` to the iterator class or whether the iterator class calls `opendir()` using information provided to it by the collection.
- Similarly, it is not yet clear whether the collection makes the first call to `readdir()`, to commence the search proper, or whether that is done by the iterator.
- The search is advanced by subsequent calls to `readdir()`, which should happen within the iterator's increment operator.
- In order for a single collection instance to support multiple distinct enumerations, the iterator class should be given ownership of the `DIR*` search handle, since it is the increment of the

iterator that brings a search to a close (whether by advancing to the `end()` position or by the iterator instance going out of scope).

Given this analysis, we can stipulate the following: Value type will be `char const*`; element reference category will be transient; iterator category will be input iterator; and collection mutability will be immutable. We can postulate an interface for `readdir_sequence` along the lines of that shown in Listing 19.4.

Listing 19.4 Initial Version of `readdir_sequence`

```
// In namespace unixstl
class readdir_sequence
{
private: // Member Types
  typedef char                            char_type;
public:
  typedef char_type const*                value_type;
  typedef std::basic_string<char_type>    string_type;
  typedef filesystem_traits<char_type>    traits_type;
  typedef readdir_sequence                class_type;
  class                                   const_iterator;
public: // Member Constants
  enum
  {
      includeDots   = 0x0008
    , directories   = 0x0010
    , files         = 0x0020
    , fullPath      = 0x0100
    , absolutePath  = 0x0200
  };
public: // Construction
  template <typename S>
  readdir_sequence(S const& dir, int flags = 0);
public: // Iteration
  const_iterator  begin() const;
  const_iterator  end() const;
public: // Size
  bool empty() const;
public: // Attributes
  string_type const&  get_directory() const; // Always trailing '/'
  int                 get_flags() const;
private: // Implementation
  static int          validate_flags_(int flags);
  static string_type  validate_directory_(char const* directory
                                        , int         flags);
private: // Member Variables
  const int         m_flags;
  const string_type m_directory;
```

```
private: // Not to be implemented
  readdir_sequence(class_type const&);
  class_type& operator =(class_type const&);
};
```

19.3.1 Member Types and Constants

The member types include a `string_type` (which we'll need later), the value type, and a forward declaration of the nested class `const_iterator`. There are two main options when writing collection-specific iterators. They can be defined either, as here, as nested classes whose names correspond to the member type role they're fulfilling or as separate classes, for example, `readdir_sequence_const_iterator`, which are then introduced as member types via `typedef`. A number of factors are involved in deciding between the two, such as whether the collection is a template, the level of (human) tolerance for verbose type names, whether the iterator type can serve for more than one collection (Section 26.8), and so on.

Tip: Define iterator classes that are used for one collection class as nested classes. This reduces namespace pollution and makes a clear point about the relationship of the iterator and its sequence to users.

The constants moderate the collection behavior. `includeDots`, `directories`, and `files` have the same meaning as they did for `glob_sequence`. We'll defer discussion of `fullPath` and `absolutePath` until later in the chapter (Section 19.3.11).

The `traits_type` is defined from `unixstl::filesystem_traits` (Section 16.3), which provides abstraction of various facilities required in the implementation. The member type `char_type`, defined to be `char`, is used throughout the class definition to cater for the possibility that the class may one day be converted to a template in order to work with the wide-character analog of the **opendir**/**readdir** API. (It defines the `wDIR` and `struct wdirent` types and manipulates them using `wopendir()`, `wreaddir()`, and so on.)

Tip: Avoid *DRY SPOT* violations by defining member types to provide a single type from which other types are defined.

`traits_type` is defined to be public even though, by rights, it should be private, because `const_iterator` needs to see it.

19.3.2 Construction

Unlike the hoops incurred with `glob_sequence` (Section 17.3.5, Chapter 18), the public construction methods of `readdir_sequence` comprise a single constructor, made flexible by use of the **c_str_ptr** string access shim (Section 9.3.1), as shown in Listing 19.5.

Listing 19.5 `readdir_sequence` **Constructor Template**

```
class readdir_sequence
{
  . . .
```

```
public: // Construction
  template <typename S>
  readdir_sequence(S const& dir, int flags = 0)
    : m_directory(stlsoft::c_str_ptr(dir))
    , m_flags(validate_flags_(flags))
  {}
  . . .
```

The `validate_flags_()` method is not further discussed here since it performs a service equivalent to the method of the same name in `glob_sequence` (Section 17.3.4).

According to the *Law of the Big Two*, there's no need for a destructor since the only resources associated with an instance are bound up in the `string_type`—the compiler-generated version suffices. Thus, this type could, without any additional input from the author, support the *Assignable* and *CopyConstructible* requirements of the *STL container* concept (C++-03: 23.1;3). However, the copy constructor and copy assignment operator have been proscribed. Why? The reason is that `readdir_sequence`, just like `glob_sequence` (and pretty much any other file system enumeration API), provides only a snapshot of the system at a given point in time. Disallowing copy semantics prevents the user from easily forgetting this fact.

Tip: Consider proscribing operations from your types for conveying information on appropriate use, as well as for classic robustness and correctness reasons, particularly for collection types that provide snapshots of their underlying collections.

19.3.3 Iteration and Size Methods

The iteration methods `begin()` and `end()` return instances of `const_iterator`. There are no mutating forms and no reverse iteration forms, due to the constraints attendant in the non-mutating single-pass nature of the **opendir/readdir** API.

As for the size methods, the characteristics of the **opendir/readdir** API mean that only `empty()` is defined; there is no `size()` method. This may seem like a glaring omission, and in a way it is. But because the API gives an element at a time, the only portable way to implement a `size()` method would be as follows:

```
size_type readdir_sequence::size() const
{
  return std::distance(begin(), end());
}
```

In terms of the method semantics, this implementation is perfectly valid. However, rather than being a constant-time operation—something expected (though not required!) of standard containers (C++-03: 23.1), and generally expected of all STL extension collections—it is likely to be along the lines of $O(n)$ (depending on the implementation of the underlying **opendir/readdir** API). Hence, the syntax and semantics may be the same, but the complexity would be significantly different. It's a Goose Rule thing (Section 10.1.3).