Presentation Conventions

What anyone thinks of me is none of my business.

—Peter Brock

Me fail English? That's unpossible.

—Ralph Wiggum, *The Simpsons*

Most presentation conventions in the book are self-evident, so I'll just touch on those that warrant a little explanation.

Fonts

The fonts and capitalization scheme discriminates between the following types of things in the body text: **API** (e.g., the **glob** API), `code`, *Concept* (e.g., the *shim* concept), `<headername .hpp>`, **Library** (e.g., the **ACE** library), `literal` or `path` (e.g., `"my string"`, `NULL`, `123`, `/usr/include`), *Pattern* (e.g., the *Façade* pattern), *Principle* (e.g., the *Principle of Least Surprise*), **shim** (e.g., the **get_ptr** shim). Code listings use the following conventions:
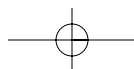
```
// In namespace namespace_name
class class_name
{
  . . . // something that's a given, or has been shown already
public: // Class Section Name, e.g., "Construction"
  class_name()
  {
    this->something_emphasized();
    something_new_or_changed_from_previous_listing();
  }
  . . .
```
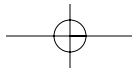
. . . versus . . .

`. . .` denotes previously seen code, boilerplate code, or previously seen or yet-to-be-specified template parameter lists. This is not to be confused with `. . .` , which denotes an ellipsis, as used in variadic function signatures and catchall clauses.

End Iterator Precomputation

To keep the code snippets digestible, I've shown handwritten iterator loops without using end iterator precomputation. In other words, the book shows listings such as:

```
typedef unixstl::readdir_sequence    rds_t;
rds_t   files(".", rds_t::files);

for(rds_t::const_iterator b = files.begin(); b != files.end(); ++b)
{
  std::cout << *b << std::endl;
}
```

rather than what I would normally write:

```
. . .
for(rds_t::const_iterator b = files.begin(), e = files.end(); b != e;
++b)
{
  std::cout << *b << std::endl;
}
```

This is likely to be more efficient in many cases (and will certainly not be less in any). The same can be said of evaluating the size() of a collection once. So, despite the fact that you won't see it anywhere else in this book, I suggest the following:

Tip: Prefer to precompute the end iterator when enumerating iterator ranges. Prefer to pre-compute the size of a collection when using indexing.

Of course, the best way to precompute the endpoint iterator is to use an algorithm, where possible, as in the following:

```
std::copy(files.begin(), files.end()
        , std::ostream_iterator<rds_t::value_type>(std::cout, "\n"));
```

Nested Class Type Qualification

I also take a few shortcuts in qualification of types where the context is unambiguous. For example, rather than writing out the full return type of Fibonacci_sequence::const_ iterator::class_type& for the Fibonacci_sequence::const_iterator:: operator ++() method, I'll just write it as follows:

```
class_type& Fibonacci_sequence::const_iterator::operator ++()
{
  . . .
```

Should you wish to compile code directly from the book, you will need to bear these expediencies in mind. Thankfully, I'm including all the test files on the accompanying CD, so you shouldn't need to go to the effort of transcription from the text.
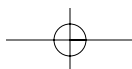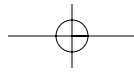
NULL

I use *NULL* for pointers, for two reasons. First, despite anything you might have heard to the contrary, there *is* a strongly typed *NULL* available for C++ (as described in Section 15.1 of *Imperfect C++*). Second, I believe that being all modern and using *0*, merely because *NULL* has no more meaning to the compiler, is just plain nuts. Code is for people first, compilers second.

Template Parameter Names

Template parameters are one- or two-letter capitals, for example, T (type/value-type), S (sequence/string), C (container/character), VP (value policy type), and so on. A few are shown in full, for example, CHOOSE_FIRST_TYPE, but in code they are one or two letters. There are two reasons for this. First, many all-uppercase words are #defined in third-party libraries and application code. Experience such a conflict once and you'll take big steps not to do so again, I promise you! Conversely, we may assume that instances of one (and maybe two) character letters are not #defined. If you find yourself using any library that does #define symbols with one or two letters, throw it away immediately in all good conscience.

The second reason is that it is not valid to create a member type with the same name as a template parameter. In other words, the following is illegal.

```
template <typename iterator>
struct thing
{
  typedef iterator    iterator; // Compile error
};
```

I strongly believe, therefore, that:

```
template <typename I>
struct thing
{
  typedef I           iterator;
};
```

is much clearer than:

```
template <typename Iterator>
struct thing
{
  typedef Iterator    iterator;
};
```

Almost without exception, I immediately use the short template parameter names to define member types, and they are *not* used further throughout the remainder of the template class definition. This is especially significant because it makes obvious the absence of member types, many of whose absence can lie unseen for a long (and, thereby, ultimately vexing) time. By having just a
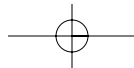
single letter, the class template definition must immediately busy itself defining all these member types, which is a *good thing*. If you don't agree, that's your prerogative. But I assure you that this is the most survivable scheme I've come across.

Member and Namespace-Scope Type Names

Member types are named `xyz_type`, except for established/standard names, such as `iterator`, `pointer`, and so on; local/namespace-scope types are named `xyz_t`.

Calling Conventions

Where they are mentioned, the three common Windows calling conventions are referred to as **cdecl** (equivalent to the calling convention used on UNIX), **fastcall**, and **stdcall**. Unless specified otherwise, all COM methods and all Windows API functions are **stdcall**, and everything else is **cdecl**, though I will always mention the calling convention when it pertains to the discussion. Where a calling convention is indicated, it is using the Microsoft compiler extension keywords `_cdecl`, `_fastcall`, and `_stdcall`; other compilers may use different keywords.

Endpoint Iterators

The C++ standard refers to the iterators returned from `end()` and `rend()`, and their equivalents, as being "past-the-end values." I use the term **endpoint iterators** for clarity and consistency. After all, these functions are not named `past_the_end()` and `rpast_the_end()`. When discussing an iterator reaching the end of its traversal, I will refer to it as having reached the `end()` point, that is, the state at which it will compare equal with the iterator returned by the collection's `end()` method.

Namespace for Standard C Names

I do not place types, function names, and other reserved names from the C standard library in the `std` namespace. This saves space in the code examples, but it also corresponds to my own practice. It's simply never going to happen that `size_t` or `strlen` will be removed from the global namespace, so to write `std::size_t` and `std::strlen` in their stead is gratuitous and distracting. (To me, anyway; other authors may have different views.)

Class Adaptors and Instance Adaptors

Some of the patterns literature uses the terms *Adaptor* and *Object Adaptor* when referring to, respectively, a compile-time adaptation of a type and a runtime adaptation of an instance of a type. Neither term pleases me much; the former is too general, the latter uses the word *object*, which is *way* overused in software development literature. I prefer the terms ***Class Adaptor*** and ***Instance Adaptor***, and that's what I'll be using throughout this book.
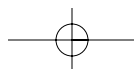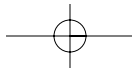
Header File Names

Header files are always mentioned with surrounding angle braces. This is to avoid ambiguity when mentioning the (stupidly named) extensionless standard header files. In other words, *<algorithm>*, rather than the more ambiguous *algorithm*, refers to the header file.