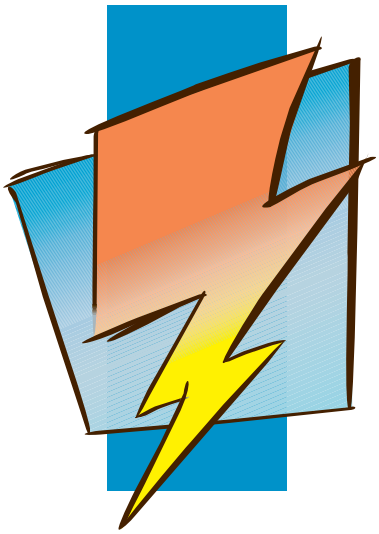

Open Watcom

Open Watcom Standard Template Library (OWSTL)

Design Documentation

Open Watcom Contributors

March 30, 2025

Contents

Chapter 1

Introduction

1.1 Overview

The Open Watcom Standard Template Library (OWSTL) is an implementation of the C++ standard library defined in ISO/IEC 14882. This document describes the design of OWSTL. Each section describes an element of the library and typically includes an overview of the design, design decisions made and the reasoning behind them, problems encountered, and explanations of the solutions to those problems. It is hoped that a peer review of the code and design documentation will be undertaken at some stage and that questions raised or resulting changes made will be documented here.

1.2 Philosophy

OWSTL is written entirely from scratch. It does not, for example, build upon an old HP/SGI code base or any other previous library. When a new element is added to OWSTL the matter should be researched by the author before they commence coding. The commercial compiler Open Watcom is based on, made a name for itself by producing high quality, fast code. The intention is to produce a high performance library to complement that. This means choosing and experimenting with the best algorithms possible and implementing them with care.

OWSTL also attempts to be highly readable to encourage code inspection and study. That will encourage new developers to maintain and improve the library and will give the greatest advantage in the long term.

1.3 Status

OWSTL is currently under development. Many elements of the library have not yet been implemented. The code is mainly templates and currently resides in under the `hdr` project. In the future, non-template classes or functions may be factored out of the template code and be built into the static and dynamic libraries. The existing library code is in `bld/plusplus/cpplib`.

For example, it should be possible to separate the rebalancing algorithms from the red-black tree code as these just manipulate pointers. They don't really need to know the contained type. Reasonably thorough regression tests can be found in `plustest/owstl`. These should be updated in parallel with new functionality or fixes made to the library itself. Some Benchmarks can be found in `bench/owstl`.

1.4 Implementor's Notes

When updating OWSTL, do the following:

- Check out latest source.
- Run the regression tests. If any are broken, fix them or open an issue.
- Update the source.
- Update the regression tests.
- Update this document.
- Update the user documents (e.g., on the Wiki, if applicable).
- Create a pull request.

Chapter 2

Algorithm

2.1 Introduction

The algorithm header (truncated to `algorith` for 8.3 file name compatibility) contains definitions of the algorithms from chapter 25 of the C++ 1998 standard.

2.2 Status

About two thirds of the required algorithms have been implemented. For a list of those remaining, see the Wiki.

2.3 Design Details

Most of the standard algorithms are function templates that operate on iterators to perform some task. Each function template is quickly addressed in the sections that follow. They are generally quite simple and looking directly at the source may provide the best information.

A number of the algorithms come in two forms that use either `operator<` or `operator==` (as appropriate), and in a form that uses a predicate. The predicate form is more general. The non-predicate form can be implemented in terms of the predicate form by using the function objects in `functional`.

In theory, implementing the non-predicate forms in terms of the predicate forms should not entail any abstraction penalty because the compiler should be able to optimize away any overhead due to the function objects. Some investigation was done using Open Watcom v1.5 to find out if that was true. In fact, the compiler was able to produce essentially identical code for the non-predicate functions that were implemented directly as it did for non-predicate functions that were implemented in terms of the predicate functions. However, at the call site, there was some abstraction penalty: the compiler issued a few extra instructions to manipulate the (zero sized) function objects.

These experiments led us to conclude that the non-predicate functions should be implemented directly for short, simple algorithms where the extra overhead might be an issue. For the more complex algorithms,

the non-predicate forms should be implemented in terms of the corresponding predicate forms. The extra overhead of doing so should be insignificant in such cases and the savings in source code (as well as the improved ease of maintenance) would make such an approach desirable.

If the compiler's ability to optimize away the function objects improves, this matter should be revisited.

2.3.1 `*_heap`

The functions `push_heap`, `pop_heap`, `make_heap`, and `sort_heap` support the manipulation of a heap data structure. Currently, only versions using an explicit `operator<` have been implemented. The versions taking a comparison object have yet to be created. Several heap-related helper functions have been implemented in namespace `std::_ow`. These functions are not intended for general use.

There is a compiler bug that prevents the signature of the internal `heapify` function from compiling. This has been worked around by providing the necessary type as an additional template parameter. See the comments in `algorith.mh` for more information.

2.3.2 `remove`, `remove_if`

These functions "remove" the value that compares equal or the element at which the predicate evaluates to true. Because iterators can't be used to access the underlying container the element can't really be removed. These functions instead copy elements from the right (an incremented iterator) over the top of the element that is "removed" and then return an iterator identifying the new end of the sequence. The initial implementation just called the `remove_copy` and `remove_copy_if` functions described below. This would perform unnecessary copies on top of the same object if there are any values at the beginning of the container that aren't to be removed. This could cause a performance hit if the object is large and there are lots of objects that don't need to be removed, therefore these functions were re-written to be independent of the `*_copy` versions and perform a check for this condition.

2.3.3 `remove_copy`, `remove_copy_if`

This makes a copy of the elements that don't compare equal, or when the predicate is false, starting at the location given by Output. It is a simple while loop over the input iterator first to last, either just skipping the element or copying it to the output.

2.3.4 `unique`

For C++ 1998 and C++ 2003 there is an open library issue regarding the behavior of `unique` when non-equivalence relations are used. The standard says that the predicate should be applied in the opposite order of one's intuition. In particular: `pred(*i, *(i-1))`. This means the predicate compares an item with its previous item.

The resolution of the open issue suggests that non-equivalence relations should not be permitted. In any case, the standard should apply the predicate between an item and the next item: `pred(*(i-1), *i)`.

The Open Watcom implementation follows the proposed resolution and thus deliberately violates the standard. Most (all?) other implementations do the same.

2.3.5 find_first_of

There are two versions of this, one that uses operator== and one that uses a binary predicate. There is a simple nested loop to compare each element with each element indexed by the second iterator range.

2.3.6 find_end

There are two versions of this, one that uses operator== and one that uses a binary predicate. The main loop executes two other loops. The first loop finds an input1 element that matches the first input2 element. When a match is found the second loop then checks to see if it is complete match for the subsequence. If it is, the position the subsequence started is noted and the main loop is iterated as there may be another match later on. Note this can't search for the substring backwards as the iterators are ForwardIterators.

2.3.7 random_shuffle

The random_shuffle template with two arguments has been implemented using the C library function rand. However, the 1998/2003 standard is unclear about the source of random numbers that random_shuffle should use. There is an open library issue about this with the C++ standardization group. See http://anubis.dkuug.dk/JTC1/SC22/WG21/docs/lwg-active.html, item #395. The proposed resolution is to allow the implementation to use rand without requiring it to do so (the source of random numbers is proposed to be implementation defined).

The problem with rand in this case is that Open Watcom's implementation of rand is limited to 16 bits of output even on 32-bit platforms. This means that random_shuffle will malfunction on sequences larger than 32K objects. This is a problem that needs to be resolved. The solution, probably, will be to provide 32-bit random number generators as an option. ⟨⟨⟨ Has it already been done? ⟩⟩⟩ **Note**

2.3.8 sort

The sort template is implemented using the QuickSort algorithm. This was shown to be significantly faster (over twice as fast) than using HeapSort based on the heap functions in this library. This implementation of QuickSort is recursive. Since each recursive call has private state, it is unclear if a non-recursive version would be any faster (at the time of this writing, no performance comparisons between recursive and non-recursive versions have been made). Stack consumption of the recursive implementation should be $O(\log(n))$ on the average, which is not excessive. However, the stack consumption would be $O(n)$ in the worst case, which would be undesirable for large n.

Chapter 3

Deque

3.1 Introduction

The class template `std::deque` provides a random access sequence with amortized $O(1)$ `push_back` and `push_front` operations.

3.2 Status

The basic functionality of `std::deque` has been implemented. This includes the specialized deque operations and deque iterators. However, the more "exotic" vector-like operations (insert and erase in the middle of the sequence) have not yet been implemented. There has been essentially no user feedback.

3.3 Design Details

3.3.1 Overall Structure

This implementation is based on a circular buffer. Like a vector, a deque object allocates more memory than it actually uses. In other words, its capacity may be greater than its size. However, unlike a vector the sequence stored in a deque is allowed to wrap around in the buffer resulting in non-contiguous storage. This means an operation such as `&deq[0] + n` may result in a pointer that is invalid even if `n` is less than the deque's size. This behavior is allowed by the standard. ⟨⟨ Reference? ⟩⟩ **Note**

A deque object maintains two indices. The `head_index` refers to the location in the buffer where the first item is stored. The `tail_index` refers to the location in the buffer just past (after possibly wrapping around) where the last item is stored. When `head_index == tail_index` the deque is empty. To avoid the potential ambiguity of this condition, the buffer is reallocated just before it is full, when `deq_length + 1 == buf_length`, so that the condition `head_index == tail_index` never occurs due to a full buffer. This makes implementing some of the deque operations much easier.

For example, deque iterators are represented using a pointer to the deque object and an index value that marks the iterator's current position in the deque's buffer. If the iterator's index value equals `head_index` this can only mean the iterator is at the beginning of the sequence. It never means that the iterator is just past the end of the sequence. This disambiguation makes implementing `operator<` and the other relational operators on iterator much more straight forward.

The general organization and style of deque's implementation follows that of the other buffered sequences, `std::vector` and `std::string`. This consistency is intentional. It is intended to make the `std::deque` code easier to understand. It also opens up some possibility that all the buffered sequences might one day share code.

3.3.2 Alternative Implementations

In addition to the circular buffer implementation an alternative approach was considered that uses contiguous storage. The idea was to store the deque's contents in the middle of the buffer so that some free space would be available on either end for fast `push_back` and `push_front` operations. If the deque grows to the point where one of the buffer ends is reached, the active contents of the deque might be recentered (if the allocated space was not too large) or completely reallocated (if the allocated space was almost full).

This contiguous storage approach allows deque to be more vector-like and might promote code sharing between deque and vector. For example, a vector would be a special kind of deque in this case. However, at the time of this writing it is unclear how such an implementation would best decide between recentering and reallocation. More analysis is necessary to understand the issues involved.

3.4 Open Watcom Extensions

Because of this implementation's use of a circular buffer it is not difficult to provide `capacity` and `reserve` methods for deque even though the standard does not require them. As with vector, the `reserve` method causes a deque to set aside enough memory so that no additional allocations or internal copies will be needed until at least the reserved size is reached.

Chapter 4

List

4.1 Introduction

4.2 Status

Missing members:

1. Err...need to look through the standard.

Completed members:

1. `explicit list( Allocator const & )`
2. `list( list const & )`
3. `~list( )`
4. `operator=( list const & )`
5. `assign( size_type, value_type const & )`
6. `get_allocator( ) const`
7. `iterator`
8. `const_iterator`
9. `reverse_iterator`
10. `const_reverse_iterator`
11. `begin( )` (+const)
12. `end( )` (+const)

13. `rbegin( )` (+const)
14. `rend( )` (+const)
15. `size( )`
16. `empty( )`
17. `front( )`
18. `back( )`
19. `push_front( value_type const & )`
20. `push_back( value_type const & )`
21. `pop_front( )`
22. `pop_back( )`
23. `insert( iterator, value_type const & )`
24. `erase( iterator )`
25. `erase( iterator, iterator )`
26. `swap( list& )`
27. `clear( )`
28. `remove( value_type const & )`
29. `splice( iterator, list & )`
30. `splice( iterator, list &, iterator )`
31. `splice( iterator list &, iterator, literator )`
32. `reverse( )`
33. `merge( list const & )`

4.3 Design Details

```
template <class Type, class Allocator>
class std::list
```

4.3.1 Description of a Double Linked List

This is a data structure that is made up of nodes, where each node contains the data, a pointer to the next node, and a pointer to the previous node. The overall structure also knows where the first element in the list is and usually the last. Obviously it requires two pointers for every piece of data held in the list, but this allows movement between adjacent nodes in both directions in constant time.

4.3.2 Overview of the Class

The class defines an internal `DoubleLink` structure that only holds forward and backward pointers to itself. It then defines a `Node` structure that inherits from `DoubleLink` and adds to it the space for the real data (of type `value_type`) that is held in the list nodes. This is done so that a special sentinel object can be created that is related to every other node in the list, but that doesn't require space for a `value_type` object. This sentinel is used by the list class to point to the first and last elements in the list. A sentinel is useful in this case (the alternative would just be individual first and last pointers) because it means the insertion and deletion code does not have to check for the special case of editing the list at the beginning and end. The sentinel is initialized pointing to itself and is used as the reference point of the end of the list. When an element is inserted or deleted before the end or at the beginning all the pointer manipulation just falls out in the wash. ⟨⟨⟨ This seems to be a good uses of sentinels. ⟩⟩⟩ **Note**

There are two allocators that need to be rebound for the `Node` and `DoubleLink` types. Two allocators are needed because objects of different types are being allocated: the node allocation allocates nodes (with their contained `value_type`) while the link allocator allocates the sentinel node of type `DoubleLink`.

4.3.3 Inserting Nodes

The work for the functions `push_front`, `push_back` and `insert` is done by the private member `push`.

The member is quite simple. It allocates a `Node` and then tries to make a copy of type in the memory allocated. The usual try-catch wrappings deallocate the memory if the construction was unsuccessful. It then modifies the pointers of the `Node` that was passed, the element before that node, and the new node so that the new node is linked into place just before o. The end of the list is signaled by the sentinel object, so if we are trying to insert before the end o is the sentinel and everything works. If we are trying to insert before the first node, the old node before the first is again the sentinel, so the pointers are all valid and everything works.

4.3.4 Deleting Nodes

4.3.5 Clearing All

Chapter 5

Red-Black Tree

5.1 Introduction

Class template `std::_ow::RedBlackTree` is an implementation of a red-black tree data structure. It is used as a common base for `std::set` and `std::map`. It can be found in `hdr/watcom/_rbtree.mh`. The intention was to allow easy replacement and experimentation with other implementations such as an AVL tree or perhaps some sort of relaxed chromatic tree suited to concurrent systems.

5.2 Status

The majority of the required functionality has been written. Regression tests have been written in parallel, but little user testing and feedback exists.

The missing members are:

1. `reverse_iterator`
2. `const_reverse_iterator`
3. `template<InputIterator> ctor( InputIterator, InputIterator, ... )`
4. `rend( )` and `rend( ) const`
5. `rbegin( )` and `rbegin( ) const`
6. `max_size( )`
7. `erase( iterator first, iterator last )`
8. `swap( RedBlackTree & )`
9. `key_comp( )`
10. `value_comp( )`

11. `find( key_type ) const`
12. `count( )`
13. `equal_range( key_type )` and `equal_range( key_type ) const`
14. Non-member operators and specialized swap algorithm

The completed member are:

1. `iterator`
2. `const_iterator`
3. `ctor( Compare, Allocator )`
4. `cpyctor`
5. `operator=`
6. `dtor`
7. `begin( )` and `begin( ) const`
8. `end( )` and `end( ) const`
9. `empty( )`
10. `size( )`
11. `insert( value_type )`
12. `insert( iterator, value_type )` (see N1780)
13. `erase( iterator )`
14. `erase( key_type cont &)`
15. `clear( )`
16. `find( key_type )`
17. `lower_bound( key_type )` and `lower_bound( key_type ) const`
18. `upper_bound( key_type )` and `upper_bound( key_type ) const`
19. `_Sane( )`
20. Internal tree balancing functions

5.3 Design Details

```
template <class Key, class Compare, class Allocator, class ValueWrapper>
class RedBlackTree
```

The type `Key` is used to index the tree; the functor `Compare` (class with `operator( )` defined) provides ordering to the keys; the class `Allocator` provides the memory allocation; the class `ValueWrapper` defines the type of the objects stored in the tree and provides an `operator( )` that knows how to extract the key from that type. `ValueWrapper` allows the same tree code to apply to sets where the key is the only thing stored and maps where the object stored has a key and a mapped value.

5.3.1 Relationship to `map` and `set`

The templates `std::set` and `std::map` take their base class as a template parameter. They select the appropriate value wrapper and inherit all the functionality. The base currently defaults to `RedBlackTree` which is the only implementation available.

5.3.2 Description of a Red-Black Tree

A red-black tree is an ordered binary tree made up of nodes, where each node can have up to two children. An ordered binary tree orders the nodes so that a left child is less than its parent and a right child is greater. It could be the other way around, and this implementation uses a comparison function and puts the child on the left if `compare( child, parent )` evaluates true. If a node has no children it is a leaf, otherwise it is an internal node. Some implementations only hold the actual data in the leaves and the internal nodes are just placeholders. This implementation has imaginary leaves - null pointers. If a node's child pointer is null then that non-existent child is a leaf, and we hold all the data in the real, existing nodes. Therefore, there is no special leaf node type, just a null pointer if there is no child with data.

A red-black tree adds a color to every node, and defines some rules that mean the tree stays balanced. A tree is balanced if the difference between the largest and smallest depth of a leaf is bounded. The invariants are:

1. Every red node has a black parent/
2. Every route from the root node to a node with 0 or 1 children has the same number of black nodes.
3. Every leaf is black (note this is assumed as leaves don't really exist in this implementation).
4. The root is black.

This data structure has been well covered in the literature, for a more detailed information see: (Prof Lyn Turba, Wellesly College, CS231 Algorithms Handout 21, 2001) (McGill University, Notes for 308-251B, http://www.cs.mcgill.ca/ cs251/, 1997) ⟨⟨⟨ Look into setting up a BibTeX database for proper references and citations. ⟩⟩⟩ **Todo**

5.3.3 Overview of the class

The tree class defines an internal `Node` structure that is made up of the object stored in the tree, node pointers for the parent and left and right children, and the node color. There is an allocator member object, `mMem`, that is rebound to allocate `Node` objects. There are pointers to the root and furthest left and right nodes. These are used to mark were to start the search, and create the `begin` and `end` iterators respectively. The `iterator` and `const_iterator` classes are derived from a common member class. There is an Open Watcom extension method `_Sane( )` that checks the integrity of the data structure. Related to this is an integer `mError` member that is assigned a value if an error is detected when `_Sane` is run. ⟨⟨⟨ This should perhaps be renamed `_Error` or made private and a `_GetError( )` method provided. ⟩⟩⟩ **Todo**

5.3.4 Inserting Elements and Rebalancing

The `insert` method calls `unbalancedInsert` and `insBalTransform`. The loop in `unBalancedInsert` moves from child to child searching for the leaf of the tree where the new item can be inserted, similarly as the `find` algorithm checks for the item. A final check is made at the end of the loop to see if the key already exists. If it does, an iterator to the existing key is returned. Otherwise, a new node is allocated and constructed. The node is linked into the tree at the place found. A try-catch is placed around the construction of the node to deallocate the node again if any exceptions are raised. This is needed to stop a memory leak that could occur because the memory has been allocated, but the exception has stopped the node from being linked into the tree (so it would never get destroyed when the tree is destroyed).

At this point, the tree is a valid binary tree but does not necessarily obey the red-black balance criteria. The new node is painted red so as not to invalidate the black-height rule, but this may introduce a violation of the red-red rule. Function `insBalTransform` is called with a pointer to the newly inserted node to correct this. This is where this implementation of a red-black tree varies from the most common implementations. Usually the balancing procedure is broken down into a series of rotations where a subtree of the tree would appear to be rotated if represented graphically. These rotations can be left or right and the procedure moves up to the parent subtree and is repeated until the violation is removed.

Instead, Open Watcom uses the concept of a transformation. (Alternatives to Two Classic Data Structures, Chris Okaski, 2005?) A sub-branch of the tree is analyzed to see which case it matches and the elements in that branch are then reorganized and recolored in one block of code. Although this isn't wildly different, it was hoped that it would allow a faster algorithm to be created because larger subtrees and special cases could be matched and manipulated in one go, and the code generator may be able to make a better job of optimizing the code because a larger block of manipulating instructions would be together. Whether this was a good decision will be born out in time. **Todo**

⟨⟨⟨ Explain why the insert methods are currently inline - compiler bug - what exactly was the problem? ⟩⟩⟩

5.3.5 Deleting Elements

Deletion is a bit more complicated than insertion. The main method that gets called is `erase( iterator )`. I did hope it may be possible to rewrite this in a way that is easier to understand. There are two main cases:

- The node to be removed has both children.
- The node has one or more null children (i.e. has 0 or 1 real child, in other words 1 or 2 leaves) - I've called this an "end node."

If it is an end node (has 0 or 1 real child) then that child can be linked into its place or the node can just be deleted. We take note of the deleted node's parent, the child, and its color. The other case where it has 2 children is more complicated. We swap the predecessor (which cannot have a right child by definition) of the deleted node into the place of the deleted node, and change its color so that part of the tree is still valid. The node being removed is now effectively the predecessors old position, so we take note of its original child, parent and color.

Now we have created a situation where we are really removing an end node. We can look at the color of the node to be removed. If it is red then there is no violation of the black height rule by removing it. Also, it cannot have a real child, so there is nothing to link in its place. If the removed end node is black there are two cases. If it has a child, that child must be red or there would have been a black height violation, thus we link the child in the place and paint it black to maintain the black height. If there was no child, we have created a black height problem - there is a lack of black on this branch. The deleted node has left a null leaf node in its place, we usually count these as black, but in this case we have to call it double black to resolve the black height problem. This isn't valid so we call `doubleBlackTransform( )` to run through a set of cases to rearrange subtrees and remove the need for double black.

Chapter 6

Stack

6.1 Introduction

```
template <class Type, class Container = std::dequeue>
class std::stack
```

This chapter describes the `std::stack` adaptor. It is called an adaptor because it wraps around a real container (the Container template parameter) to store objects. It just provides a different interface to the underlying container. The `std::stack` adaptor lacks the `begin` and `end` methods, so you can't use iterators or the standard algorithms with it.

6.2 Status

The default container is currently a vector as deque has yet to be written

All members are complete:

1. `explicit stack( const Container &x = Container( ) )`
2. `empty( ) const`
3. `size( ) const`
4. `top( )` and `top( ) const`
5. `push( const value_type & )`
6. `pop( )`
7. `_Sane( )`
8. Relational operators

6.3 Design Details

The `_Sane` method is used to check if the stack is in a valid state. It is an Open Watcom extension. The method returns `true` if the stack is valid and `false` otherwise. The method works by calling the `_Sane` method of the underlying container.

Chapter 7

String

7.1 Introduction

The class template `std::string` provides dynamic strings of objects with a type given by the type parameter `CharT`. The behavior of `CharT` objects is described by a suitable traits class. By default, a specialization of `std::char_traits<CharT>` is used. Specializations of `std::char_traits` for both character and wide character types are part of the library and are used without any further intervention by the programmer.

Most of the methods in class template `std::string` are located in `hdr/watcom/string.mh`. This file is also used to generate the C library header `string.h` and the corresponding "new style" C++ library header `cstring`. This is accomplished by executing `wsplice` over the file multiple times using different options. The material that goes into the C++ library header `string` appears in `string.mh` below the C library material.

The class template `std::char_traits` along with its specializations for character and wide character, the definition of `std::string`, and certain methods of `std::string` are located in `hdr/watcom/_strdef.mh`. This file generates the header `_strdef.h` which is not to be directly included in user programs. It is, however, included in `string` thus completing the contents of `string`. The reason for this separation of `string` is because of the exception classes. The standard exception classes use strings and yet some of the methods of string throw standard exceptions. This leads to circular inclusions which are clearly unacceptable. To resolve this problem, the parts of `string` that are needed by the standard exception classes are split off into `_strdef.h`. These parts do not themselves need the standard exceptions and so the circular reference is broken.

7.2 Status

Most of the required functionality has been implemented together with moderately complete regression tests. There has so far been very little user feedback, however.

The main component that is missing is the I/O support for `std::string`. Implementing this component has been put on hold until the iostreams part of the library is reworked. In the meantime, users will have

to do string I/O using C-style strings and convert them between `std::string`. This is a significant issue; it is assumed that most standard programs will do I/O on strings directly and the library doesn't currently support such programs no matter how complete the `std::string` implementation itself might be.

In addition to the problem above, the template methods of `std::string` have not been implemented because the compiler does not yet support template methods sufficiently well.

7.3 Design Details

7.3.1 Copy-On-Write?

This implementation of `std::string` does not use a copy-on-write or a reference counted approach. Instead, every string object maintains its own independent representation. This was done, in large measure, to simplify the implementation so that a reasonable `std::string` could be offered quickly. However, there are a number of difficulties with making `std::string` reference counted, and it is worth reviewing those issues here.

The fundamental problem is that the `std::string` interface leaks references to a string's internal representation. It could be argued that this is a design problem with `std::string`. Consider the program in Listing 7.1.

Listing 7.1: Potential Reference Leakage

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string s1( "Hello" ), s2;
    char &c( s1[0] );

    s2 = s1;  // s1 and s2 perhaps share representations
    c = 'x';  // Does s2 also change?
    if( s2[0] == 'x' )
       cout << "Wrong!\n";
    else
       cout << "Right!\n";
    return 0;
}
```

The value semantics of `std::string` require that modifying one string should not influence the value seen in another logically distinct string. Thus, all correct implementations of `std::string` should produce "Right!" for the program above.

To deal with this case properly while using reference counted strings, the implementation must "unshare" the representation whenever a method is called that leaks a reference to that representation. The method

`operator[]` is one example of such a method. In fact, section 21.3, paragraph 5 of the C++ standard contains explicit language regarding this issue. The standard allows implementations to invalidate references, pointers, and iterators to the elements of a `basic_string` sequence whenever, for example, the non-const `operator[]` is called. However, this leads to rather unexpected behavior in at least two respects. In particular:

- Accessing a string might be an O(n) operation.
- Accessing a string might cause a `std::bad_alloc` exception to be thrown.

The first issue is a concern to those doing time sensitive operations, such as those writing embedded systems (Open Watcom's support for 16 bit 8086 targets might be attractive to such programmers). In fact, `std::string` provides a `reserve` method specifically to give the programmer some degree of control over when allocations are done. Copying a string's representation unexpectedly when a string is accessed frustrates this intention.

The second issue is a concern to those writing robust, exception safe code. To build code that is exception safe it is important to know when exceptions might be thrown. A savvy programmer might know that calling `std::string::operator[]` might throw an exception. However, because that is an unnatural side-effect many programmers won't be expecting it and thus using such an implementation will be error prone. Note that on some systems, notably Linux, the operating system will usually terminate the program when it runs out of memory before `std::bad_alloc` can be thrown. However, that is not the case on smaller, real-mode systems like DOS. Thus for Open Watcom this issue is a concern.

In a multithreaded program reference counted strings encounter other problems. Since Open Watcom supports a number of multithreaded targets this is also a concern.

The C++ 1998 standard does not address the semantics of programs in the face of multithreading. However, most programmers implicitly assume the following behavior (described by SGI in the documentation for their STL implementation). ⟨⟨⟨ Should this discussion be moved to a more generic part of this document? Some of this would be applicable to the whole OWSTL library. ⟩⟩⟩ **Todo**

- Two threads can read the same object without locking. This means that if reading an object changes its internal state, the implementation must provide appropriate locking.
- Two threads can operate on logically distinct objects without locking. This means that if objects share information internally, the implementation must provide appropriate locking.
- If two threads operate on the same object and at least one of the threads is modifying that object, the programmer must provide locking. This means that the implementation does not need to protect itself from this case.

Reference counted strings must deal with both situations 1 and 2 above. This means they must provide locks on the representations and use them when appropriate. The problem with this is that strings are rather low level objects and locking them is generally inappropriate. Most strings are used entirely by one thread; locks are usually only needed on larger structures. For example consider the following function:

```
typedef std::map<std::string, std::string> string_map_t;

string_map_t global_map;

void f( )
{
    // Modify the global_map.
}
```

If more than one thread is modifying the global map it would be appropriate to include a lock for the entire map. Locking the individual strings in the map would most likely be too fine-grained since a single transaction might involve updating several strings. It would be important to serialize the entire transaction. Locking the components of the transaction separately would be incorrect.

Yet a reference counted implementation of `std::string` must add locking to the strings themselves to ensure correct behavior when apparently unrelated strings are simultaneously modified. This would be adding a large amount of logically unnecessary locking overhead in cases such as the one above. This overhead can cause reference counted strings to have very poor performance when used in a multithreaded environment. ⟨⟨⟨ reference? ⟩⟩⟩ This is particularly ironic considering that reference counting is intended to improve string performance. **Todo**

Concerns about the day-to-day performance of Open Watcom's non-reference counted implementation have been partially addressed by the results of some (minimal) benchmark tests. See `bld/bench/owstl`. These tests show that the current performance of `std::string` is at least competitive with that offered by other implementations. More complete benchmark testing is needed to verify this result.

It is interesting to note that `gcc`, which at the time of this writing (2005) uses a reference counted approach, has extraordinarily poor performance on these benchmark tests. If this result stands up to further investigation it would be dramatic evidence that a reference counted approach does not automatically ensure good performance. In fact, I am led to wonder if the `gcc` maintainers did any benchmark studies of their implementation or if they just assumed that it would be fast because it is reference counted. Either way, this highlights the importance, in my mind, of following up performance assumptions by making real measurements on the final implementation. One should always verify that any change designed to improve performance actually does improve performance before committing to it.

7.3.2 Design Overview

This implementation of `std::string` uses a single pointer and two counters to define the buffer space allocated for a string. One counter measures the length of the allocated space while the other measures the number of character units in that space that are actually used. In order to meet the complexity requirements of the standard, `string` allocates more space than it needs, increasing that amount of space by a constant multiplicative factor whenever more is needed. This implementation uses a multiplicative factor of two. ⟨⟨⟨ Note: other factors, such as 1.5, might be more desirable; a factor of two causes somewhat inefficient memory reuse characteristics. Reference? ⟩⟩⟩ The capacity of a string is always an exact power of two. When a string is first created it is given a particular minimum size for its capacity (currently 8) or a capacity that is the smallest power of two larger than the new string's length, whichever is larger. **Clarify**

A string's capacity is never reduced in this implementation. Once a string's capacity is increased, the memory is not reclaimed until the string is destroyed. This was done on the assumption that if a string

was once large it will probably be large again. Not returning memory when a string's length shrinks reduces the total number of memory allocation operations and reduces the chances of an out of memory exception being thrown during a string operation. However, this design choice is not particularly friendly to low memory systems. Considering that Open Watcom targets some small machines, an alternative memory management strategy might be worth offering as an option. In the meantime programmers on such systems should be careful to destroy large strings when they are no longer needed rather than, for example, just calling `erase`.

7.3.3 Relationship to Vector

The `std::string` template is very similar in many ways to the `std::vector` template. In fact, in OWSTL both implementations use a similar representation technique and a similar memory management approach. However, the implementation of `std::vector` is more complicated because the objects in a vector need not be of a POD type (as is the case for string) so they need to be carefully copied and initialized using appropriate methods. In contrast, the `CharT` type used by `std::string` can be copied and moved with low level memory copying functions (see `std::char_traits`).

7.4 Open Watcom Extensions

Because of the widespread demand for case-insensitive string manipulation, OWSTL provides a traits class that includes case-insensitive character comparisons. An instantiation of `std::string`, called `_watcom::istring` is provided that uses this traits class.

Chapter 8

Type Traits

8.1 Introduction

The header `type_traits` is based on the metaprogramming section of n1836 "Draft Technical Report on C++ Library Extensions." It contains a set of templates that allow compile-time testing and modification of types.

8.2 Status

About half of the required functionality has been implemented so far. There are currently a few compiler bugs stopping some parts being implemented.

The missing templates are:

1. `is_member_object_pointer`
2. `is_member_function_pointer`
3. `is_enum`
4. `is_union`
5. `is_class`
6. `is_function`
7. `struct is_object`
8. `is_scalar`
9. `is_compound`
10. `is_member_pointer`

11. is_pod
12. is_empty
13. is_polymorphic
14. is_abstract
15. has_trivial_constructor
16. has_trivial_copy
17. has_trivial_assign
18. has_trivial_destructor
19. has_nothrow_constructor
20. has_nothrow_copy
21. has_nothrow_assign
22. has_virtual_destructor
23. is_signed
24. is_unsigned
25. alignment_of
26. rank
27. extent
28. is_same
29. is_base_of
30. is_convertible
31. remove_extent
32. remove_all_extents
33. add_pointer
34. aligned_storage

8.3 Design Details

8.3.1 Querying Types

This is implemented by specializing templates for the types that the test holds true. The class derives from a helper class that contains a static const value. The important cases are when this static const is a bool and is true or false. The user can then access `is_void<type>::value` to see if the test is positive. A set of macros are used to help make the definitions look a bit less cluttered. There is a default case which declares the main template and is usually false. There are then other macros that define the specializations. There are also macros that define 4 specializations for the const volatile qualified variations of the type.

8.3.2 Modifying Types

This works similarly. The template is specialized for the type with the modifier and the class contains a typedef type that refers to the modified type. Macros aren't used for the modifiers as they tend to have subtle differences for each template and in many cases there isn't the need for 4 different CV variations.

8.3.3 Use in Main Library

This header should be a help for writing the constructors and member functions of the standard containers that are required to have different behavior for iterators and integral types.

Chapter 9

Vector

9.1 Introduction

The class template `std::vector` provides a dynamic array of objects with a type given by the type parameter. Unlike `std::string` vectors can be instantiated with non-POD types. This complicates the implementation of `std::vector` considerably, as discussed below.

9.2 Status

Most of the required functionality has been implemented. Some of the methods are not yet exception safe.

9.3 Design Details

The internal structure of `vector` is very similar to that of `string`. Any enhancement or bug fix applied to either of these templates should be reviewed for possible application to the other. Like a string, a vector allocates more raw memory than it needs. This allows the logical size of the vector to increase without necessarily requiring a reallocation of memory. However, unlike a string, a vector can contain objects with user defined copy constructors and user defined `operator=`. In addition, copying and assigning objects in a vector might cause an exception to be thrown. These details make implementing `vector` more difficult than implementing `string`.

For example, consider a vector of size 100 with 200 units of memory allocated. Now suppose that 10 new objects are inserted in the middle of this vector. The 10 objects at the end of the vector need to be copying onto the raw memory just past the end using a copy constructor. However, the other objects that are moved will be placed on top of existing objects and thus must be copied with an `operator=`.

If an exception occurs while the new objects are being constructed, the objects constructed so far can be destroyed and the vector can be left in an unmodified state. However, if an exception occurs after the new objects have been created but during the assignment of the remaining objects it is somewhat unclear how to best proceed. If the new objects are destroyed data may be lost since the original copies of those objects

may have already been overwritten. Yet trying to restore the vector to its initial state is probably unwise; if an exception has occurred while copying objects around, further copying is unlikely to be successful. There is little choice but to leave the vector in a corrupted, partially modified state.