CHAPTER 4

Optimize String Use: A Case Study

> A few can touch the magic string,
> and noisy fame is proud to win them:
> Alas for those that never sing,
> but die with all their music in them!"
>
> —Oliver Wendell Holmes, "The Voiceless" (1858)

The C++ `std::string` class templates are among the most used features of the C++ standard library. For instance, an article in the Google Chromium developer forum stated that `std::string` accounted for half of all calls to the memory manager in Chromium. Any frequently executed code that manipulates strings is fertile ground for optimization. This chapter uses a discussion of optimizing string handling to illustrate recurring themes in optimization.

Why Strings Are a Problem

Strings are simple in concept, but quite subtle to implement efficiently. The particular combination of features in `std::string` interact in ways that make an efficient implementation almost impossible. Indeed, at the time this book was written, several popular compilers provided `std::string` implementations that were non-conforming in various ways.

Furthermore, the behavior of `std::string` has been changed over the years to keep up with changes in the C++ standard. This means that a conforming `std::string` implementation from a C++98 compiler may not behave the same way as a `std::string` implementation after C++11.

Strings have some behaviors that make them expensive to use, no matter the implementation. *They are dynamically allocated, they behave as values in expressions, and their implementation requires a lot of copying.*

Strings Are Dynamically Allocated

Strings are convenient because they automatically grow as needed to hold their contents. By contrast, C library functions (`strcat()`, `strcpy()`, etc.) act on fixed-size character arrays. To implement this flexibility, strings are allocated dynamically. Dynamic allocation is expensive compared to most other C++ features, so no matter what, strings are going to show up as optimization hot spots. The dynamically allocated storage is automatically released when a string variable goes out of scope or a new value is assigned to the variable. This is more convenient than having to manually release the storage, as you would with a dynamically allocated C-style character array as in the following code:

```
char* p = (char*) malloc(7);
strcpy(p, "string");
   ...
free(p);
```

The string's internal character buffer is still a fixed size, though. Any operation that makes the string longer, like appending a character or string, may cause the string to exceed the size of its internal buffer. When this happens, the operation gets a new buffer from the memory manager and copies the string into the new buffer.

`std::string` implementations do a trick to amortize the cost of reallocating storage for the character buffer as the string grows. Instead of making a request to the memory manager for the exact number of characters needed, the string implementation rounds up the request to some larger number of characters. For instance, some implementations round up the request to the next power of 2. The string then has the capacity to grow to twice its current size before needing to call the into the memory manager again. The next time an operation needs to extend the length of the string, there is room in the existing buffer, avoiding the need to allocate a new buffer. The benefit of this trick is that the cost of appending characters to a string approaches a constant asymptotically as the string grows longer. The cost of this trick is that strings carry around some unused space. If the string implements a policy of rounding up requests to a power of 2, up to half the storage in a string may be unused.

Strings Are Values

Strings behave as *values* (see "Value Objects and Entity Objects" on page 112) in assignments and expressions. Numeric constants like 2 and 3.14159 are values. You

can assign a new value to a variable, but changing the variable doesn't change the value. For example:

```
int i,j;
i = 3; // i has the value 3
j = i; // j also has the value 3
i = 5; // i now has the value 5, j still has the value 3
```

Assigning one string to another works in the same way, behaving as if each string variable has a private copy of its contents:

```
std::string s1, s2;
s1 = "hot";  // s1 is "hot"
s2 = s1;     // s2 is "hot"
s1[0] = 'n'; // s2 is still "hot", s1 is "not"
```

Because strings are values, the results of string expressions are also values. If you concatenate strings, as in the statement `s1 = s2 + s3 + s4;`, the result of `s2 + s3` is a newly allocated temporary string value. The result of concatenating `s4` to this temporary string is *another* temporary string value. This value replaces the previous value of `s1`. Then the dynamically allocated storage for the first temporary string and the previous value of `s1` are freed. This adds up to a lot of calls into the memory manager.

Strings Do a Lot of Copying

Because strings behave as values, modifying one string must not change any other strings. But strings also have mutating operations that change the contents of a string. Because of these mutating operations, each string variable must behave as if it has a private copy of its string. The simplest way to implement this behavior is to copy the string when it is constructed, assigned, or passed as an argument to a function. If strings are implemented this way, then assignment and argument passing are expensive, but mutating functions and non-`const` references are cheap.

There is a well-known programming idiom for things that behave as values but are expensive to copy. It is called "copy on write," and often abbreviated COW in C++ literature (see "Implement the "Copy on Write" Idiom" on page 136). In a COW string, the dynamically allocated storage can be shared between strings. A reference count lets each string know if it is using shared storage. When one string is assigned to another, only a pointer is copied, and the reference count is incremented. Any operation that changes a string's value first checks to see that there is only one pointer to that storage. If multiple strings point to the storage, any mutating operation (any operation that may change the contents of the string) allocates new storage and makes a copy of the string before making its change:

```
COWstring s1, s2;
s1 = "hot";  // s1 is "hot"
s2 = s1;     // s2 is "hot" (s1 and s2 point to the same storage)
s1[0] = 'n'; // s1 makes a new copy of its storage before
```

```
// changing anything
// s2 is still "hot", s1 is "not"
```

Copy-on-write is such a well-known idiom that developers may easily assume `std::string` is implemented that way. But *copy-on-write isn't even permitted for C++11-conforming implementations*, and is problematic in any case.

If strings are implemented using copy-on-write, then assignment and argument-passing are cheap, but non-`const` references plus any call to a mutating function requires an expensive allocate-and-copy operation if the string is shared. COW strings are also expensive in concurrent code. Every mutating function and non-`const` reference accesses the reference count. When the reference count is accessed by multiple threads, each thread must use a special instruction to get a copy of the reference count from main memory, to be sure no other thread has changed the value (see "Memory fences" on page 302).

In C++11 and later, the burden of copying is somewhat reduced by the presence of rvalue references and the move semantics (see "Implement Move Semantics" on page 137) that go with them. If a function takes an rvalue reference as argument, the string can do an inexpensive pointer copy when the actual argument is an rvalue expression, saving one copy.

First Attempt at Optimizing Strings

Suppose that profiling a large program reveals that the function `remove_ctrl()` reproduced in Example 4-1 consumes significant time in the program. This function removes control characters from a string of ASCII characters. It looks innocent enough, but the performance of the function as written is terrible, for many reasons. In fact, this function is a compact demonstration of the danger of completing a coding task without giving thought to performance.

Example 4-1. remove_ctrl(): code ready to optimize

```
std::string remove_ctrl(std::string s) {
    std::string result;
    for (int i=0; i<s.length(); ++i) {
        if (s[i] >= 0x20)
            result = result + s[i];
    }
    return result;
}
```

`remove_ctrl()` processes each character in argument string `s` in a loop. The code in the loop is what makes this function a hot spot. The `if`-condition gets a character

from the string and compares it to a literal constant. This is unlikely to be the problem. The controlled statement on line 5 is another story.

As noted previously, the string concatenation operator is expensive. It calls into the memory manager to construct a new temporary string object to hold the concatenated string. If the argument to `remove_ctrl()` is typically a string of printable characters, then `remove_ctrl()` constructs a new temporary string object for nearly every character in `s`. For a string with 100 characters, that's 100 calls into the memory manager to create storage for the temporary string, and another 100 calls to release the storage.

In addition to the temporary strings allocated for the result of the concatenation operation, additional strings may be allocated when the string expression is assigned to `result`, depending on how strings are implemented:

- If strings are implemented using the copy-on-write idiom, then the assignment operator performs an efficient pointer copy and increments the reference count.
- If strings have a non–shared buffer implementation, then the assignment operator must copy the contents of the temporary string. If the implementation is naïve, or `result`'s buffer does not have enough capacity, then the assignment operator also allocates a new buffer to copy into. This results in 100 copy operations and as many as 100 additional allocations.
- If the compiler implements C++11-style rvalue references and move semantics, then the fact that the concatenation expression's result is an rvalue allows the compiler to call `result`'s move constructor instead of its copy constructor. The result is that the program performs an efficient pointer copy.

The concatenation operation also copies all characters previously processed into the temporary string each time it is executed. If there are n characters in the argument string, `remove_ctrl()` copies $O(n^2)$ characters. The result of all this allocation and copying is poor performance.

Since `remove_ctrl()` is a small, standalone function, it is possible to build a test harness that calls the function repeatedly to measure exactly how much performance can be improved through optimization. Building test harnesses and measuring performance is covered in Chapter 3. The code for the test harness, and other code from the book, can be downloaded from my website.

I wrote a timing test that repeatedly called `remove_ctrl()` with a 222-character argument string containing several control characters. On average, each call took 24.8 microseconds. This number isn't important by itself, as it depends on my PC (Intel i7 tablet), operating system (Windows 8.1), and compiler (Visual Studio 2010, 32-bit, Release build). Rather, it forms a baseline for measuring performance improvement.

In the following sections, I describe a set of optimization steps, and the resulting improvement in performance of the `remove_ctrl()` function.

Use Mutating String Operations to Eliminate Temporaries

I began optimizing `remove_ctrl()` by removing allocation and copy operations. Example 4-2 is an improved version of `remove_ctrl()` in which the concatenation expression in line 5 that produced so many new temporary string objects is replaced by the mutating concatenating assignment operator, `+=`.

Example 4-2. remove_ctrl_mutating(): mutating string operators

```
std::string remove_ctrl_mutating(std::string s) {
    std::string result;
    for (int i=0; i<s.length(); ++i) {
        if (s[i] >= 0x20)
            result += s[i];
    }
    return result;
}
```

This small change has a dramatic effect on performance. The same timing test now showed an average of only 1.72 microseconds per call, a 13x improvement. This improvement comes from eliminating all the calls to allocate temporary string objects to hold the concatenation expression result, and the associated copying and deleting of temporaries. Depending on the string implementation, allocation and copying on assignment are also eliminated.

Reduce Reallocation by Reserving Storage

The function `remove_ctrl_mutating()` still performs an operation that lengthens `result`. This means `result` is periodically copied into a larger internal dynamic buffer. As previously discussed, one possible implementation of `std::string` doubles the amount of storage allocated each time the capacity of the string's character buffer is exceeded. When `std::string` is implemented with this rule, reallocation might happen as many as 8 times for a 100-character string.

If we assume that strings are generally printable characters, with only occasional control characters to remove, then the length of the string argument `s` provides an excellent estimate of the result string's eventual length. Example 4-3 improves on `remove_ctrl_mutating()` by using `std::string`'s `reserve()` member function to preallocate the estimated amount of storage. Not only does use of `reserve()` eliminate reallocation of the string buffer, but it also improves the cache locality of the data accessed by the function, so we get even more mileage out of this change.

Example 4-3. remove_ctrl_reserve(): reserving storage

```
std::string remove_ctrl_reserve(std::string s) {
    std::string result;
    result.reserve(s.length());
    for (int i=0; i<s.length(); ++i) {
        if (s[i] >= 0x20)
            result += s[i];
    }
    return result;
}
```

The removal of several allocations causes a significant improvement in performance. A test using `remove_ctrl_reserve()` consumes 1.47 microseconds per call, an improvement of 17% over `remove_ctrl_mutating()`.

Eliminate Copying of String Arguments

So far, I have been successful at optimizing `remove_ctrl()` by removing calls on the memory manager. It is thus reasonable to continue looking for allocations to remove.

When a string expression is passed into a function by value, the formal argument (in this case, `s`) is copy-constructed. This may result in a copy, depending on the string implementation:

- If strings are implemented using the copy-on-write idiom, then the compiler generates a call to the copy constructor, which performs an efficient pointer copy and increments the reference count.
- If strings have a nonshared buffer implementation, then the copy constructor must allocate a new buffer and copy the contents of the actual argument.
- If the compiler implemented C++11-style rvalue references and move semantics, then if the actual argument is an expression, it will be an rvalue, so the compiler will generate a call to the move constructor, resulting in an efficient pointer copy. If the actual argument is a variable, then the formal argument's copy constructor is called, resulting in an allocation-and-copy. Rvalue references and move semantics are described in more detail in "Implement Move Semantics" on page 137.

`remove_ctrl_ref_args()` in Example 4-4 is an improved function that never copies `s` on call. Since the function does not modify `s`, there is no reason to make a separate copy of `s`. Instead, `remove_ctrl_ref_args()` takes a `const` reference to `s` as an argument. This saves another allocation. Since allocation is expensive, it can be worthwhile eliminating even one allocation.

Example 4-4. remove_ctrl_ref_args(): argument copy eliminated

```
std::string remove_ctrl_ref_args(std::string const& s) {
    std::string result;
    result.reserve(s.length());
    for (int i=0; i<s.length(); ++i) {
        if (s[i] >= 0x20)
            result += s[i];
    }
    return result;
}
```

The result is a surprise. The timing test of `remove_ctrl_ref_args()` took 1.60 microseconds per call, 8% worse than `remove_ctrl_reserve()`.

What's going on here? Visual Studio 2010 copies string values on call, so this change should save an allocation. Either that savings isn't realized, or something else related to the change of `s` from string to string reference must consume that savings.

Reference variables are implemented as pointers. So, everywhere `s` appears in `remove_ctrl_ref_args()`, the program dereferences a pointer that it did not have to dereference in `remove_ctrl_reserve()`. I hypothesize that this extra work might be enough to account for the reduced performance.

Eliminate Pointer Dereference Using Iterators

The solution is to use iterators over the string, as in Example 4-5. String iterators are simple pointers into the character buffer. That saves two dereference operations versus the non-iterator code in the loop.

Example 4-5. remove_ctrl_ref_args_it(): iterator version of remove_ctrl_ref_args()

```
std::string remove_ctrl_ref_args_it(std::string const& s) {
    std::string result;
    result.reserve(s.length());
    for (auto it=s.begin(),end=s.end(); it != end; ++it) {
        if (*it >= 0x20)
            result += *it;
    }
    return result;
}
```

The timing test for `remove_ctrl_ref_args_it()` produced a satisfying result of 1.04 microseconds per call. It is definitely superior to the non-iterator version. But what about making `s` a string reference? To find out if this optimization actually improved anything, I coded an iterator version of `remove_ctrl_reserve()`. The timing test for `remove_ctrl_reserve_it()` took 1.26 microseconds per call, down from 1.47 micro-

seconds. The string reference argument optimization definitely improved performance.

In fact, I coded iterator versions of all the `remove_ctrl()`-derived functions. Iterators were a definite win in all cases versus the subscripting versions (however, in "Second Attempt at Optimizing Strings" on page 80, we'll see that this is not always the case).

`remove_ctrl_ref_args_it()` contains one other optimization of note. The value `s.end()`, used to control the `for` loop, is cached on loop initialization. This saves another *2n* indirections, where *n* is the length of the argument string.

Eliminate Copying of Returned String Values

The original `remove_ctrl()` function returns its result by value. C++ copy-constructs the result into the calling context, though the compiler is permitted to *elide* (that is, simplify by removing) the copy construction if it can. If we want to be *sure* that there is no copy, we have a couple of choices. One choice that works for all versions of C++ and all string implementations is to return the string as an out parameter. This is actually what the compiler does when it elides a copy construction. An improved version of `remove_ctrl_ref_args_it()` appears in Example 4-6.

Example 4-6. remove_ctrl_ref_result_it(): copy of return value eliminated

```
void remove_ctrl_ref_result_it(
    std::string& result,
    std::string const& s)
{
    result.clear();
    result.reserve(s.length());
    for (auto it=s.begin(),end=s.end(); it != end; ++it) {
        if (*it >= 0x20)
            result += *it;
    }
}
```

When a program calls `remove_ctrl_ref_result_it()`, a reference to some string variable is passed into the formal argument `result`. If the string variable referenced by `result` is empty, then the call to `reserve()` allocates storage for enough characters. If the string variable has been used before—if the program called `remove_ctrl_ref_result_it()` in a loop—its buffer may already be big enough, in which case no new allocation takes place. When the function returns, the string variable in the caller holds the return value, with no copy required. The beautiful thing about `remove_ctrl_ref_result_it()` is that in many cases, it eliminates every allocation.

Measured performance of `remove_ctrl_ref_result_it()` is 1.02 microseconds per call, about 2% faster than the previous version.

`remove_ctrl_ref_result_it()` is *very* efficient, but its interface is open to misuse in a way that the original `remove_ctrl()` was not. A reference—even a `const` reference—doesn't behave exactly the same as a value. The following call will produce unintended results, returning an empty string:

```
std::string foo("this is a string");
remove_ctrl_ref_result_it(foo, foo);
```

Use Character Arrays Instead of Strings

When a program requires the ultimate in performance, it is possible to bypass the C++ standard library altogether and handcode the function using the C-style string functions, as in Example 4-7. The C-style string functions are harder to use than `std::string`, but the gain in performance can be impressive. To use the C-style string functions, the programmer must choose either to manually allocate and free character buffers, or to use static arrays dimensioned to worst-case sizes. Declaring a bunch of static arrays is problematic if memory is at all constrained. However, there is usually room to statically declare large temporary buffers in local storage (that is, on the function call stack). These buffers are reclaimed at negligible runtime cost when the function exits. Except in the most constrained embedded environments, it is no problem to declare a worst-case buffer of 1,000 or even 10,000 characters on the stack.

Example 4-7. remove_ctrl_cstrings(): coding on the bare metal

```
void remove_ctrl_cstrings(char* destp, char const* srcp, size_t size) {
    for (size_t i=0; i<size; ++i) {
        if (srcp[i] >= 0x20)
            *destp++ = srcp[i];
    }
    *destp = 0;
}
```

`remove_ctrl_cstrings()` took 0.15 microseconds per call in the timing test. This is 6 times faster than its predecessor, and an astonishing 170 times faster than the original function. One reason for the improvement is the elimination of several function calls, with a corresponding improvement in cache locality.

Excellent cache locality may, however, contribute to a simple performance measurement being misleading. In general, other operations between calls to `remove_ctrl_cstrings()` flush the cache. But when it is called in a tight loop, instructions and data stay in cache.

Another factor affecting `remove_ctrl_cstrings()` is that it has a markedly different interface to the original function. If called from many locations, changing all the calls is a significant effort, and may be one optimization too many. Nonetheless, `remove_ctrl_cstrings()` is an example of just how much performance can be gained by the developer willing to completely recode a function and change its interface.

Stop and Think

I think we might be going a bridge too far.

—Lt. General Frederick "Boy" Browning (1896–1965)

Comment to Field Marshal Montgomery on September 10, 1944, expressing his concerns about the Allied plan to capture the bridge at Arnhem. Browning's concern was prophetic, as the operation was a disaster.

As noted in the previous section, an optimization effort may arrive at a point where simplicity or safety must be traded for additional performance gains. `remove_ctrl_ref_result_it()` required a change in function signature that introduced a potential incorrect usage not possible with `remove_ctrl()`. For `remove_ctrl_cstrings()`, the cost was manually managing the temporary storage. For some teams, this may be a bridge too far.

Developers have differing opinions, sometimes very strong ones, about whether a particular performance improvement justifies the added complexity of the interface or increased need to review uses of the function. Developers who favor an optimization like returning a value in an out parameter might say that the dangerous use case isn't all that likely, and could be documented. Returning the string in an out parameter also leaves the function's return value available for useful things like error codes. Those who oppose this optimization might say there is nothing obvious to warn users away from the dangerous use case, and a subtle bug is more trouble than the optimization is worth. In the end, the team must answer the question, "How much do we need this performance improvement?"

I can offer no guidance on exactly when an optimization effort has gone too far. It depends on how important the performance improvement is. But developers should note the transition and take a moment to stop and think.

C++ offers developers a range of choices between simple and safe code that is slow and radically fast code that must be used carefully. Advocates of other languages may call this a weakness, but for optimization, it is one of the greatest strengths of C++.

Summary of First Optimization Attempt

Table 4-1 summarizes the results of the optimization effort for `remove_ctrl()`. These results were obtained by pursuing one simple rule: remove memory allocations and associated copy operations. The first optimization produced the most significant speedup.

Many factors affect the absolute timings, including the processor, fundamental clock rate, memory bus frequency, compiler, and optimizer. I've provided test results of both debug and release (optimized) builds to demonstrate this point. While the release build code is *much* faster than the debug code, improvement is visible in both debug and release builds.

Table 4-1. Performance summary VS 2010, i7

Function	Debug	Δ	Release	Δ	Release vs. debug
`remove_ctrl()`	967 µs		24.8 µs		3802%
`remove_ctrl_mutating()`	104 µs	834%	1.72 µs	1,341%	5923%
`remove_crtl_reserve()`	102 µs	142%	1.47 µs	17%	6853%
`remove_ctrl_ref_args_it()`	215 µs	9%	1.04 µs	21%	20559%
`remove_ctrl_ref_result_it()`	215 µs	0%	1.02 µs	2%	21012%
`remove_ctrl_cstrings()`	1 µs	9,698%	0.15 µs	601%	559%

The percentage of improvement appears far more dramatic in release builds. This is probably an effect of Amdahl's Law. In the debug build, function inlining is turned off, which increases the cost of every function call and causes the fraction of run time devoted to memory allocation to be lower.

Second Attempt at Optimizing Strings

There are other roads the developer can travel in the quest for better performance; we'll explore a few more options in this section.

Use a Better Algorithm

One option is to attempt to improve the algorithm. The original `remove_ctrl()` uses a simple algorithm that copies one character at a time to the result string. This unfortunate choice produces the worst possible allocation behavior. Example 4-8 improves on the original design by moving whole substrings to the result string. This change has the effect of reducing the number of allocate-and-copy operations. Another optimization introduced in `remove_ctrl_block()` is caching the length of the argument string to reduce the cost of the loop termination clause of the outer `for` loop.

Example 4-8. remove_ctrl_block(): a faster algorithm

```
std::string remove_ctrl_block(std::string s) {
    std::string result;
    for (size_t b=0, i=b, e=s.length(); b < e; b = i+1) {
        for (i=b; i<e; ++i) {
            if (s[i] < 0x20)
                break;
        }
        result = result + s.substr(b,i-b);
    }
    return result;
}
```

`remove_ctrl_block()` runs the timing test in 2.91 microseconds, about 7 times faster than the original `remove_ctrl()`.

This function in turn can be improved in the same way as before, by replacing the concatenation with a mutating assignment (`remove_ctrl_block_mutate()`, 1.27 microseconds per call), but `substr()` still constructs a temporary string. Since the function appends characters to `result`, the developer can use one of the overloads of `std::string`'s `append()` member function to copy the substring without creating a string temporary. The resulting function, `remove_ctrl_block_append()` (as shown in Example 4-9), runs the timing test in 0.65 microseconds per call. This result handily beats the best time of 1.02 microseconds per call for `remove_ctrl_ref_result_it()`, and is 36 times better than the original `remove_ctrl()`. This is a concise demonstration of the power of selecting a good algorithm.

Example 4-9. remove_ctrl_block_append(): a faster algorithm

```
std::string remove_ctrl_block_append(std::string s) {
    std::string result;
    result.reserve(s.length());
    for (size_t b=0,i=b; b < s.length(); b = i+1) {
        for (i=b; i<s.length(); ++i) {
            if (s[i] < 0x20) break;
        }
        result.append(s, b, i-b);
    }
    return result;
}
```

These results can in turn be improved by reserving storage in `result` and eliminating the argument copy (`remove_ctrl_block_args()`, 0.55 microseconds per call), and by removing the copy of the returned value (`remove_ctrl_block_ret()`, 0.51 microseconds per call).

One thing that did not improve the results, at least at first, was recoding `remove_ctrl_block()` using iterators. However, after both the argument and return value had been made reference parameters, the iterator version went suddenly from being 10 times as expensive to 20% less expensive, as shown in Table 4-2.

Table 4-2. Performance summary of second remove_ctrl algorithm

	Time per call	Δ vs. previous
`remove_ctrl()`	24.8 µs	
`remove_ctrl_block()`	2.91 µs	751%
`remove_ctrl_block_mutate()`	1.27 µs	129%
`remove_ctrl_block_append()`	0.65 µs	95%
`remove_ctrl_block_args()`	0.55 µs	27%
`remove_ctrl_block_ret()`	0.51 µs	6%
`remove_ctrl_block_ret_it()`	0.43 µs	19%

Another way to improve performance is to mutate the argument string by removing control characters from it using `std::string`'s mutating `erase()` member function. Example 4-10 demonstrates this approach.

Example 4-10. remove_ctrl_erase(): mutating the string argument instead of building a result string

```
std::string remove_ctrl_erase(std::string s) {
    for (size_t i = 0; i < s.length(); )
        if (s[i] < 0x20)
            s.erase(i,1);
        else ++i;
    return s;
}
```

The advantage of this algorithm is that, since `s` gets shorter, there is never any reallocation, except possibly for the returned value. The performance of this function is excellent, completing the test in 0.81 microseconds per call, 30 times faster than the original `remove_ctrl()`. A developer arriving at this excellent result on the first attempt might be excused for declaring victory and retiring from the field of battle without any further effort. *Sometimes a different algorithm is easier to optimize or inherently more efficient.*

Use a Better Compiler

I ran the same timing tests using the Visual Studio 2013 compiler. Visual Studio 2013 implements move semantics, which should have made some of the functions considerably faster. However, the results were mixed. Running under the debugger, Visual Studio 2013 was 5%–15% percent faster than Visual Studio 2010. Running from the

command line, VS2013 was 5%–20% percent slower. I tried the Visual Studio 2015 release candidate, but it was slower yet. This may have been due to changes in the container classes. A new compiler may improve performance, but it's something the developer must test, rather than taking it on faith.

Use a Better String Library

The definition of `std::string` was originally quite vague, to allow for a wide range of implementations. The demands of efficiency and predictability eventually drove clarifications to the C++ standard that eliminated most novel implementations. The behavior defined for `std::string` is thus a compromise, having evolved out of competing design considerations over a long period of time:

- Like other standard library containers, `std::string` provides iterators to access the individual characters of the string.
- Like C character strings, `std::string` provides an array-like indexing notation using `operator[]` to access its elements. `std::string` also provides a mechanism to obtain a pointer to a C-style null-terminated version of the string.
- `std::string` has a concatenation operator and value-returning functions that give it value semantics, similar to BASIC strings.
- `std::string` provides a limited set of operations that some users find constraining.

The desire to make `std::string` as efficient as C-style `char` arrays pushes the implementation toward representing the string in contiguous memory. The C++ standard requires that the iterators be random-access, and forbids copy-on-write semantics. This makes it easier to define and to reason about what actions invalidate iterators into a `std::string`, but it limits the scope for clever implementations.

Furthermore, the `std::string` implementation that comes with a commercial C++ compiler must be straightforward enough that it can be tested, to guarantee that it produces standard-conforming behavior and acceptable efficiency in every conceivable situation. The cost to the compiler vendor for making a mistake is high. This pushes implementations toward simplicity.

The standard-defined behavior of `std::string` induces some weaknesses. Inserting a single character in a million-character string causes the whole suffix of the string to be copied, and perhaps reallocated. Similarly, all the value-returning substring operations must allocate and copy their results. Some developers search for optimization opportunities by lifting one or more of the previous constraints (iterators, indexing, C-string access, value semantics, simplicity).

Adopt a richer library for std::string

Sometimes using a better library means nothing more than providing additional string functions. Here are a few of the many libraries that work with `std::string`:

Boost string library
: The Boost string library provides functions for tokenizing, formatting, and more exotic manipulations of `std::string`. It's a treat for those who are deeply in love with the standard library's `<algorithm>` header.

C++ String Toolkit Library
: Another choice is the C++ String Toolkit Library (StrTk). StrTk is particularly good at parsing and tokenizing strings, and is compatible with `std::string`.

Use std::stringstream to avoid value semantics

C++ already contains several string implementations: the templatized, iterator-accessed, variable-length character strings of `std::string`; the simple, iterator-based `std::vector<char>`; and the older, C-style null-terminated character strings in fixed-size arrays.

Although C-style character strings are tricky to use well, we've already seen an experiment that showed that replacing C++'s `std::string` with C-style character arrays improved performance dramatically under the right conditions. Neither of these implementations is perfect for every situation.

C++ contains yet another kind of string. `std::stringstream` does for strings what `std::ostream` does for output files. The `std::stringstream` class encapsulates a dynamically resizable buffer (in fact, usually a `std::string`) in a different way, as an *entity* (see "Value Objects and Entity Objects" on page 112) to which data can be appended. `std::stringstream` is an example of how putting a different API on top of a similar implementation can encourage more efficient coding. Example 4-11 illustrates its use.

Example 4-11. std::stringstream: like string, but an object

```
std::stringstream s;
for (int i=0; i<10; ++i) {
    s.clear();
    s << "The square of " << i << " is " << i*i << std::endl;
    log(s.str());
}
```

This snippet shows several optimal coding techniques. Since `s` is modified as an entity, the long inserter expression doesn't create any temporaries that must be allocated and copied into. Another deliberate practice is that `s` is declared outside of the

loop. In this way, the internal buffer inside `s` is reused. The first time through the loop the buffer may have to be reallocated several times as characters are appended, but subsequent iterations are unlikely to reallocate the buffer. By contrast, if `s` had been declared inside the loop, an empty buffer would have been constructed each time through the loop, and potentially reallocated as the insertion operator appended characters.

If `std::stringstream` is implemented using a `std::string`, it can never truly outperform `std::string`. Its advantage lies in preventing some of the programming practices that lead to inefficiency.

Adopt a novel string implementation

A developer may find the whole string abstraction inadequate. One of the most important C++ features is that abstractions like strings are not built into the language, but instead provided as templates or function libraries. Alternate implementations have access to the same language features as `std::string`, so performance may be improved by a sufficiently clever implementer. Lifting one or more of the constraints listed at the beginning of this section (iterators, indexing, C-string access, simplicity) grants a custom string class access to optimizations denied to `std::string`.

Many clever string data structures have been proposed over time that promise to significantly reduce the cost of reallocating memory and copying string contents. This may be a siren song, for several reasons:

- Any pretender to the throne of `std::string` must be both more expressive and more efficient than `std::string` in a wide variety of situations. Most proposed alternative implementations don't come with good guarantees of increased general performance.
- Replacing every occurrence of `std::string` in a large program with some other string is a significant undertaking, with no assurance that it will make a difference to performance.
- While many alternative string concepts have been proposed, and several have been implemented, it will take more than a few minutes of Googling to find a string implementation as complete, well tested, and well understood as `std::string`.

Replacing `std::string` may be more practical when considering a design than when optimizing one. It may be possible for a big team with time and resources. But the payoff is uncertain enough to make this optimization perhaps a bridge too far. Still, for the brave and the desperate there is code out in the wild that might help:

`std::string_view`

: `string_view` solves some of the problems of `std::string`. It contains an unowned pointer to string data and a length, so that it represents a substring of a `std::string` or literal character string. Operations like substring and trim are more efficient than the corresponding value-returning member functions of `std::string`. `string_view` is on its way to appearing in C++14. Some compilers have it now in `std::experimental`. `string_view` has mostly the same interface as `std::string`.

  The problem with `string_view` is that the pointer is unowned. The programmer must ensure that the lifetime of any `string_view` is not longer than the lifetime of the `std::string` it points into.

`folly::fbstring`

: Folly is a whole library of code used by Facebook in its own servers. It includes a very highly optimized direct replacement for `std::string` that implements a nonallocated buffer to optimize short strings. `fbstring`'s designers claim measurable performance improvements.

  Because of its heritage, Folly is unusually likely to be robust and complete. Folly is supported on Linux.

A Toolbox of String Classes

: This article and code from the year 2000 describes a templatized string type with the same interface as the SGI implementation of `std::string`. It provides a fixed-maximum-size string and a variable-length string type. It is a *tour de force* of template metaprogramming magic, which may make it hard for some people to understand. For someone committed to designing a better string class, it is a viable candidate.

"C++03 Expression Templates"

: This is a 2005 paper presenting some template code to solve the specific problem of concatenation. Expression templates override the + operator, to create an intermediate type that represents concatenation of two strings or a string and a string expression symbolically. Expression templates defer allocation and copying to the end of the expression performing one allocation when the expression template is assigned to a string. Expression templates are compatible with `std::string`. The existing code can improve performance significantly when a string expression is a long list of concatenated substrings. The same concept could be extended to a whole string library.

The Better String Library

: This code archive contains a general-purpose string implementation that is different from `std::string` but contains some powerful features. If many strings

are constructed from parts of other strings, `bstring` allows one string to be formed from an offset and length within another string. I worked with a proprietary implementation of this idea that was very efficient. There is a C++ wrapper class called `CBString` for the `bstring` library.

`rope<T,alloc>`
: This is a string library suitable for making insertions and deletions in very long strings. It's not compatible with `std::string`.

Boost String Algorithms
: This is a library of string algorithms to supplement the `std::string` member functions. They are built around a find-and-replace concept.

Use a Better Allocator

Inside every `std::string` is a dynamically allocated array of `char`. `std::string` is a specialization of a more general template that looks like this:

```
namespace std {
    template < class charT,
               class traits = char_traits<charT>,
               class Alloc = allocator<charT>
               > class basic_string;

    typedef basic_string<char> string;
    ...
};
```

The third template parameter, `Alloc`, defines an *allocator*, a specialized interface to the C++ memory manager. `Alloc` defaults to `std::allocator`, which calls `::operator new()` and `::operator delete()`, the global C++ memory allocator functions.

The behavior of `::operator new()` and `::operator delete()`, and of allocator objects, is covered in more detail in Chapter 13. For now, I'll just say that `::operator new()` and `::operator delete()` have a very complex and difficult job, allocating storage for all the many kinds of dynamic variables. They have to work for big and small objects, and for single- and multithreaded programs. Their design is a compromise to achieve such generality. Sometimes, a more specialized allocator can do a better job. Thus, `Alloc` can be specified as something other than the default to provide a specialized allocator for `std::string`.

I coded an extremely simple allocator to demonstrate what kind of performance improvement might be achievable. This allocator manages a few fixed-size blocks of memory. I created a new `typedef` for a kind of string that uses this allocator. Then I changed the original, very inefficient version of `remove_ctrl()` to use the special strings, as shown in Example 4-12.

Example 4-12. Original version of remove_ctrl() with simple fixed block allocator

```
typedef std::basic_string<
    char,
    std::char_traits<char>,
    block_allocator<char, 10>> fixed_block_string;

fixed_block_string remove_ctrl_fixed_block(std::string s) {
    fixed_block_string result;
    for (size_t i=0; i<s.length(); ++i) {
        if (s[i] >= 0x20)
            result = result + s[i];
    }
    return result;
}
```

The result was dramatic. `remove_ctrl_fixed_block()` ran the same test in 13,636 milliseconds, about 7.7 times faster than the original.

Changing allocators is not for the faint of heart. You can't assign strings based on different allocators to one another. The modified example shown here only works because `s[i]` is a `char`, rather than a one-character `std::string`. You can copy the contents of one string to another by converting it to a C string, for instance, by saying `result = s.c_str();`.

Changing all the occurrences of `std::string` to `fixed_block_string` has a massive effect on a code base. For this reason, if a team thinks they may fiddle with their strings, creating a project-wide `typedef` early on in the design is a good idea:

```
    typedef std::string MyProjString;
```

Then experiments involving a global change can be performed in one place. This will still only work, though, if the new string has the same member functions as the one it replaces. Differently allocated `std::basic_string`s have this property.

Eliminate String Conversion

Among the complexities of the modern world is that there is more than one kind of character string. Generally, string functions only permit like kinds of string to be compared, assigned, or used as operands or arguments, so the programmer must convert from one kind of string to another. Any time conversion involves copying characters or allocating dynamic memory is an opportunity to improve performance.

The library of conversion functions themselves can be tuned. More importantly, the design of a large program can limit conversion.

Conversion from C String to std::string

One common source of wasted computer cycles is unnecessary conversion from null-terminated character strings to `std::string`. For instance:

```
std::string MyClass::Name() const {
    return "MyClass";
}
```

This function must convert the string constant `MyClass` to a `std::string`, allocating storage and copying the characters into the `std::string`. C++ does this conversion automatically because `std::string` has a constructor that takes a `char*` argument.

The conversion to `std::string` is unnecessary. `std::string` has a constructor that accepts a `char*` argument, so the conversion will happen automatically when the value returned by `Name()` is assigned to a string or passed to a function that takes a string argument. The previous function could as easily have been written as follows:

```
char const* MyClass::Name() const {
    return "MyClass";
}
```

This delays conversion of the returned value to the point where it is actually used. At the point of use, conversion is often not needed:

```
char const* p = myInstance->Name(); // no conversion
std::string s = myInstance->Name(); // conversion to std::string
std::cout << myInstance->Name();    // no conversion
```

What makes string conversion a big problem is that a big software system may have multiple layers. If one layer takes a `std::string` and the layer below it takes a `char*`, there may be code to reverse the conversion to `std::string`:

```
void HighLevelFunc(std::string s) {
    LowLevelFunc(s.c_str());
}
```

Converting Between Character Encodings

Modern C++ programs have to deal with comparing (for instance) a literal C string (ASCII, in signed bytes) to a UTF-8 (unsigned, variable bytes per character) string from a web browser, or with converting output character strings from an XML parser that produces UTF-16 word streams (with or without endian bytes) to UTF-8. The number of possible combinations is daunting.

The best way to eliminate conversions is to pick a single format for all strings, and store all strings in that format. You might want to provide specialized comparison functions between your chosen format and C-style null-terminated strings, so they don't have to be converted. I like UTF-8 because it can represent all Unicode code

points, is directly comparable (for equality) with C-style strings, and is produced by most browsers.

In large and hastily written programs, you may find a string converted from an original format to a new format, and then back to its original format as it passes through layers of the software. The fix for this is to rewrite member functions at the class interfaces to take the same type of strings. Unfortunately, this task is like adding `const`-correctness to a C++ program. The change tends to ripple through the entire program in ways that make it difficult to control the scope of the change.

Summary

- *Strings are expensive to use because they are dynamically allocated, they behave as values in expressions, and their implementation requires a lot of copying.*
- *Treating strings as objects instead of values reduces the frequency of allocation and copying.*
- *Reserving space in a string reduces the overhead of allocation.*
- *Passing a `const` reference to a string into a function is almost the same as passing the value, but can be more efficient.*
- *Passing a result string out of a function as a reference reuses the actual argument's storage, which is potentially more efficient than allocating new storage.*
- *An optimization that only sometimes removes allocation overhead is still an optimization.*
- *Sometimes a different algorithm is easier to optimize or inherently more efficient.*
- *The standard library class implementations are general-purpose and simple. They are not necessarily high-performance, or optimal for any particular use.*