What happens when you apply a generic programming approach to string management in C++? **Kevlin Henney** finds out

The next best string

THE PREVIOUS TWO COLUMNS HAVE EXPLORED the strengths and weaknesses of raw C-style strings, the standard library `basic_string` template and some alternative approaches to expressing strings[1, 2].

C-style strings still have their place, but suffer from being too low-level and error-prone for many common string tasks. `std::basic_string` has a clumsy interface in places, but it is standard. It is also better than many proprietary strings and is still essentially usable. But in addition to its interface design problems, it also suffers from another simple limitation: there is only one of it. In other words, it defines a single string type rather than a framework for strings—an underachiever in a utility space pushed and pulled by so many different needs.

While imperfect, the raw or `basic_string` approach was certainly more accessible than the difficulties and workarounds thrown up by an inheritance-based approach to generalisation—an approach that makes such a meal of things that you are tempted to get a knife and fork and join in.

At heart, a string is a value that represents a sequence of characters. It is largely defined by both its functional use and external, largely non-functional requirements—quality requirements such as performance for certain operations or memory usage.

Inside `std::basic_string` is a small and sufficient string interface struggling to get out, and the basis of an open approach to string generalisation. That subset is to be found in the STL and that approach is generic programming.

Any container that can express a sequence of characters has the potential to be used as a string—such as `std::vector<char>`—although `std::string` will suit many tasks and many developers as it stands. The real strength of the generic programming approach is not so much the data structure side as the algorithmic side: function templates working through iterators allow common (and less common) tasks to be written independently of the chosen representation for a string.

Search...

To find the first occurrence of a character in a string, use `std::find`. Here is the pseudo-code skeleton:

```
position of found character =
               std::find(
        beginning of string,
        end of string,
        character to find);
```

This works for any type playing the role of a string, whether using:

```
std::basic_string:

std::string text;
...
std::string::iterator first_space =
    std::find(text.begin(), text.end(), ' ');
```

Or `std::vector`:

```
std::vector<char> text;
...
std::vector<char>::iterator first_space =
    std::find(text.begin(), text.end(),
 ' ');
```

Or raw C-style strings:

```
char *text;
...
```

FACTS AT A GLANCE

- **STL-based sequences, in combination with STL-based operations, often provide a simple and effective approach to string representation and manipulation.**
- **Standard algorithmic function templates support all the usual searching and modification operations required for string manipulation, and more.**
- **Conversion between 'proper' string types and STL containers is trivial.**
- **A simplified regular-expression pattern matching function template, independent of string type, is a useful application and demonstration of generic programming techniques.**

```
char *first_space =
    std::find(text, std::strchr(text, '\0'), ' ');
```

This approach is clearly more widely applicable than `std::basic_string`'s own `find` member functions. It is also simple to write a consistent loop to step through the loop, finding each successive character match:

```
std::string text;
...
for(std::string::iterator found = text.begin();
    (found = std::find(found, text.end(), ' ')) !=
text.end();
    ++found)
{
    ...
}
```

To search backwards from the end of the string does not require a separately named function, as with `std::basic_string's rfind` member. The same `find` function can be used, but this time with reverse iterator range obtained from the sequence's `rbegin` and `rend` members.

Finding the first occurrence from a set of characters is also a straightforward task:

```
const std::string vowels = "aeiou";
std::string::iterator first_vowel =
    std::find_first_of(
        text.begin(), text.end(),
        vowels.begin(), vowels.end());
```

But the algorithmic independence really begins to shine when we step beyond what `std::basic_string's` members are capable of on their own:

```
std::string::iterator first_digit =
    std::find_if(text.begin(), text.end(), std::isdigit);
```

The standard `find_if` template takes a predicate—either a function or function object returning `bool` based on a single argument—that is used to test for the found condition. In this case, it is whether or not a character is a digit, as specified by the C library's `isdigit` function.

You can also locate occurrences of a substring:

```
std::string::iterator found =
    std::search(
        text.begin(), text.end(),
        subtext.begin(), subtext.end());
```

Continuing in this vein, the other non-modifying search-based operations defined in `<algorithm>` also prove useful: `search_n`, `find_end`, `adjacent_find`, `count`, `count_if` and `mismatch`.

...and destroy

Let's revisit that common task of replacing particular characters from a string, such as a dash to a space in a numeric or alphanumeric code[3]:

```
std::string text;
...
std::replace(text.begin(), text.end(), '-', ' ');
```

The generic nature of the standard algorithmic function templates allows you to understand and forgive the initial apparent quirkiness of the idiom used for removing values from a sequence. For instance, removing spaces from a sequence, to compact it:

```
text.erase(
    std::remove(text.begin(), text.end(), ' '),
    text.end());
```

Why is that the `std::remove` operation doesn't actually live up to its name and remove the specified character bodily from the string? Because it doesn't know anything about the string as an object: its keyhole view on the world is through an iterator range, so the best it can do is compact that range to exclude the specified character, and return where the end of the new compacted range would be. It is up to you, the caller, to make a decision as to what to do with that end value. Given that the values beyond it are unspecified, the most common task is to shrink the sequence to fit using its `erase` member, i.e. “drop the sequence from the end of the new compacted range to the end of the current sequence”. This means that `std::remove` also makes sense for C-style strings, where truncating the string is a trivial task:

```
char *text;
...
*std::remove(text, std::strchr(text, '\0'), ' ') = '\0';
```

The compositional nature of the STL, and in particular the ability to attribute paramters to many algorithmic functions using functions or function objects, reduces the need for wide interfaces that try to cater for all needs explicitly. “Why is there no simple mechanism for shifting all the characters in a string to either upper or lower case?” is a `std::basic_string` FAQ that can be partly addressed from this perspective. The real answer is that, outside of English, the meaning of such a case transformation, or the meaning of case insensitivity, is not always either obvious or trivial[4, 5]. However, where the simple per-character transformation offered by the C library `tolower` and `toupper` is sufficient, it is possible to scale their use to a whole string without the need for any new functions:

```
std::string text;
...
std::transform(
    text.begin(), text.end(), text.begin(), std::toupper);
```

The `std::transform` operation transforms the contents of one iterator range into another according to the supplied function or function object. In this case, the output range, `text`, starts in the same place as the input range, `text`, so that the transformed string overwrites the original.

And when you've had enough of it all, you can just blank out the string:

```
std::fill(text.begin(), text.end(), ' ');
```

String to string

When it comes to some operations, such as I/O or concatenation, types designed specifically for general string use tend to offer an easier usage path. Because of the work involved, input is best left to types written with that feature in mind. However, simple output for arbitrary sequences is not difficult if that is only an occasional need. The metaphor of copying to output gives the following solution:

```
typedef std::ostream_iterator<char> out;
```

```
std::vector<char> text;
...
std::copy(text.begin(), text.end(), out(std::cout));
```

The relevant output stream is wrapped as an output iterator and then used as the target for the copied range. `typedefs`, or wrapper functions, can improve readability and convenience.

For concatenation, general string types such as `std::basic_string` and SGI's `rope` template[6] typically offer a number of convenient `append` and `operator+=` overloads:

```
std::string text;
...
text.append(3, '.');
text += "?!";
```

For arbitrary sequences, the equivalent effect can be achieved a little less conveniently through insertion members:

```
std::vector<char> text;
...
text.insert(text.end(), 3, '.');
const char *const suffix = "?!";
text.insert(text.end(), suffix, std::strchr(suffix, '\0'));
```

This fragment highlights that when it comes to dealing with string literals, either `const char *` or `std::string` tend to offer the path of least resistance. Other sequence types tend to fare better where literals and I/O are not common usage. Therefore, if you find yourself working with general sequence types for strings, it is worth knowing how to convert from one form to the other. This is where the range-based constructors and functions, such as `insert`, fit in. Initialising a sequence from an arbitrary sequence of a different type is simple:

```
std::string text;
...
std::vector<char> vext(text.begin(), text.end());
```

The equivalent assignment can be effected in a similar fashion using an intermediate temporary object:

```
text = std::string(vext.begin(), vext.end());
```

Alternatively, the common range-based `assign` offers a more direct route:

```
text.assign(vext.begin(), vext.end());
```

Clearly, the underlying type you choose for your strings should be determined by usage requirements as well as performance and other such properties, rather than by blind orthodoxy—"I will only ever use `std::string`"—or a reactionary stance—"I will never use `std::string`". `std::string` should certainly be considered the default string type, but remember that defaults are there to be overridden when you know better.

String , set and match

Up until this point in the column, the focus has been on the range of expression afforded by combining orthogonal elements from the standard library or widely used libraries, such as SGI STL[6]. It is worth closing with a quick look not at how to provide a data structure that conforms to string expectations, but at how to write an algorithm that can be used freely across different string types. In the last column[2], I hinted that regular-expression pattern matching is not a capability that should be used as a criterion for specialising a new string type. Pattern matching is not an operation that should be tied to any one concrete string implementation; it is an independent idea that should be expressed independently.

If you are serious about regular-expression pattern matching you should consider using Regex++, the Boost regular expression library[7] from John Maddock. This is a fully featured, STL-based, POSIX regular-expression conforming library. If something smaller and simpler will do, supporting only a subset of the regular expression syntax, the short algorithm specified in *The Practice of Programming*[8] may be of interest. It supports matching against the beginning of a string (^), against its end ($), against any character (.), against zero-to-many repetitions of a character (*) and, of course, against characters directly.

Following in the style of many standard operations, the `regex_match` function template takes a pair of iterators to delimit its search text and a pair of iterators to define the regular expression string:

```
std::string text = "the cat sat on the mat";
std::string regex = "^t.*on";
bool found = regex_match(
                text.begin(), text.end(),
                regex.begin(), regex.end());
```

As with the original code, `regex_match` simply returns whether or not the search string matches the search expression, rather than returning the substring that matches—that variation is left as an exercise for the interested reader. Because it is so common to use null-terminated string constants to specify the search expressions, an overload of `regex_match` is provided for this purpose:

```
std::string text = "the cat sat on the mat";
bool found = regex_match(
                text.begin(), text.end(), "^t.*on");
```

The code for these two overloads is shown in listing 1. As you can see, there's not much to it: all the work is done by the `regex` helper class, shown in listing 2. The class holds a C++-refactored version of the original C code. The `regex` class is acting as a module—effectively, a `namespace` with a private section—rather than as a type describing instances. This is why the private details are kept `private` and the exposed `match` function is `static`.

Tangible benefits

The refactoring leaves the original control flow as it was, but makes the code follow STL style. This refactoring is far from gratuitous; conforming to the STL means that its usage model is well understood and its operation is freed from some unnecessary assumptions. The new version has the following tangible benefits:

- Search strings can contain non-terminating null characters, as can regular expressions. This is handy for searching binary data.
- Substrings can be searched without having to be copied, simply by passing in the relevant iterator range.
- The search string does not have to be `char *` based. The requirement is in terms of forward iterators.
- The character type does not have to be `char`. This means that not only are `signed char`, `unsigned char` and `wchar_t` acceptable, but also any user-defined character type for which comparisons to `'^'`, `'$'`, `'.'`, `'*'` and `'\0'` are supported. Note that, with the exception of `'$'`, the standard requires this

be true for `wchar_t`. In practical terms, however, you can expect it to be supported.

The requirement that `'\0'` be understood is relevant only to the overload of `regex_match` that takes a pointer for its regular expression. This must be able to find the terminating null of the string to pass through as the upper iterator bound. It cannot rely on the standard `strchr` function for support, because this works only for `char`-based strings. Instead, it uses a non-standard variant of `find` (`unguarded_find`[9], shown in listing 3) to search for a value that is known to be in the string—hence, the absence of an upper bound on the iteration.

Hopefully, this exploration of the generic approach to using and representing strings has shown that you can add to the meaningful set of operations applicable to a value type without modifying the encapsulated data structure, and you can provide or adopt alternative data structures without affecting or duplicating the algorithms that work on it. This is what is really meant by the terms 'orthogonal', 'loosely coupled' and 'extensible'—terms that are used often enough that they sometimes lose their real currency. The generic approach to value types should be contrasted with the limitations and complexity imposed by either a single-type-for-all solution[1] or an inheritance-based model[2]. ■

References

1. Kevlin Henney, "Stringing things along", *Application Development Advisor*, July–August 2002.
2. Kevlin Henney, "Highly strung", *Application Development Advisor*, September 2002.
3. Kevlin Henney, "If I had a hammer...", *Application Development Advisor*, July–August 2001.
4. Matthew Austern, "How to Do Case-Insensitive String Comparison", *C++ Report*, May 2000.
5. Scott Meyers, *Effective STL*, Addison-Wesley, 2001.
6. SGI Standard Template Library, www.sgi.com/tech/stl/
7. Boost C++ Libraries, http://boost.org
8. Brian W Kernighan and Rob Pike, *The Practice of Programming*, Addison-Wesley, 1999.
9. Matthew Austern, "Searching in the Standard Library", *C/C++ Users Journal online*, November 2001, www.cuj.com/experts/1911/austern.htm

Kevlin Henney is an independent software development consultant and trainer. He can be reached at www.curbralan.com.

Listing 1

```
template<typename text_iterator, typename regex_iterator>
bool regex_match(
    text_iterator text_begin, text_iterator text_end,
    regex_iterator regex_begin, regex_iterator regex_end)
{
    return regex::match(text_begin, text_end,
                        regex_begin, regex_end);
}

template<typename text_iterator, typename regex_char>
bool regex_match(
    text_iterator text_begin, text_iterator text_end,
    const regex_char *regex)
{
    return regex::match(text_begin, text_end,
                        regex, unguarded_find(regex, '\0'));
}
```

Listing 2

```
class regex
{
public:
    template<typename text_iterator, typename regex_iterator>
    static bool match(
        text_iterator text_begin, text_iterator text_end,
        regex_iterator regex_begin, regex_iterator regex_end)
    {
        if(*regex_begin == '^')
            return match_here(text_begin, text_end,
                              ++regex_begin, regex_end);
        for(;;)
        {
            if(match_here(text_begin, text_end,
```

```
                          regex_begin, regex_end))
                 return true;
             if(text_begin != text_end)
                 ++text_begin;
             else
                 return false;
         }
     }
private:
    template<typename text_iterator, typename regex_iterator>
    static bool match_here(
        text_iterator text_begin, text_iterator text_end,
        regex_iterator regex_begin, regex_iterator regex_end)
    {
        if(regex_begin == regex_end)
            return true;
        regex_iterator regex_next = regex_begin;
        if(*++regex_next == '*')
            return match_many(text_begin, text_end,
                              ++regex_next, regex_end,
                              *regex_begin);
        if(*regex_begin == '$' && regex_next == regex_end)
            return text_begin == text_end;
        if(text_begin != text_end &&
           (*regex_begin == '.' ||
            *regex_begin == *text_begin))
            return match_here(++text_begin, text_end,
                              regex_next, regex_end);
        return false;
    }

    template<
        typename text_iterator, typename regex_iterator,
        typename value_type>
    static bool match_many(
        text_iterator text_begin, text_iterator text_end,
        regex_iterator regex_begin, regex_iterator regex_end,
        const value_type &value)
    {
        do
        {
            if(match_here(text_begin, text_end,
                          regex_begin, regex_end))
                return true;
        }
        while(text_begin != text_end &&
              (*text_begin++ == value || value == '.'));
        return false;
    }
};
```

Listing 3

```
template<typename input_iterator, typename value_type>
input_iterator unguarded_find(
    input_iterator begin, const value_type &value)
{
    while(*begin != value)
        ++begin;
    return begin;
}
```