Kevlin Henney discusses the use of strings in C++, and finds that one type doesn't fit all

Stringing things along

HOW LONG IS A PIECE OF STRING? OR, TO be precise, what is a string? For C programmers the answer has always been easy: it is a null-terminated sequence of characters. This is lamentably still the answer of choice for some C++ programmers. For others working day-to-day with a specific proprietary or in-house library, it is possible that their answer may be the resident string class. C++ programmers educated in the standard library will respond that the there is a string type defined by the standard, and that that type is the one true answer for C++. But there is a much greater diversity than is suggested by these singular or orthodox answers.

OK, so what *is* a string? A reasonably high-level answer would be a sequence of characters. A slightly more precise answer would be a value that represents a sequence of characters. The shift in emphasis is a subtle but worthwhile one.

Why objects aren't equal

Objects live in a class-based society where equal rights are not conferred on all. Many programmers believe that all classes have the same basic rights as one another and that all should follow the same rules of development. For instance, they should all have functional copy constructors and assignment operators. Sometimes it's comforting to have a mechanical set of rules to follow, but it can also be a means of going wrong with confidence. While some guidelines can be mechanical and automatic, most of those concerned with the bigger design picture must be framed in more general terms to be of genuine long-term use[1, 2].

Returning to copying, does it make sense to support copying for all types? Copying strings seems like a useful idea, as does copying lists, monetary quantities, dates and so on. But what about copying objects that represent entities in the application domain, such as bank accounts or payment transactions? Copying a bank account is generally considered to be a criminal offence on this side of the keyboard.

What about copying objects that play an infrastructure role in programs, such as databases or resource wrappers? Remembering that in various situations temporary objects can be generated or elided at will by the compiler, is copying a whole database the kind of operation that is casual enough to support via something as transparent as a copy constructor? Considering the issue of resource wrappers, what does it mean to uniquely acquire a specific system resource...and then copy it? Such encapsulation was the basis of the `scoped` type, which was explored and defined in previous instalments of this column[3,4]. Copying was strictly forbidden because, regardless of whether or not it was technically achievable, it made no actual sense. All too often there is a temptation to "solve the solution" rather than "solve the problem".

When you look at objects this way it becomes clear that there are some fundamental differences between different kinds of objects, and that these differences go beyond the simple feature of copyability that I picked on for demonstration. It turns out that for most kinds of objects, copying is either a non-requirement or its absence is actually the requirement.

For value objects, copying is not only sensible—it is pretty much a basic requirement. Value objects represent fine-grained pieces of information

FACTS AT A GLANCE

- **Strings come in all shapes, sizes and programmers' preferences. They are value types that hold sequences of characters for appropriate definitions of value, character and sequence.**
- **Value object types are different from many other object types because they focus on fine-grained information rather than application entities. Therefore they are copyable, normally live on the heap or in other objects, and are not generally accessed by pointers.**
- **The core language offers C-style, null-terminated strings.**
- **The standard library offers a single string type. Unfortunately, it is by no means the best encapsulation. However, it is should still be used in preference to proprietary or lower-level solutions.**

in a system, such as numeric quantities or textual data. They can be manipulated, composed and combined freely, often with the intuitive assistance of operators. Subscript into a string `s` at index `i`? Use `s[i]`. Concatenate two strings, `a` and `b`, together? Take the small leap to consider concatenation to be a form of addition, and `a + b` follows. Values are passed around either by copy or by reference, but not by pointer. Why not by pointer? Because a pointer emphasises indirection and object identity, two things that are of little importance—and indeed act as cumbersome distractions—when working with value objects. Values live most comfortably on the stack or as data members of other objects, bounded to their enclosing scope's lifetime, rather than freely on the heap. Of course, their internal representation may live on the heap, but that detail should be self-contained[3] and hidden behind a veil of encapsulation.

OK, so now that we've understood that a string is some kind of a value, what are we offered in the standard? And how does that measure up against our expectations?

Copying a bank account is generally considered to be a criminal offence on this side of the keyboard

In the core language, inherited from C, we have a strongly reinforced convention that a string is a null-terminated array of characters. The character type can be either a `char` (conventionally ASCII or one of its relatives) or `wchar_t`, which may hold a suitable wide-character type (e.g. 32-bit Unicode) and which ranks as possibly the worst keyword ever to make it into a serious programming language. The null character has the integer value `0`, and is expressed most conveniently as `'\0'` or `char(0)` for `char`, or `L'\0'` or `wchar_t(0)` for `wchar_t`.

The death knell for NULL?

Note that the common C programmer mistake of using `NULL` for this purpose is just that: a mistake. In C, `NULL` is intended to denote the null pointer—not the null character—and depending on the platform, it may be defined as `0`, `0L` or `(void *) 0`. In C++ it is commonly accepted that `NULL` should not be used at all, because it can lead to surprising ambiguity during overload resolution. This is a point I will return to in a moment.

So how do we work with built-in strings? Clumsily. The only appropriate operator on offer is for subscripting, i.e. `string[index]`, and everything else must be done either by hand or by the `str` and `mem` function families defined in the C standard, e.g. `strlen` to find the length and `strcpy` to perform unmanaged copying. To define a general string value, you must either allocate suitable space dynamically on the heap (remembering to clear it up later) or declare an array of sufficient size. Neither of these two means of allocation allow convenient resizing, although heap-based allocation does at least allow flexibility in initial size and the opportunity to replace one string with another—assuming that your pointer bookkeeping is up to scratch.

Arrays cannot be copied by assignment, by copy construction, as return values or as function arguments. The first three cases are illegal and the last is sneakily substituted with a pointer. In other words,

```
void write(const char line[80]);
```

is rewritten by the compiler as:

```
void write(const char *line);
```

If you really want to ensure that an 80-character array is passed through, you have to resort to the syntactically challenging:

```
void write(const char (&line)[80]);
```

A point about pointers

This is hard-nosed about its requirement, because only a compile-time declared array of 80 characters will do. The upshot of all this is that, intentionally or otherwise, built-in strings are normally passed around as pointers, and are most commonly thought of this way. Pointers apparently behave as values for many purposes: they can be assigned and passed around freely, and they have some operator support. However, the copying results in aliasing and there is no management of the string in any way. There is also the small matter of null. Strings considered as pointers can refer either to a sequence of characters...or not. This extra out-of-band value sometimes sees action in expressing special circumstances, particularly exceptional outcomes. C++ has better mechanisms for signalling bad news, and the use of null to signal special *not applicable* values is an occasionally useful programming trick that receives too much airtime. Empty strings are often useful alternatives. However, an empty string is a string, whereas a null pointer is not.

The legitimacy of the null pointer value in the context of strings as pointers also gives us an unpleasant side effect. Given:

```
void write(const char *line);
void write(int);
```

The common macro definition of `NULL` as `0` means that the following code will pick out the second `write` function rather than the more obviously pointer-oriented first `write`:

```
write(NULL); // wrong write
```

So `NULL` is misleading as a pointer value, given that the following disambiguates the call:

```
write(static_cast<const char *>(0)); // right write
```

To sum up, C-style strings don't measure up at all well against our value type expectations. They can still be valuable in certain contexts, but marks out of ten for general-purpose use hug the lower end of the scale as well as the lower layers of abstraction.

The standard provides the `std::basic_string` class template for all your string needs—or at least, that is the intent. The main motivation for templating is to capture the different possible character types in a single body of code: strings of `char`, `wchar_t` or a character type of your own invention all use the same implementation code. A couple of standard `typedef`s offer you the most commonly used names for working with `std::basic_string`:

```
namespace std
{
    typedef basic_string<char> string;
    typedef basic_string<wchar_t> wstring;
}
```

What `basic_string` does well is to act as an encapsulated type from the perspective of memory management. On the other hand, it is mired in convenience. It has lots of little features that are thought to be convenient, but the general effect of including them all is counterproductive. The redundancy and complexity starts before we even reach the `class` keyword in its definition:

```
namespace std
{
    template
    <
        typename char_type,
        typename traits = char_traits<char_type>,
        allocator_type = allocator<char_type>
    >
    class basic_string
    {
        ...
    };
}
```

Using your own traits

Not content simply with offering parameterisable variation on character type, you can, if you wish, provide your own set of traits for working with your selected character type[5]. At first it looks like the second parameter could be used for dealing with differences between character sets, i.e. a useful hook for addressing internationalisation. However, not only is it not that useful, but a template parameter on the core string type is a long way from being the best way to deal with internationalisation issues such as locale-sensitive string comparison. The second parameter offers a set of unrelated facilities, some of which are focused on character values on its own, some of which are focused on memory manipulation of character type sequences, and the remainder of which are focused on I/O. For a sensibly defined character type, the policy parameter offers no useful extras; for poorly defined character types, the solution is to fix the character type rather than shore it up with a policy parameter.

Adding to the syntax soup is the third parameter: the allocator type. This wart can be found on all of the standard container classes. Allocators were an attempt to parameterise the memory model used for allocating container elements, prompted originally by historical shortcomings of the amateurish DOS memory model. Originally it was hoped that the idea of generalising memory models would afford transparent access to persistent storage or shared memory representations. As the process of standardisation nailed down the semantics of allocators, their complexity increased and their generality decreased. We now know that they are useful for remarkably little[6], and probably even less than that[7].

Fortunately, the second and third parameters are defaulted, which means that out of three degrees of parameterisation, only one is of any use. But you can at least ignore them in most cases. They represent failed experiments in generality and policy-based design. Such gratuitous but limiting overgeneralisation is also found in the function members offered in the `basic_string` interface.

The most obvious example is with functions that search for a given character or substring, of which there are twenty-four in `basic_string`. You might think, given that number, that all your searching needs would be covered. You would be wrong. Those twenty-four member functions offer less behaviour than the fourteen searching algorithms defined in the standard `<algorithm>` header. In other words, `basic_string` duplicates existing extensible functionality, but without the extensibility or functionality.

Jekyll, meet Hyde

Another feature that you may notice if you look at the interface is that `basic_string` seems to be caught between two interface styles: on the one hand, there are a lot of functions that work in terms of numeric indices and on the other, there are a lot of familiar STL-based functions that work in terms of iterators. This Jekyll and Hyde interface is a historical artefact that reflects the traditional and index-based style of interface first adopted for strings, and the later incorporation of STL functionality to make strings look like STL sequences. Given the choice, focus on the consistency, expressiveness and extensibility of the STL approach rather than the other more limited functions.

The desire to please all of the people all of the time manifests itself even in something as innocuous as subscripting. The intuitive way to subscript an index is using `operator[]`, but there is also an `at` function that adds a difference and a twist: access is range-checked with an exception, whereas there is no requirement

Given the choice, focus on the consistency, expressiveness and extensibility of the STL approach rather than the other more limited functions

that `operator[]` must result in defined behaviour for any out-of-bounds access. What `at` considers to be out of bounds is subtly different to what `operator[]` considers to be out of bounds—there's nothing like consistent design...and that is nothing like it at all.

So, what is the design objective of `at`? I'm not sure that I can state it in a way that is consistent with the way that the more natural `operator[]` is used and how the rest of the interface supported by STL sequences is expressed. If the aim is to provide an option for range-checked access, that is certainly laudable, but why is this feature provided only for indexing and not for iteration? The inconsistency in the design seems to dilute its intention somewhat. If it is a genuine design goal, then it should be reflected in the interface design as whole. As it is, the quality-of-failure offered by at is limited at best and, at worst, wasted.

Why 'at' isn't good enough

Consider how such a function might be used: the idiomatic approach to indexing is to use `operator[]`. To do anything else takes a conscious effort to adopt an alternative. Once that conscious effort is made, the problem that `at` attempts to address has already been solved. The standard documents the intent of such exceptions as, "The distinguishing characteristic of logic errors is that they are due to errors in the internal logic of the program. In theory, they are preventable." Once you make the decision to use `at`, you are at the same doorstep as making the problem (and the use of `at`) preventable. It is not whether or not you should have range checking, but whether it makes any sense to have both checked and

unchecked options available in the same class.

The train of thought that leads to using `at` also leads away from it—"I had better use `at` instead of `operator[]` because I might index out of range...oh, OK, now that I know that, I will just check and fix the code so that mistake doesn't happen." This reminds me of a set of macros I once saw intended to make C programming easier for Fortran programmers (including `IF, THEN` and so on). There was a definition for `ONE` that was `0 (!)`. It was supposed to allow for loops to be written using traditional Fortran one-based indexing rather than C's zero-based indexing. The user was supposed to make the effort to write `ONE` instead of `1`... by which point they had already remembered enough to write `0` instead!

It is not that I don't believe in `at` because I have a belief in unsafe code; I don't believe in `at` because it doesn't work to make code safe. It tries to address a broad quality-of-failure configuration issue with a single member function, which is definitely the wrong hammer for the screws it is trying to drive in. Programmers who believe that `at` will make their program safer and more secure probably do not understand enough about safety and security to make that stick.

I may appear to be coming down unnecessarily hard on `std::basic_string`. After all, I use it all the time and advocate it frequently[8], and it satisfies the need for self-containedness[1] one expects of a value object type. My concern is that outside of a subset of its features, it is quite a poor design: it leads by example from the rear, not the front. The failed ambition to please all comers has resulted in something of a dog's dinner of an interface. Additionally, it embodies the flawed assumption that the diverse needs of different string users can be accommodated in one type. The design starts off in the wrong place, and goes downhill from there. So, in summary, use `std::string` and `std::wstring` in preference to raw C-style strings and proprietary string types, but don't use `std::basic_string` as a model example of design. ■

References

1. Kevlin Henney, "Six of the best", *Application Development Advisor*, May 2002.
2. Kevlin Henney, "Half a dozen of the other", *Application Development Advisor*, June 2002.
3. Kevlin Henney, "Making an exception", *Application Development Advisor*, May 2001.
4. Kevlin Henney, "One careful owner", *Application Development Advisor*, June 2001.
5. Kevlin Henney, "Flag waiving", *Application Development Advisor*, April 2002.
6. Scott Meyers, *Effective STL*, Addison-Wesley, 2001.
7. Kevlin Henney, email to Scott Meyers detailing why allocators and shared memory do not mix, now listed in the errata for *Effective STL*, *www.aristeia.com*.
8. Kevlin Henney, "The miseducation of C++", *Application Development Advisor*, April 2001

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