Kevlin Henney continues his examination of strings this issue, tackling the challenge of writing his own string class

Highly Strung

THE PREVIOUS COLUMN[1] PROVIDED A working definition of a string: a value that represents a sequence of characters. It also provided a critique of the two basic string types clearly offered by the C++ standard:

1. C-style strings are in the core language. They are null-terminated arrays of characters. They are also tedious and error-prone to work with.
2. The `std::basic_string` class template, with its more public facing `std::string` and `std::wstring` typedefs. It is a self-contained type whose instances can contain embedded nulls. On the downside, it suffers from an indecisive interface that cannot make up its mind what it is trying to be, other than all things to all people. This results in a penalty rather than a scored goal.

History should have taught us that it is unlikely that a single encapsulated string type will fit all developers' needs: writing your own string class used to be a popular activity. It is not as trivial as it might seem, because the two sides to the story—"What does the interface look like?" and "How will it be implemented?"—can each have a variety of different endings.

The House of the Rising String

Writing a string class or two is still a worthwhile exercise, and one that all C++ programmers should tackle at one time or another. I do not advocate this in order to reinvent the wheel, gratuitously ignoring existing standard classes, but because it is a good C++ workout: interface design, value type concepts, operator overloading, copying, conversion implementation and prevention, representation design, memory management and so on. It is a healthy run through C++'s features and how to employ them in a design. Just as anyone learning the guitar will inevitably learn *House of the Rising Sun*—it's been done before—and anyone learning woodwork will often construct a small footstool—it's been done before—you do it for the practice and the exercise, not necessarily for the originality or utility.

There are a couple of conclusions that you should be able to draw. First, designing an effective interface is not as trivial as it first appears ("This should be no problem, we all know what a string is..."), and second, it is unlikely that one type, either from the interface or the implementation perspective, will satisfy all needs.

The second point leads to two different approaches: either design a type that tries to be everything to all people, or establish an extensible design that accommodates most of the variation that developers are likely to need. Where the latter option is effectively a framework, the former often ends up a patchwork. You stitch together lots of special cases, but there are inevitably holes.

One or two minor problems...

So how do you create a string framework? It is easy to assume that you'll use inheritance and class hierarchies in a framework, but you'd be mistaken: inheritance is the wrong tool for the job. The previous column[1] emphasised that different object types follow different rules for their design, and one of the conventions that value types follow is that they do not generally or successfully live in class hierarchies.

String hierarchies have surface appeal, but are deeply troublesome in practice. For example, consider the following fragment of a possible

FACTS AT A GLANCE

- **Designing value types often seems trivial until you try it, especially something like the common string.**
- **Supporting different implementations and capabilities through inheritance is a not a good match for value types. The resulting contortions are often quite painful.**
- **Variation and flexibility for value types is most simply supported through generic programming.**
- **STL-based sequences, in combination with STL-based algorithms, often provide a simpler and more effective approach to string representation and manipulation.**

interface class for strings of char:

```
class string
{
public:
    virtual ~string();
    virtual string &operator+=(const string &) = 0;
    virtual bool empty() const = 0;
    virtual char &operator[](std::size_t) = 0;
    ...
};
```

Against this, a library can provide different implementations with different performance or space tradeoffs, e.g. a string bounded to a fixed-upper limit. An implementation could also provide extended interfaces, e.g. a string that supports pattern matching:

```
template<std::size_t max_length>
class bounded_string : public string
{
    ...
private:
    char data[max_length + 1];
};

class regexable_string : public string
{
public:
    iterator begin(const char *pattern);
    ...
};
```

At first it seems that string provides an easy-to-use common interface to such different implementation types:

```
void concatenate(string &lhs, const string &rhs)
{
    lhs += rhs;
}
```

This works uniformly with independent string implementations:

```
bounded_string<80> bound = "an";
regexable_string match = "b";
concatenate(bound, bound);
concatenate(match, bound);
```

But what about conversions? As a value type, it makes sense to support a safely implicit conversion from string literal types into the required string object type. The problem is, what is the required string object type? The following will not compile:

```
concatenate(match, "a");
```

The string literal, "a", is to all intents and purposes seen to be of type `const char *`. However, concatenate expects a `const string &` or something that can be converted to a string. Such an implicit conversion requires an executable converting constructor—one that can take a single argument and is not tagged explicit—which string cannot offer because it is an abstract class, and which by definition cannot be instantiated.

This problem manifests itself again with return types. It is reasonable to expect `operator+` to be available for concatenating strings. Because string is abstract, the following won't work. String is required to play the decidedly concrete roles of being a local variable type and a return type:

```
string operator+(const string &lhs, const string &rhs)
{
    string result = lhs;
    result += rhs;
    return result;
}
```

Making `string` concrete is not the solution: this is a hack that makes a mockery of the idea of introducing a class hierarchy in the first place. It means that, far from being an interface, it will offer a default implementation that will typically be ignored in derived classes. This means that there will be a uniquely favoured implementation in the hierarchy. Now what should that default representation be? Given that we were trying to escape the idea that one implementation would be favoured over another, such an approach would not be a successful demonstration of our design skill. Inheriting from a concrete implementation, only to ignore it, is a poor use of inheritance and a good source of problems[2,3].

Another approach

Let us say that we turn back from this blind alley and try another approach: we will favour the left-hand side and choose its type as the underlying type of the result. However, as we are only working through the string interface, and won't know the actual type of the left-hand side at compile time, we will need to introduce some form of polymorphic copying. The Virtual Copy Constructor idiom[3,4,5] provides a solution to this problem:

```
class string
{
public:
    ...
    virtual string *clone() const = 0;
    ...
};

class regexable_string : public string
{
public:
    regexable_string(const regexable_string &);
    virtual string *clone() const
    {
        return new regexable_string(*this);
    }
    ...
};
```

And likewise for other descendants of string. You will notice that you are now working heap memory and pointers, something that was flagged to be something of a "no-no" for value types[1]. If you harboured any doubts, you're now about to find out why this recommendation exists:

```
string *operator+(const string &lhs, const string &rhs)
{
    string *result = lhs.clone();
    *result += rhs;
    return result;
}
```

Ugh. When you start mixing different levels of indirection to the same type, that's the code telling you something is wrong with the design. This introduces obvious inconsistencies into the code: add two self-contained string values together and you get a pointer to a string object allocated on the heap that you must now manage. Working through pointers to get to values means that operator overloading defeats the transparent syntactic benefit they were supposed to offer: you always have to dereference before using them. The function now imposes ungainly call syntax and a memory management burden on any caller, so the string hierarchy is no longer self-contained and does not support symmetric acquisition and release[2]:

```
string *result = match + bound;
(*result)[0] = 'B';
...
delete result;
```

Deciding to skip the visible level of indirection on picking up the return is effectively an open invitation to memory leaks:

```
{
    string &result = *(match + bound);
    ...
} // oops, memory leak
```

And there are programmers who are not content with such obvious memory leaks: they wish to brush them under the carpet, pretend they're not there and make them even harder to find. Depending on the intent and sensibility of the author, the following code is either incompetence or deceit:

```
string &operator+(const string &lhs, const string &rhs)
{
    string &result = *lhs.clone();
    result += rhs;
    return result;
}
```

Never, ever, do this. Do not return dynamically allocated objects via references if the caller is obliged to deallocate the object. References are supposed to make working with a level of indirection transparent. They emit an idiomatic and subliminal message: "Don't worry about ownership or lifetime management, it's all taken care of, just use the object. Go ahead, you know you want to." Don't work against this deeply rooted assumption: references are not like pointers; that is just common mythology. If you have written such code, go and fix it; if someone else has written such code, tell them to fix it.

Putting up with pointers

So, taking stock, we seem to be stuck with pointers. How do make sure that we avoid memory leaks? You can use `std::auto_ptr` or `scoped`[6] to pick up the result:

```
std::auto_ptr<string> result(match + bound);
```

However, we can go one step further with a more belt-and-braces approach. Ensure that the return value from `operator+` looks after itself: use objects to automate from the moment they're released[7]. You can transfer ownership either using `std::auto_ptr` or `scoped`'s more explicit transfer feature[8]:

```
std::auto_ptr<string> operator+(
    const string &, const string &);
```

Yes, this is a technical solution––just—but it is neither convenient nor elegant. In terms of usability and other objectives, it is a dismal design failure.

And this set of creational problems is just the tip of the iceberg. How do you provide iterators at the interface level if the concrete iterator depends on the concrete type? How do you provide for subscript operators that check the result of any assignment that is returned, e.g. to prevent assignment of null characters for null-terminated representations? How do you 'unsupport' operations that the interface commits its descendants to when it is realised that the interface is too general for some implementations, e.g. read-only strings do not support non-`const` operations?

Each one of these problems is technically soluble, as are the ones that I have chosen not to list. I could outline the specific solutions to you, but ultimately I would only convince you that to write a decent string class framework based on inheritance involves far more design effort than the essential problem warrants. The problems with using plain `char *` seem comparatively minor.

Genericity and generativity

When I said that the code was trying to tell you something was wrong with the design, I did not mean fix it on a per-function or per-problem basis: I meant scrap it and start again. To support different underlying implementations and different specific interfaces, while supporting a common subset of operations across value types...such a major engineering effort should not be necessary.

Many programmers persist in tackling the problem at the wrong level, considering it to be a localised, function-by-function issue, and with the wrong tools—inheritance and pointer gymnastics. I am reminded of when I was a student, in a shared house where bits of the plumbing were made of lead. One day a pipe in the kitchen sprang a leak. It was a slight leak at first, and we were in two minds as to whether to get a plumber out then (at 24-hour call-out rate) or the next day. My housemate unfortunately decided the issue when no one else was in the room: he put a tack in the pipe. "I thought it would just plug the hole... not make the hole bigger," he said as the two of us stood in the rising swamp that covered what was once the kitchen floor. I grabbed the yellow pages, and ultimately developed a healthy paranoia about leaks that has kept my programs in good stead ever since. My housemate went on to do a PhD in physics, before then becoming a programmer.

I have seen similar problems recur for value types (often not recognised as such, which is perhaps half the problem) with such frequency that it is little wonder a lot of C++ legacy code sends programmers running to the job pages. No major design effort is required to fix these problems, just a shift in perspective. It is possible to achieve an open, extensible, liberal solution far more easily with generic programming. Generic programming is based on the use of compile-time polymorphism (principally, overloading and templates[9]) with value types. Only concrete types that support copying are used. Commonality of interface is defined in terms of how a type can be used, rather than its parentage. Inheritance is not used for this purpose. Any commonality of implementation is the private affair of a given type, i.e. it can choose to use inheritance or delegation, but it is not the concern of the programming model.

Not just about templates

It is worth clarifying that generic programming is not simply using templates. Generic programming, more precisely, separates non-primitive tasks from encapsulated, collective data structures: the data structures need not be templated, although they often are, e.g. `std::vector`. But it is the tasks that are templated as algorithmically oriented function templates; iterators form the bridge between the two disjointed worlds of data structures and algorithms.

As an example of such a separation, consider the idea of the pattern-matching string, `regexable_string`, mentioned earlier. Why is this type special? Why does it have the ability to be searched using regular expressions, but not the `bounded_string` or `std::basic_string` class templates? The answer is that there is nothing that sets it apart like this. Where `bounded_string` represents an implementation, `regexable_string` offers a particular facility that could apply equally well to other string types. This partitioning is founded on the wrong criteria but is, alas, common in many OO designs. It often leads to unnecessary code duplication or multiple inheritance contortions. The generic programming approach expresses independent ideas independently: pattern matching is orthogonal to the specific type of a string, so it is achieved through separate types or functions, accommodating different string implementations without being coupled to any one of them.

Templates can also be employed for the policy-based[5,10,11] parameterisation of data structures, allowing the control of different points of variation in the representation or execution of the data structure. This application is more the preserve of *generative programming*[12] than it is of generic programming. Although closely related, they are distinct, with separate aims and consequences.

One consequence of generative programming is that it is all too easy to get carried away with the generalisation afforded by the style, introducing a new policy parameter or trait for every conceivable degree of freedom you may ever and never wish for. When I hinted at a more open and liberal approach to strings, a single über-string type parameterised beyond either belief or comprehension is not what I had in mind. Such overgeneralisation can render the code either unusable or at least uncomfortable to use for the majority, which leads to the same thing: shelfware. Often the most effective designs are the simplest, so when identifying scope for generalisation, the degrees of freedom should be grounded in reality rather than fancy.

Variance and standard deviation

To return to where we started, the standard offers potentially many types that fit the description of "a value that represents a sequence of characters". Inside `std::basic_string` is a small type struggling to get out. And what does this type look like? Well, it shares a common interface with other STL sequences, such as `std::vector`, and excludes the index-based operations and arbitrary constraints that were supposed to help optimisation. For instance, `std::basic_string` is supposed to allow reference-counted implementations as an optimisation. The compromise on its semantics are such that it can just about support reference counting, but more often as a 'pessimisation' than an optimisation[13].

Equating strings to STL sequences of characters may at first seem to offer limited behaviour. But remember that the data structure should focus on offering primitive operations, such as property queries and concatenation, while the more sophisticated behaviour is contracted out to separate algorithms that work on iterators rather than indices.

So, in addition to `std::string`, what other types will support the STL-centric view of strings? If you require strings that use contiguous memory, and a guarantee that they do not indulge in wasteful reference-counting tricks, you may find that `std::vector<char>` fits your needs perfectly well. If you are creating large strings that should be reasonably efficient in their use of memory, but for which contiguous memory is not a strong requirement, `std::deque<char>` may fit the bill. If you require a type that supports large strings and efficient whole-string operations, you may wish to consider SGI's `rope` class template[14], an STL-based string type for heavy duty use. Alternatively, you may decide to create your own custom type to address a particular problem. Because the required sequence interface is so small, creating a new type is less laborious than trying to implement something like `std::basic_string`.

And what about common string operations? The standard `<algorithm>` header offers you many function templates for searching–e.g. `std::find` for single characters and `std::search` for substrings–and modification–e.g. `std::replace` to replace a particular character value with another or `std::remove` to drop a particular value. It is also straightforward for you to provide your own task-specific algorithms independently of any specific type. In conclusion, the effective design and use of strings is not rocket science, once you find the right trajectory. ■

References

1. Kevlin Henney, "Stringing things along", *Application Development Advisor*, July–August 2002
2. Kevlin Henney, "Six of the best", *Application Development Advisor*, May 2002
3. Scott Meyers, *More Effective C++*, Addison-Wesley, 1996.
4. James O Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992.
5. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns*, Addison-Wesley, 1995.
6. Kevlin Henney, "Making an exception", *Application Development Advisor*, May 2001
7. Kevlin Henney, "The rest of the best", *Application Development Advisor*, June 2002
8. Kevlin Henney, "One careful owner", *Application Development Advisor*, June 2001
9. Kevlin Henney, "Promoting polymorphism", *Application Development Advisor*, October 2001
10. Andrei Alexandrescu, *Modern C++ Design*, Addison-Wesley, 2001
11. Kevlin Henney, "Flag waiving", *Application Development Advisor*, April 2002
12. Krzysztof Czarnecki and Ulrich W Eisenecker, *Generative Programming*, Addison-Wesley, 2000
13. Kevlin Henney, "Distinctly Qualified", *C/C++ Users Journal* online, May 2001, www.cuj.com/experts/1905/henney.htm
14. SGI Standard Template Library, www.sgi.com/tech/stl/

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