Polymorphism is one of the most difficult OO concepts for beginners to understand, and the fact that there are four types of it in C++ doesn't help. **Kevlin Henney** cuts through the whole tangled mess

Promoting polymorphism

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

"Polywhat?"

"Polymorphism. It's Greek – means *many shapes*."

"Right... but what's that got to do with objects? Other than, of course, shapes."

"Not a lot, actually, but what it really means is *dynamic dispatch*."

"And what's that in Greek?"

"Er, dunno... but that's not important. If you really want to understand OO..."

MANY PROGRAMMERS FEEL BEWILDERED TO say the least when introduced to polymorphism. A colleague, course or book has been making reasonable progress teaching the other core OO concepts, such as encapsulation and inheritance, and then polymorphism strikes. A smokescreen of statements that are true but not useful is thrown up – for instance, the transliteration from Greek. The topic is then either left to one side or demonstrated through examples which, while enlightening, are often unrelated to the terminology they were intended to illuminate.

Polymorphism is both more important and more general than many OO introductions make out. It's not about inheritance, so it doesn't logically follow it in the introduction of the *Big Three of OO* – encapsulation, then inheritance, then polymorphism. Inheritance is the least important, its premature introduction often confusing key issues such as "what is a type?" and "what is a subtype?" These questions are particularly important when dealing with C++.

A type tells you which operations can be performed on an object. They may or may not be the same as the set of members in a class, and they may or may not be the same as the compiler's direct notion of type as a source-declared name. You can relate types with respect to one another according to the operations they support (more or fewer, with stronger or weaker semantics). This relationship is subtyping and supertyping, which may or may not relate to a class hierarchy.

Polymorphism allows you to write functions that work uniformly with different types, and the different notions of what a type is give rise to different forms of polymorphism – four, to be precise.[1] They are all supported by C++:

- *Inclusion polymorphism*: This is the form most familiar to OO programmers, expressed as it is in C++ through inheritance and virtual functions.
- *Parametric polymorphism*: The use of a type parameter gives rise to a different concept of type conformance and substitutability. Class and function templates deliver the foundations of generic programming.
- *Overloading polymorphism*: This is the use of a common function name, including symbolic names in the case of operators, to refer to different actual functions that are distinguished with respect to the type and number of their arguments.
- *Coercion polymorphism*: A value of one type may be converted to a value of another. The context of use determines whether a conversion is required to fulfil a particular operation, and the value's type determines whether it is possible implicitly. C++ supports implicit conversions between many of its built-in types, such as from int to double, and allows programmers a similar privilege for their own types, through converting constructors and user-defined conversion operators.

Polymorphism isn't strictly either a dynamic dispatch or a class-related concept. It is clear that in its many

FACTS AT A GLANCE

- **There are four types of polymorphism that are relevant to the C++ developer: parametric, inclusion, overloading and coercion.**
- **C APIs to data sources can often be wrapped up with stream objects.**
- **Stream access can often be wrapped up with input iterators.**
- **A lot of fine-grained C++ design revolves around the appropriate application of the appropriate polymorphism.**

forms, it offers a powerful means of framing and articulating designs. The use and availability of a particular polymorphism affects style, most notably giving rise to classic object orientation in the first instance and generic programming in the next three, and all four may be unified under the heading of modern object-oriented programming.

Of course, the use of *a style* does not guarantee the presence of *style*, per se. That is a matter of good taste and design and, in the case of polymorphism, is best governed by the principle of substitutability[2]. The remainder of this article touches on the non-inclusion varieties of polymorphism at work in a simple problem and, at a more concrete level, demonstrates what is involved in writing your own iterator types to work smoothly with the standard library.

Something to wrap

Let's tackle a simple but sometimes useful programming task: listing the names of files in a given directory. The facility for this is provided as part of the API for many operating systems, Win32 and Unix included. Such APIs are normally provided in C. Consequently, an easy trap to fall into is also to write the usage code in a C style. The following simple program uses the standard Posix functions and types to list the files in the current directory:

```
#include <dirent.h>
#include <iostream>
int main()
{
   DIR *dir = opendir(".");
   while(dirent *entry = readdir(dir))
      std::cout << entry->d_name << std::endl;
   closedir(dir);
   return 0;
}
```

There are a few things that give away that this is a C++ rather than a C program:

- Most obviously, there is the use of the standard C++ I/O stream library.
- entry is declared as a pointer to dirent as opposed to struct dirent, because C++ does not require the use of the struct keyword to identify the type name. Indeed, the received C++ style discourages it.
- Less subtle than the omission of struct is the use of a declaration in a condition. Condition declarations work in terms of bool or anything that is implicitly convertible to it, such as a pointer. entry exists, and has scope, only for the loop. It is initialised on each iteration and, if the result is non-null, the loop body is executed. readdir returns null on end of stream or error, which causes the loop to end.

Even in this short space, the structure of the program is still being dictated by a C mindset. I'm not referring to the absence of any class definitions. A resource is being acquired and released explicitly through the use of global functions, in spite of the error-prone and tedious nature of this task. Such paired actions are common enough that we can wrap them up, making for briefer and safer code. This problem was explored in previous columns[3,4], with the scoped class template presented as a solution. Here it is adapted for use in the example:

```
struct close_dir
{
   void operator()(DIR *to_close)
   {
      closedir(dir);
   }
};

int main()
{
   scoped<DIR *, close_dir> dir(opendir("."));
   while(dirent *entry = readdir(dir))
      std::cout << entry->d_name << std::endl;
   return 0;
}
```

Now, let's get serious. What has this really bought us? We've just moved the code around a bit, adding a new type and reducing the mainline by a single statement. Not exactly a big enough saving or point of style to get excited about, although it's certainly safer than before.

Stepping into the stream

It's tempting to extend scoped to embrace initialisation in the same way that finalisation is currently parameterised. However, this is more often the path of diminishing returns than the road to improved abstraction. Stephen Hawking received a comment from his publisher during the writing of *A Brief History of Time*, to the effect that every equation he included in the book would halve the number of sales, so he included only "E = mc²". A similar point has sometimes been observed about templates: each additional generalising parameter will decrease the number of users by half. scoped is fine as it is, and further attempts to make it more useful it will typically decrease its utility.

What we need to make the code clearer and simpler is a named type. This doesn't need to be very complex: it needs only to mirror the basic functionality of <dirent.h>. The opaque DIR type represents a stream-like abstraction that, for our purposes, delivers a string on each read. Following the relationship between the C I/O FILE type and the C++ I/O streams, we can provide a simple Wrapper Facade[5] that offers a portable interface for directories:

```
class dir_stream
{
public:
   explicit dir_stream(const std::string &);
   ~dir_stream();
   operator const void *() const;
   dir_stream &operator>>(const std::string &);
private:
   dir_stream(const dir_stream &);
   dir_stream &operator=(const dir_stream &);
   ...
};
```

Here's the example revisited to show the dir_stream class in use:

```
int main()
{
   dir_stream dir(".");
   for(std::string entry; dir >> entry;)
      std::cout << entry << std::endl;
   return 0;
}
```

The overloaded stream extraction operator, operator>>, offers a comfortable and familiar syntax for pulling the next filename from the open directory stream. The other form of polymorphism at play in this example is coercion: a dir_stream instance can be treated

as a Boolean, resulting in a truth value when it's OK and falsehood on end of stream or error. This means the streaming operation can be incorporated into a loop continuation check.

Note, however, that the user-defined conversion operator results in const void * rather than bool. The native C++ bool type is essentially a degenerate int type, which means that if you use it in an arithmetic expression it will cheerfully compile. Because such runtime behaviour is neither useful nor meaningful, it seems best to nip this one in the bud and catch it at compile time. A const void * represents perhaps a better Boolean type than bool, allowing only truthhood, negation and – less desirably – comparison with pointer types. An added benefit of const void * over bool is that it prevents indulgence in a particularly pointless expression favoured by those uncomfortable with logic (a worrying sign in a programmer): dir == true will not compile.

In contrast with the Boolean conversion, the constructor is declared explicit. This prevents implicit conversions from strings to dir_streams – such polymorphism wouldn't make sense.

The only other interface feature to draw attention to in dir_stream is that it cannot be copied. Copying isn't meaningful for such a type and is neither trivial to interpret or to implement. So, although the copy constructor and copy assignment operator are declared private, this capability (or its absence) is a published feature that should be listed along with the more conventional public functions when describing the interface.

The implementation of dir_stream could use scoped, but for no loss of clarity and little loss of brevity, and also to ensure that the code presented in this article stands on its own. It is shown here in terms of the raw Posix interface:

```
class dir_stream
{
public:
   explicit dir_stream(const std::string &dir_name)
    : handle(opendir(dir_name.c_str()))
   {
   }
   ~dir_stream()
   {
      close();
   }
   operator const void *() const
   {
      return handle;
   }
   dir_stream &operator>>(std::string &rhs)
   {
      if(dirent *entry = readdir(handle))
         rhs = entry->d_name;
      else
         close();
      return *this;
   }
private:
   dir_stream(const dir_stream &);
   dir_stream &operator=(const dir_stream &);
   void close()
   {
      if(handle)
      {
         closedir(handle);
         handle = 0;
      }
   }
   DIR *handle;
};
```

If you're working under Win32 rather than a Posix-conforming platform, you have two choices: translate the implementation code to use Win32 API functions or use an API wrapper layer[6].

Many streams to cross

We can make directory listing slightly more interesting, and the result more useful, by considering how to list the contents of named directories:

```
int main(int argc, char *argv[])
{
   if(argc == 1)
   {
      dir_stream dir(".");
      for(std::string entry; dir >> entry;)
         std::cout << entry << std::endl;
   }
   else
   {
      for(int arg = 1; arg != argc; ++arg)
      {
         dir_stream dir(argv[arg]);
         for(std::string entry; dir >> entry;)
            std::cout << entry << std::endl;
      }
   }
   return 0;
}
```

This program will list the files in the named directories, otherwise just the files in the current directory if none are named. It's a little light on error handling, which is intentional but fixable. However, in the context of this article, there's still something of C and cut-and-paste styles at work here. The following code introduces better argument handling and decomposition:

```
void list_dir(const std::string &name)
{
   dir_stream dir(name);
   for(std::string entry; dir >> entry;)
      std::cout << entry << std::endl;
}

int main(int argc, char *argv[])
{
   std::vector<std::string> args(argv + 1, argv + argc);
   if(args.empty())
      args.push_back(".");
   std::for_each(args.begin(), args.end(), list_dir);
   return 0;
}
```

The program arguments of interest – so excluding the program name in argv[0] – can often be more easily manipulated as a container of strings than as an array of char *.

Many loops with common intent can be refactored and replaced by standard algorithm functions, rather than being repeated in source code. Even without any additional loop abstraction, it makes economic sense to factor out the common directory listing code into a separate function, list_dir. Combining this with the standard for_each algorithm reduces the program to its essential logic. For each argument, list the files in the directory of that name.

Rolling your own iterator

If you want to do something other than print out the names you're streaming in, you have to rewrite the content of the inner loop. For instance, to populate a container, you would replace the function with a function-object type that held a reference to the target container and had an operator() that populated it, using insert or push_back as necessary.

To stay on message and refactor the loop that pulls the file names in from the stream, you need an iterator. One does not exist (yet) but, given the relationship between std::istream and std::istream_iterator, it's easy to imagine what it would look like:

```
void list_dir(const std::string &name)
{
   typedef std::ostream_iterator<std::string> out;
   dir_stream dir(name);
   dir_iterator begin(dir), end;
   std::copy(begin, end, out(std::cout, "\n"));
}
```

Or, more functionally:

```
void list_dir(const std::string &name)
{
   typedef std::ostream_iterator<std::string> out;
   std::copy(
      dir_iterator(dir_stream(name)), dir_iterator(),
      out(std::cout, "\n"));
}
```

Or, in sorted order:

```
void list_dir(const std::string &name)
{
   typedef std::ostream_iterator<std::string> out;
   std::set<std::string> sorted(
      dir_iterator(dir_stream(name)), dir_iterator());
   std::copy(
      sorted.begin(), sorted.end(),
      out(std::cout, "\n"));
}
```

Rather than labouring the point of a loop – a concept with which many developers are already familiar – these variations takes a more declarative approach to describing the solution. An iterator can wrap access to a directory stream and this can be traversed to its end, indicated by a default-constructed iterator. This iterator can then be used in the same context as other iterators, such as in a for loop, as arguments to a standard algorithm function or to initialise a container.

Parametric polymorphism

This brings us to parametric polymorphism. We have seen a little syntax for using a dir_iterator, in as much as seeing its construction and destruction capabilities, but what else is required of this type to make it work with the std::set constructor and the std::copy function template? Iterators are pointer-like objects that decouple iteration from their target. The basic Iterator pattern[7] captures this separation, but the pointer semantics are a C++-specific idiom, fitting the general design pattern more appropriately into its context. However, not all iterators can, or should, support a full pointer interface:

- *Input iterators* are for single-pass input. They support copying, pre- and post-increment operators, dereferencing only for reading and basic equality testing.
- *Output iterators* are for single-pass output. Their operations are similar to input iterators, except that they support dereferencing only for writing.
- *Forward iterators* may be used in multi-pass algorithms and support dereferencing for both reading and writing purposes.
- *Bidirectional iterators* additionally support pre- and post-decrement operators.
- *Random access iterators* support full-blown pointer syntax, including pointer arithmetic and related operations.

These five categories each define a type – a set of operations that must be supported on an object. They are also related as subtypes. A forward iterator is both an input iterator an output iterator; a bidirectional iterator is a forward iterator, plus extra; a random access iterator is a bidirectional iterator, plus extra.

The std::set constructor requires that its arguments are at least input iterators. The std::copy algorithm requires that its first two arguments are at least input iterators and that its last argument is at least an output iterator. From this we can deduce that a dir_iterator is at least an input iterator. In fact, given stream semantics – you can never step twice into the same stream – an input iterator is the most that it can be. The following code is an implementation of such a dir_iterator:

```
class dir_iterator :
 public std::iterator<std::input_iterator_tag, std::string>
{
public:
   dir_iterator()
    : dir(0)
   {
   }
   explicit dir_iterator(dir_stream &stream)
    : dir(&stream)
   {
      *dir >> value;
   }
   const std::string &operator*() const
   {
      return value;
   }
   const std::string *operator->() const
   {
      return &value;
   }
   dir_iterator &operator++()
   {
      if(dir)
         *dir >> value;
      return *this;
   }
   dir_iterator operator++(int)
   {
      dir_iterator old = *this;
      ++*this;
      return old;
   }
   bool operator==(const dir_iterator &rhs) const
   {
      return at_end() && rhs.at_end();
   }
   bool operator!=(const dir_iterator &rhs) const
   {
```

```
        return !at_end() || !rhs.at_end();
    }
private:
    bool at_end() const
    {
        return !dir || !*dir;
    }
    std::string value;
    dir_stream *dir;
};
```

Polymorphism overload

In terms of polymorphism, we can see that in support of parametric polymorphism, overload polymorphism is used extensively. The overloaded operators pretty much follow standard form and expectations. For instance, the post-increment operator returns the old value, not the incremented one. The only point worth drawing your attention to is the test for equality (or inequality). Because of the single-pass, stream-consuming nature of this iterator, it doesn't make much sense to compare arbitrary iterators. The only time that comparison of their values has a definite and useful outcome is when the iterators have reached the end of their traversal.

The derivation from std::iterator represents the only use of inheritance in this article, but it doesn't represent inclusion polymorphism. While, strictly speaking, it doesn't break subtyping guidelines, it's certainly inheritance for convenience. It offers a set of typedefs that allow your iterator type to work with the std::iterator_traits compile-time reflection facility.

Although it's a simple problem, the directory iteration task demonstrates what's required in wrapping up a data source as a stream and providing access to that stream via an iterator. This may be directly useful to you, but more than likely you have other data sources that are more important to you, such as token streams, data feeds and event notifications.

Programming style is influenced by all manner of features, and in the case of C++, the different forms of polymorphism seem to underpin many of them. A broader appreciation of polymorphism can help to simplify and decompose certain problems according to their frame. ■

References

1. Luca Cardelli and Peter Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", *Computing Surveys*, 17(4):471-522, December 1985
2. Kevlin Henney, "Substitutability", *C++ Report* 12(5), May 2000. Also available from www.curbralan.com
3. Kevlin Henney, "Making an Exception", *Application Development Advisor*, May 2001
4. Kevlin Henney, "Safely Releasing Objects into the Wild", *Application Development Advisor*, June 2001
5. Douglas C Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, Wiley, 2000
6. Kevlin Henney, "Portability: Listing Directories", Cvu 9(5), July 1997. Code available from www.curbralan.com
7. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995