"Architecture" has become one of the vaguest terms in software development. What's clear to **Kevlin Henney** is the importance of a solid structure, which requires the right balance of coupling and cohesion

The perfect couple

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

THE WORD "ARCHITECTURE" SEEMS TO BLESS all manner of activities and artefacts in modern software development. It has become the new word for "good". Anything that involves more than one programmer working across more than one machine is now architecture-driven, and any method adopted is architecture-centric (or is it the other way around?).

To some people, architecture is the use of popular distributed middleware of some kind. Alternatively, it is often used to give an air of respectability to what might otherwise appear to be a chaotic jumble of components. It is sometimes used to elevate the status of a design decision – "that's an architectural decision" – or of an individual – "I'm a technical architect."

That's not to belittle architecture; architecture is too important an issue to be left to marketing departments or elitists. A full study of what is meant by software architecture is beyond the scope of this column. Fellow *Application Development Advisor* columnist Alan O'Callaghan and I have been exploring approaches to architecture in workshops and tutorials at a number of conferences. To get a feel for where we're coming from, I would suggest tracking Alan's column in this magazine.

However, there are some observations that can be made about the nature of software architecture, that are of immediate relevance to the C++ programmer. Architecture is not a separate phase or activity in development, somehow divorced from other development concerns. It defines the form and characteristics of a system, taking into account social and economic factors alongside the more obvious technical ones. Architectural style can be considered a guiding hand in a design. Architecture is not about blueprints, nor is it concerned solely with the big abstract boxes in your system, the large blobs in some diagram sometimes known as "PowerPoint architecture". Architecture applies to all scales, from the big chunks down to the intimate bits of the code.

So, any development that is architecture-driven must have a clear focus on structure, style and scale. The little picture deserves as much airtime as the big one because – depending on whether you regard the glass as half-empty or half-full – it's either the god or the devil that lies in the details. Detail matters.

Making and breaking the connection

Structure arises from separation and connection; the partitioning of a program with respect to responsibilities and concerns, and the interfaces that sit at the partition boundaries. Partitioning a system in C++ takes you into the detail of the language to a finer design level. What conventions will the object's class adhere to? What other classes are involved? In which header file in which directory should the code live? What other files will depend on the header file? How often is the code likely to change? Embracing these concerns is what the term "architecture-driven" entails – which means that most approaches claiming this status aren't at all.

OK, but is this level of detail that important? Why does source code matter to a software development project? I said, structure is related to separation and connection. What happens if you don't treat the

FACTS AT A GLANCE

- Architecture is concerned with structure, both its properties and its motivation.
- Dependency management is vital in large systems to reduce build times, minimise ripple effect and improve comprehensibility.
- The need for file inclusion in headers can be reduced using forward declarations for certain classes.
- A Cheshire Cat class hides its implementation in the corresponding implementation file.
- Interface classes omit implementation detail, which is deferred to a derived class.
- Templates decouple usage from the specific type used.

structure of the software as a peer of the software's functionality? Structure often ends up being the poor cousin of functionality, leading to code with high coupling and low cohesion. What we are aiming for is low coupling and high cohesion, which means that many developments are a long way off target.

Far from being concepts of interest solely to academics, coupling and cohesion relate to concepts that are as practical as you could wish for. They define, respectively, the interconnectedness and intraconnectedness of components, whether functions, classes or subsystems, and their interfaces, whether internal or external. The more tightly coupled a system is, the harder it is to understand, change and test. Given that these three activities pretty much define software development (although the first and last sometimes seem to be treated as optional) it's easy to see that loose coupling has benefits that are practical and tangible rather than theoretical and abstract. The value of a system lies ultimately in its source, so active dependency management is a vital part of keeping software soft.

Can we fix it? Yes, we can. There are a number of different C++ practices that can keep a system's fine-grained architecture supple and responsive, each with different trade-offs that allow you to select or combine them appropriately. These idioms also reduce build times significantly, which takes the irritation out of iteration.

Independence of declaration

My previous column[1] presented the dir_stream and dir_iterator classes for listing the contents of a directory. Depending on how you want to present it, these two classes could live in the same header or separate ones. Assuming they are part of the same design effort, I would be inclined to include them both in the same header. If they are conceived of and deployed separately, or you anticipate using dir_stream without using dir_iterator, separation may be more appropriate.

So, should you decide to keep them separate, what are the conceptual and necessary physical dependencies involved? The dir_iterator class depends on dir_stream, but the converse is not true. It is tempting to define the header for dir_iterator as follows, assuming that the dir_stream class is defined in a header "dir_stream.hpp":

```
#include "dir_stream.hpp"
#include <iterator>
#include <string>

class dir_iterator :
  public std::iterator<std::input_iterator_tag, std::string>
{
public:
    dir_iterator();
    explicit dir_iterator(dir_stream &);
    const std::string &operator*() const;
    const std::string *operator->() const;
    dir_iterator &operator++();
    dir_iterator operator++(int);
    bool operator==(const dir_iterator &) const;
    bool operator!=(const dir_iterator &) const;
private:
    bool at_end() const;
    std::string value;
    dir_stream *dir;
};
```

Although there is a conceptual dependency on dir_stream at this level, in the header, there is no actual physical dependency. Only pointers and references to dir_stream are declared, and nothing requires the definition of dir_stream's content. The common C++ practice in such cases is to drop the inclusion of "dir_stream.hpp" in favour of a forward declaration:

```
#include <iterator>
#include <string>

class dir_stream;

class dir_iterator :
  public std::iterator<std::input_iterator_tag, std::string>
{
    ...
};
```

The "dir_stream.hpp" header must now be included in the source file containing the definition of the dir_iterator member functions, and wherever else in the program a dir_stream object is created or has its member functions called.

For classes as small as these, there is only a small saving: the returns on investment for this practice become more visible for larger or more complex classes that aggregate more dependencies. A surprising number of projects raise complaints about build times, but fail to adopt this practice. Precompiled headers address the problem only at a superficial level. Note that using forward declarations makes precompiled headers even more effective.

A couple of words of warning: first, don't forward declare anything in third-party libraries. Appearances can deceive and external libraries are entitled to change certain details. For example, in pre-standard C++, a common forward declaration to avoid, including the large I/O stream headers, was for the ostream type:

```
class ostream;
```

This does not, however, have the desired effect in the context of standard C++, which moved the goalposts a bit:

```
namespace std
{
    ...
    template<
        typename char_type,
        typename traits = char_traits<char_type> >
            basic_ostream;
    typedef basic_ostream<char> ostream;
    ...
}
```

For this reason, the standard C++ library encloses forward declarations for its I/O streams library in a separate <iosfwd> header. For more complex class definitions you might consider adopting the same forward declaring header practice.

The other word of warning concerns consequences. There is no such thing as a design decision that does not affect the design. In this case, the use of forward declaration rather than inclusion means that dir_iterator cannot be implemented with inlined functions. Design is all about weighing up pros and cons to arrive at an appropriate decision. Remember that a decision to use forward declarations is surprisingly easy to reverse, should you change your mind and opt for inlined function definitions.

Herding cats

A delegation technique based on forward declarations can also be used to reduce the dependencies and platform coupling in the dir_stream class. The dir_stream class was previously implemented in terms of

the features found in the POSIX <dirent.h> header:

```
#include <dirent.h>
#include <string>

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

If you are not working on a POSIX system, you can recreate the POSIX API by wrapping your native API appropriately. However, you may prefer to implement dir_stream directly per platform rather than mimic a standardised API to achieve portability. Alternatively, you may not care about platform at all, but would rather not pollute your header, and impose a corresponding burden on other dependent files with potentially large system headers.

The Cheshire Cat idiom[2] (also known as the Fully Insulating Concrete Class idiom[3]) and the Pimpl idiom[4] is perhaps one of the oldest class restructuring idioms in C++, dating back to Glockenspiel's CommonView library in the late 1980s. Thanks to nested forward declarations, modern C++ provides more encapsulated support for the idiom than early C++:

```
#include <string>

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

The representation of a dir_stream object has been removed from the header file, leaving only a trace – no more than a smile – in the form of a pointer. The type of pointer is fully elaborated in the corresponding source file that defines all the member functions:

```
#include "dir_stream.hpp"
#include <dirent.h>

class dir_stream::body
{
public:
   body(const char *);
   ~body();
   DIR *handle;
private:
   body(const body &);
   body &operator=(const body &);
};

... // member function definitions
```

The details of the nested body type are private not only to the dir_stream class but also to the corresponding implementation source file. You can change the implementation details for portability, debugging or fixing, without requiring recompilation of any files except the dir_stream implementation file. All that is required to incorporate any change is relinking. Because the dir_stream class holds only a pointer it remains binary compatible, no matter what changes occur in the implementation. This also makes it a good technique for defining classes exported by DLLs.

In terms of portability, the unsightly mess of conditional compilation directives that normally accompany such headers can now be isolated and kept out of the way. In terms of safety, a Cheshire Cat also guards against acts of object terrorism, such as #define private public.

To keep the good news in check, there are a couple of implementation consequences to be observed in Cheshire Cats. As with the previous forward declaration technique, you cannot define dir_stream's member functions inline. The design is based on separation, adding a level of indirection that must be memory managed. Therefore the body object must be allocated in the dir_stream constructor and deallocated in the destructor, or when the stream runs dry. You may chose to use an auto_ptr or a scoped object[5] to attend to the needs of the heap object.

Tax-free inheritance

In an object-oriented system, the inheritance relationship is the strongest form of coupling there is. Consider changing the public or protected features of a base class: the tsunami-like scale of the ripple effect gives rise to a problem known as the fragile base class problem[6]. You are strongly encouraged to keep base classes stable. And there is nothing more stable than a class with no implementation. An interface or pure abstract class is one whose only ordinary members are functions that are public and pure virtual:

```
#include <string>

class dir_stream
{
public:
   virtual ~dir_stream();
   virtual operator const void *() const = 0;
   virtual dir_stream &operator>>(std::string &) = 0;
private:
   dir_stream(const dir_stream &);
   dir_stream &operator=(const dir_stream &);
};
```

C++ programmers are sometimes reluctant to write a class that appears to do nothing. It is always tempting to add a few default features. Resist the urge and you can cash in on the decoupling benefits of a pure interface.

In other words, inheritance can also be used to reduce the coupling in a system. Consider an alternative to the delegation-based approach of the Cheshire Cat, with alternative implementations and representation hiding offered with respect to inheritance:

```
#include "dir_stream.hpp"

class posix_dir_stream : public dir_stream
{
```

```
public:
    explicit posix_dir_stream(const std::string &);
    virtual ~posix_dir_stream();
    virtual operator const void *() const;
    virtual posix_dir_stream &operator>>(std::string &);
private:
    void close();
    DIR *handle;
};
```

This practice has very different trade-offs from the Cheshire Cat. The only place where the concrete class must strictly be visible is at the point of creation. From that point on, access via the dir_stream base is sufficient. Sometimes, to enforce this usage, programmers override the virtual functions as private rather than public[7]. Object factories can also be used to hide the underlying concrete class.

On the downside, the use of interface classes encourages heap creation and access by pointer. This is not normally a disadvantage at all, but in the case of directory streams, which are more appropriately stack objects, an extra level of indirection makes the operator overloading clumsy to use. So, in the case of dir_stream, the Cheshire Cat seems to provide a more appropriate solution to the use of interface decoupling. However, do not assume that you can generalise a one-size-fits-all rule for other cases. Assume the opposite: design is about the conscious comparison of alternatives, not about the blind application of rigid rules.

Generically yours

It may seem counter-intuitive, but there are scenarios in which templates reduce the coupling in a system. This is counter-intuitive because templates normally impose the physically coupled burden of including implementation detail in header files. However, as with inheritance, the mechanism can be used both to increase and decrease explicit dependencies.

Consider an alternative requirement for dir_stream. It is not the inclusion of <dirent.h> that you wish to decouple, but the use of the <string>. In coupling terms, this standard header is perhaps a little too well connected. At the point of usage, the user of dir_stream will have the appropriate string definition visible, but at the point of definition it can be omitted. Focusing on the relevant templated functions:

```
#include <dirent.h>

template<typename string_type>
const char *c_str(const string_type &string)
{
    return string.c_str();
}

class dir_stream
{
public:
    template<typename string_type>
    explicit dir_stream(const string_type &dir_name)
      : handle(opendir(c_str(dir_name)))
    {
    }
    template<typename string_type>
    dir_stream &operator>>(string_type &rhs)
    {
        if(dirent *entry = readdir(handle))
            rhs = entry->d_name;
        else
            close();
        return *this;
    }
    ...
};
```

Architecture is not about blueprints, nor is it concerned solely with the big abstract boxes in your system, the large blobs in some diagram sometimes known as "PowerPoint architecture"

The two functions that rely on strings – the constructor and the stream extraction operator – are now written in terms of an argument type that will be deduced at compile time at the point of use. The argument type must satisfy the requirements laid down in each function. In the constructor, the requirement is that the string type be used as an argument to a function named c_str that returns something that is convertible to a const char *. In the extraction operator the string type must be assignable from a char *, the type of entry->d_name.

With this particular decoupling comes generality. The default implementation of the c_str helper function assumes the presence of a member function also named c_str. If you have another string type that supports a c_str member, such as SGI's rope[8], the code will work just as well. Alternatively, you can provide an overload to work with a string type of your choice, including vanilla char *:

```
const char *c_str(const char *string)
{
    return string;
}
```

The generic decoupling approach forms the basis of much of the STL, in particular its algorithmic function templates.

In some instances, the decoupling techniques demonstrated may be complementary. However, it is more likely that their respective demands and merits will be at odds with one another. Architecture is a matter of appropriate style and selection. ■

References

1. Kevlin Henney, "The Four Faces of Polymorphism", *Application Development Advisor*, October 2001, available from www.appdevadvisor.co.uk
2. Robert B Murray, *C++ Strategies and Tactics*, Addison-Wesley, 1993
3. John Lakos, *Large-Scale C++ Software Design*, Addison-Wesley, 1996
4. Herb Sutter, *Exceptional C++*, Addison-Wesley, 2000
5. Kevlin Henney, "Making an Exception", *Application Development Advisor*, May 2001, available from www.appdevadvisor.co.uk/
6. Clemens Szyperski, *Component Software*, Addison-Wesley, 1998
7. Kevlin Henney, "Total Ellipse", *C/C++ Users Journal online*, March 2001, www.cuj.com/experts/1903/henney.htm
8. SGI Standard Template Library, www.sgi.com/tech/stl/