Programmers can find themselves deluged by too much information. So if it's all too much, start here. **Kevlin Henney** recommends six good practices that no programmer can afford to forget

Six of the best

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

THE PROGRAMMER'S WORLD IS FILLED WITH more than enough detail, including book upon article upon in-house document of recommendations of what to do and what not to do. Whether these are devoured whole, digested in part or even read at all varies between programmers.

Despite its enormity, this body of wisdom on design and practice should not be ignored, but at the same time it can be too much to take on at once without mental buffer overflow: "Go out and read up on patterns and architecture; read all the recommended books on C++ good practice; don't forget to get up to speed on UML, IDL, XML." Abandon hope? Or choose a starter pack of practices and work from there?

It is this idea of a limited set of recommendations with maximum effect that I have become interested in. What follows is a concrete 12-point list of practices that you can apply out of the box to the code in front of you. To accommodate detail – and add a little suspense – I will cover the first six in this column and the remaining half dozen in the next.

1. Make roots fully abstract

You have decided – or it has been decided for you by existing code – that you are developing a class hierarchy. What should go at the top?

There is a temptation to put a concrete class at the root of the hierarchy, one that provides a default representation of some kind, with the hope of making it easier for derived classes to be written. The intent is generous, but it is overly so. A concrete class suggests a complete and whole concept. If you can further specialise it into another concrete class, clearly it was incomplete and at least conceptually abstract. Making the root concrete forces derived classes to know a great deal about its implementation in order to work out how to override or even ignore parts of it. This is an act that suggests the root knows and does too much.

A concrete base class also ties users of that class into the implementation details, pulling in all the header dependencies. A concrete class is likely to undergo changes as it evolves, which causes a compilation and testing ripple effect. Still on the subject of testing, a concrete class is difficult to stub out when you need a dummy in its place for testing purposes. In short, concrete root classes make class hierarchies harder to work with in the long term, not easier.

The advice to avoid inheriting from concrete classes[1] and ensure that the tree of inheritance has concrete leaves only[2] is not new, but it can be taken further. Make the root of the hierarchy fully abstract, i.e. no data members and ordinary member functions are public and pure virtual only. With the Interface Class idiom[3] the root of the hierarchy is far more stable, as there is no implementation that can change. It is clearer because it is uncluttered by incidental default implementation guesses. If you want to provide some kit parts that make it easier to derive further, they can be provided in a variety of abstract classes the next level down rather than just one.

This recommendation is relatively easy to put into practice. It is easy to spot a hierarchy without a fully abstract base and it is trivial to refactor – there is a well documented Extract Interface refactoring that fits the bill precisely[4]. I had feedback recently from a project that I had mentored, and of all the design techniques I had introduced, this simple one was considered to have the most widespread effects. It had radically simplified testing, code comprehensibility and design evolution.

2. Inherit for subtyping only

Inheritance is a powerful tool. In the right hands, it can be used to create a loosely coupled system with cohesive units of responsibility. *Make Roots roots fully abstract* is an example of this. In the wrong hands, it is a blunt tool that serves only to bludgeon a family of classes into incomprehensibility; forensic inspection of a tangled genetic web is needed to determine exactly how a given class behaves. The quantity of inheritance is not necessarily an indicator of whether or not a piece of code is any good. To do this, you have to look at the quality of inheritance.

In C++, it is public derivation that causes most of the grief because it is a strong type relationship with implications for the derived class and its users. A class hierarchy is most easily comprehensible, most easily tested and most easily adapted when it follows classification based on the externally visible properties of a type (*subtyping*) and not on its internal representation (*subclassing*). It is common for the paths of subtyping and subclassing to coincide: where 'is

FACTS AT A GLANCE

- **Make roots fully abstract**
- **Inherit for subtyping only**
- **Include only what you need**
- **Consider duplicate code a programming error**
- **Use self-contained objects**
- **Make acquisition and release symmetric**

a kind of' and 'is implemented as' suggest the same hierarchy. But where they do not, it is a failure to follow subtyping that will most likely confuse the code and its readers.

Therefore, the tree of specialisation should be formed with respect to substitutability[5,6], with the most general concepts at the top (hence why you should make your roots fully abstract) and the specifics branching out. Each specialisation should be substitutable for its base class. Where a pointer to the base is expected, an instance of the derived will work. To do anything else reduces the meaningful behaviour of a system, making it a patchwork of special cases. Sometimes the requirements of a framework will force your design away from this recommendation, but in other cases you can refactor the code to support it.

3. Include only what you need

Or, put another way, don't include what you don't need. #include textually substitutes the contents of a header (and anything it includes in turn) into a source file. This mechanism can become a dominant source of drag in the build process. Precompiled headers are an attempt to treat the symptoms but they do nothing to address the root causes, which are related to design dependencies rather than compiler execution speed.

Including what you don't need doesn't help anyone. I recall looking at a header file that seemed to include just about everything imaginable – almost all the C and C++ library, most of the GUI library, large chunks of the lower level OS API and many application-specific headers. This shopping list of #includes was headed with the unhelpful comment, 'just in case'. Just in case of what? I have been wondering for years. The tragedy of this common header is that it was included by almost every file in the system, contributing to a build speed that even a snail would be embarrassed to race against.

Unnecessary #include dependencies are not only wasteful, they are confusing. The physical dependencies in the code should give you some idea of the conceptual dependencies. Unused or redundant #includes throw a smokescreen in the way of this rule of thumb. Elsewhere in the project I just described was another common header that was clearly unaware of the impressive #include list of the just-in-case header, and had decided to run up a dependency list all of its own. The list of included files was suspiciously similar, but was headed by the somewhat bold claim of 'include those files that are always needed'. Only the most nightmarish of source files could ever require all those headers simultaneously.

Therefore, any #include whose contents are not being used should simply be dropped. Where a header is included only so that a pointer to a struct or class needs to be but not dereferenced, a forward declaration of the type should replace the #include. You can also reduce the number of #includes by reducing your need for them. Making your roots fully abstract removes a lot of implementation detail from heavily used headers; the Cheshire Cat idiom[3] uses forward declarations to teleport representation detail from a header file into the corresponding implementation source file.

The recommendation to include only what you need also implies avoiding unnecessary centralisation, such as listing all error codes or application-specific constants in a single header.

4. Consider duplicate code a programming error

Duplicate code has a bad smell[4] and violates the catchily named DRY principle (Don't Repeat Yourself)[7]. Duplicate code attracts bugs like a flame: a code fix in one place does not automatically update any of its duplicates.

The accumulation of duplicate code also has an obvious side effect: it increases the amount of source code. Contrary to management belief, more is not better in this case. Every problem has an optimal code size. Too short and the code is cryptic line noise, too long and you can't see the wood for the trees. Duplicate code can obscure the intent of code, causing you to wade rather than run through it, slowed by the detail and a haunting sense of déjà vu – is this code actually the same as that code, or is it subtly different?

Duplicate code should be refactored away at the earliest possible opportunity. You can pull common code into a separate function or class, sometimes pulling it up into a common base class as long as you inherit for subtyping only. Many of the other practices also have the effect of collapsing and eliminating common duplications, such as the recommendation to use self-contained objects.

5. Use self-contained objects

C++'s relationship with C is both a weakness and a strength. Its weaknesses become apparent in matters of habit and style inherited from C. For instance, a worrying amount of string manipulation is still performed on raw char arrays. The user of the string must allocate the string (having measured out the right length) and deallocate it when done... or get the string from somewhere else and remember to deallocate it... or allocate it and assume that whoever it is passed to will be deallocating it. Even leaving aside the excitement of accidental buffer overruns, the fact that these options exist at all makes dealing with raw char strings tedious and error-prone. This is why string classes exist: they look after themselves and let the programmer get on with what they were supposed to be doing. For all its other featuristic flaws, the standard string type greatly simplifies string-based code. Compare the following function[8]:

```
char *full_name(const char *first, const char *last)
{
   const size_t length = strlen(first) + strlen(last) + 1;
   char *result = new char[length + 1];
   strcpy(result, first);
   strcat(result, " ");
   strcat(result, last);
   return result;
}
```

And its exception and memory-safe usage:

```
{
   char *name = full_name(first, last);
   try
   {
      ... // perform tasks using name
   }
   catch(...)
   {
      delete[] name;
      throw;
   }
}
```

With the following function:

```
string full_name(const string &first, const string &last)
{
   return first + " " + last;
}
```

And its safe usage:

```
{
   string name = full_name(first, last);
   ... // perform tasks using name
}
```

But this recommendation is more than a gentle reminder to use std::string: classes: they should not be in the habit of exposing their memory or data structure management. Both standard and non-standard container types remove the tedium, duplication and house-keeping nature of working with lower level representations. They unask questions of representation management by simply not exposing it. Where these classes do not quite fit the bill, you can adapt them to match your own requirements more precisely[9,10,11]. More generally, using self-contained objects also implies keeping data private and wrapping API details[3]. You can find further refinements when you make foots fully abstract and include only what you need. When writing classes for self-contained objects, ensure that you make your acquisition and your release symmetric, so that they can be truly self-contained.

6. Make acquisition and release symmetric

The ownership problems with raw strings highlight another more general problem with any dynamically created object: what destroys it? This is a question that must be asked of any piece of C++ code. If the answer is unclear, you can bet safe money on a memory trashing bug or a leak at some point in the future (or present, or past). Generalising a little, the release policy of any acquired resource should be clear; memory is the most obvious and popular example.

Code with unclear options for deletion is sometimes justified in the name of flexibility. At best, 'flexibility' is a woolly term, an excuse for vagueness rather than a quantifiable reason. More honestly, such options – supported by flags or guesswork – arise from uncertainty, causing much of the same in their wake: "If the second flag is set, the fourth caller of the third function must destroy the object, otherwise it is destroyed automatically, except on Sundays, Wednesdays and public holidays." The deprecated strstream type in the standard library promotes confusion of ownership into an art form.

So, what should be responsible for disposing of an object once it has been dispensed? The simplest and least error-prone strategy is to ensure that whatever created it destroys it. The element responsible may be as small as a block within a function, so that new and delete, for instance, mirror one another directly in the code. The 'whoever' may extend to a whole object's life, so that whatever it creates in its constructor is destroyed in its destructor. If there is any replacement to be done it is handled by the object: destruction and new creation are fully encapsulated and replacements are categorically never passed in from outside the object. Again, *Use self-contained objects*.

What about factory objects? Surely they break this recommendation for symmetry? Yes, potentially, but they shouldn't. A factory object offers a function that returns a newly created object:

```
class product;
class factory
{
public:
   product *create();
   ...
};
```

Many factories leave the disposal to the recipient:

```
void example(factory *creator)
{
   product *created = creator->create();
   ... // use created
   delete created; // assuming that delete is correct
}
```

This is a shortcoming rather than a feature of most object factory designs. If the details of creation are encapsulated, then why not the details of destruction as well? The opportunity for object pooling and recycling is there, but it does not have to be exercised. Even if disposal is just a straight delete, a symmetric interface that offers both the creation and disposal operations is better encapsulated:

```
class product;
class factory
{
public:
   product *create();
   void dispose(product *);
   ...
};
```

And easier to work with:

```
void example(factory *creator)
{
   product *created = creator->create();
   ... // use created
   creator->dispose(created);
}
```

The message is simple: "Take it back to wherever you got it from." Such symmetry makes a design easier to comprehend and bugs easier to spot. There's less to remember to get it right, and any omission – and hence asymmetry – stands out like a sore thumb. Symmetry should only be broken where the result is both simpler and safer.

Protection of investment

These recommendations are all about protecting and getting return on investment. If you are about to pour, or have just poured, a great deal of programmer effort into developing a system, it would seem foolish not to get the best value for the time and money invested. Without code, there is no system, so the code matters. The recommendations take an 'act locally, think globally' view of piecemeal improvement, rather than a master-planned big-bang approach. ■

References

1. Scott Meyers, *More Effective C++*, Addison-Wesley, 1996
2. Kevlin Henney, 'Total Ellipse', *C/C++ Users Journal* online, March 2001, www.cuj.com/experts/1903/henney.htm
3. Kevlin Henney, 'The perfect couple', *Application Development Advisor*, November–December 2001, available from www.appdevadvisor.co.uk
4. Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999; see also www.refactoring.com
5. Kevlin Henney, 'Substitutability', *C++ Report*, May 2000, available from www.curbralan.com
6. James O Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992
7. Andrew Hunt and David Thomas, *The Pragmatic Programmer*, 2000; also www.pragmaticprogrammer.com
8. Kevlin Henney, 'The miseducation of C++', *Application Development Advisor*, April 2001
9. Kevlin Henney, 'Bound and checked', *Application Development Advisor*, January-February 2002
10. Kevlin Henney, 'Look me up sometime', *Application Development Advisor*, March 2002
11. Kevlin Henney, 'Flag waiving', *Application Development Advisor*, April 2002