Following his *Six of the best* column last issue, **Kevlin Henney** concludes his set of 12 best practice tips for writing elegant, well-trimmed C++ code

The rest of the best

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

THERE IS NO SHORTAGE OF TECHNICAL WISDOM on how to develop clear and robust code, so why is the paned (sic) expression on a programmer's face—sifting through all of the source windows trying to make sense of the encrypted code—such a common sight? There are companies whose development culture does not encourage best practice: a penny-wise/pound-foolish approach. However, there are many companies and developers that want to push themselves to the state of the art, but seem swamped and bemused by how much state there really is to that art.

In the last column[1], I looked at the first six of a set of 12 recommendations that can potentially make a big difference to a body of C++ code. The short list does not claim to cover all design practices appropriate for a C++ system, but it does offer an easily memorable and easily practised set of guidelines that offer the greatest immediate return on investment—the most bang for your buck, the most oomph for your euro, the most kerrang for your quid.

The story so far:

1. Make roots fully abstract:
Classes should be either classifiers of behaviour or implementations of behaviour. Class hierarchies should not muddle the roles of abstract and concrete classes. A fully abstract class—no data and ordinary member functions are public pure virtual only—offers the lowest coupling and clearest root to a class hierarchy.

2. Inherit for subtyping only:
There is a strong temptation to use inheritance (in the sense of public rather than non-public derivation) as a convenience mechanism to build on parts of existing code. Inheritance for this purpose is a remarkably blunt tool when compared to the more precise aggregate-and-forward approach. Inheritance is most effectively employed as a tool for classifying externally visible behaviour (subtyping) than for arranging internal representation (subclassing).

3. Include only what you need:
Redundant #includes and excessive dependencies slow down both the programmer and the compiler. Many uses of #include can be replaced by forward declarations. Other heavy conceptual dependencies that lead to physical dependencies can often be reduced by delegation and templating. Some conceptual dependencies are incorrectly conceived: for instance, centralising error codes or application-related constants in a single header.

4. Consider duplicate code a programming error:
Where the lack of an architectural vision can lead to a loss of the big picture, duplicated code can cause loss of the small picture.

5. Use self-contained objects:
Objects that expose their representation management details, such as publicising data members or allowing users to replace dynamically allocated objects that they depend on, tend to be difficult to use correctly. Objects should manage their own representation fully, preventing access or inappropriate manipulation by users. Consider the difference in ease of use between using std::string and a char * for manipulating strings.

6. Make acquisition and release symmetric:
Acquisition of resources, such as dynamically allocated memory, should be matched by a release at the same scope, in the same object or through the same interface. Transfer of object ownership should be avoided, so that object factory interfaces should support both creation and disposal operations.

The remaining six recommendations continue the theme of avoiding unnecessary centralisation and exposure to risk, filtering the necessary complexity from the unnecessary.

7. Use objects to automate release

Make acquisition and release symmetric simplifies the style to be used for resource acquisition and release. For instance, where possible, memory should be released in the same context in which it is allocated, where the context could be a block or an object.

```
{
    product *created = new product;
    ... // use product
    delete created;
}
```

Transferring responsibility for release is tricky,

FACTS AT A GLANCE

- **7. Use objects to automate release**
- **8. Flow don't jump**
- **9. Sharpen fuzzy logic**
- **10. Hand redundant code its notice**
- **11. Let the code make the decisions**
- **12. Prefer code to comments**

and disciplined symmetry reduces the chance that something has been overlooked. However, the symmetry can be lost if we take failure into account: what if an exception occurs? Sometimes the longest way around really is the longest way home:

```
{
   product *created = new product;
   try
   {
      ... // use product
   }
   catch(...)
   {
      delete created;
      throw;
   }
   delete created;
}
```

The symmetry has been broken, and in its place we have long-winded, repetitive and error-prone code—ironic, as it is coping with the possibility of failure that has inspired this code. As we *consider duplicate code a programming error* this situation calls for a tidier remedy, and an antidote similar to the advice to *use self-contained objects* suggests itself. Although not necessarily fully self-contained or symmetric, the responsibility for release can be encapsulated in an object's destructor:

```
{
   auto_ptr<product> created(new product);
   ...
}
```

Not all acquisition and release is concerned with individual objects on the heap, but the encapsulation of control flow can be generalised[2, 3]:

```
{
   scoped<FILE *, closer> file(fopen(name, mode));
   ...
}
```

If something is tedious and error-prone, automate it.

8. Flow, don't jump

It's tough being a salmon. A life spent at sea is a prelude to a special forces survival course for mating, jumping uphill against the natural flow of a stream. To look at the control flow of some programs is to stand by a turbulent stream like a hapless and hopeless bear waiting to knock a salmon out with its paws.

The basic ingredients of continuous control flow are sequence (where one statement follows on to the next), selection (if else and switch) and iteration (while, for and do while). These are the basic ingredients of structured programming, and the lessons learnt from structured programming should not be forgotten. Sometimes discontinuous control flow mechanisms seem attractive (return, break, continue and goto) and offer apparent short cuts. Early return and break can sometimes be justified, but continue and goto should be considered surplus to requirements.

Most uses of discontinuous control flow are like a drug: they are addictive, and it can be difficult to stop. It can all start with something seemingly simple and innocuous:

```
while(in_range(value))
{
   if(value == delimiter)
      break;
   ... // now carry out intent of loop
}
```

One fix leads to the next, and before you know it there is a function that is impossible to understand and difficult to debug because of all its jump points—and debugging is suddenly important because bugs seem to be attracted to long functions with jumpy control flow. Ironically, jumpy code is often written in the name of convenience (it's supposedly "easier this way"). It's actually easier this way:

```
while(in_range(value) && value != delimiter)
{
   ... // carry out intent of loop
}
```

When it comes to the infamous goto, I must confess that I am still surprised to find C++ programmers that feel the need to use them when the alternatives are invariably simpler. I ran across a piece of code recently that was chock full of gotos. In spite of having once been a Fortran programmer, I didn't have a clue what was going on...and, when asked, neither did the programmers who wrote and maintained it. They mumbled something about making it easier to handle errors and to clean-up resources. We walked through it and refactored it: without the gotos it was shorter—losing two thirds of its former length—and clearer—a bug that had been hiding in the previous control-flow salad revealed itself.

When you *consider duplicate code a programming error, use objects to automate release* and *sharpen fuzzy logic* you find a lot of the jumpiness disappears from code. Other recurring jumps suggest different selection or looping structures, or encourage the use of exceptions.

Error return codes tend to encourage a verbose coding style where most of the code is structured to propagate the error. Error code propagation increases the number of explicit paths in your code, which by definition makes the code more complex. Return values should, on the whole, be used to return useful values rather than good news/bad news bulletins.

Hang on, don't exceptions contradict the message about smooth flow, jumpy code, salmon and streams? Not at all. One of the main problems with jumpy code is its disrespect for modularity. Refactoring one large function into many is often a process of splitting out a specific sequence or loop, giving it a name and figuring out what local variables need to be passed in and results returned. Data flow respects modularity, but what do you do with control that doesn't flow? How to make the effect of a break or early return statement non-local is less obvious than just passing and returning copies of local variables. Extra status values often have to be introduced and passed around in a game of pass the control flow. Exception flow, however, is trivial to refactor because it remains unchanged: an exception leaves a block and a function, or a function called by a function, in the same way.

9. Sharpen fuzzy logic

Like overly jumpy control flow, a piece of logic can sometimes accumulate mass like a snowball. Before you know it, you haven't a clue what is going on and the logic has gone fuzzy. I could have named this recommendation "review fuzzy logic", but there would have been no point: the only thing to do with complex logic is to simplify it. The most direct way to apply the scalpel is to recall some of the basic transformations and operations that are possible with Booleans. For instance, a high incidence of true and false literals in code can often be an indicator that some simplification is possible:

```
if(failed)
    return false;
else
    return true;
```

Becomes the slightly less pedestrian:

```
return !failed;
```

De Morgan's law allows you to move freely between:

```
!(a || b)
```

and:

```
!a && !b
```

In other words, if neither a nor b is true, then both a and b are false. Other operator relationships also help you to simplify things, such as:

```
!(i <= j)
```

to:

```
i > j
```

A surprising amount of logic in production code will submit itself to these and many other simple transformations. If you are confronted with a long or tortuous piece of logic that seems to shrug off any attempt at reduction, wrap it in a function: give it a good home and a good name rather than letting it clutter up the main flow. You may already have arrived at this conclusion if you find the logic appearing in more than one place and have chosen to *consider duplicate code a programming error*. In other cases, *flow don't jump* is reinforced by simplifying complex logic, which in turn often allows you to *hand redundant code its notice* and *let the code make the decisions*.

10. Hand redundant code its notice

Redundant code is code that has no genuine effect. It varies from easy-to-spot redundant checks to more elaborate arrangements that reveal their no-op nature only on closer inspection.

Some redundant pieces of code are trivial and can also be considered under *sharpen fuzzy logic*, such as the repetitive:

```
if(failed == true) ...
```

Which says the same thing twice:

```
if(failed) ...
```

And the similarly wordy:

```
return value == delimiter ? true : false;
```

which needs no more detail than:

```
return value == delimiter;
```

Redundant code can be more intricate than a few logical excesses or a bit of control flow waffle. An example I found recently was related to thread-safe synchronisation in a multi-threaded environment. The code carefully ensured that each public member function locked a mutex on entry and unlocked it on exit (*use objects to automate release* makes this task simpler and safer). However, not all the functions needed locking: it makes no sense to synchronise a constructor because during construction the object cannot be shared meaningfully between threads; functions with empty bodies have nothing to do, and can always do so safely; functions that do not refer to their data members, such as those that operate only on their arguments or functions that return constant values, do not need synchronisation; and functions that refer only to data members that are immutable values do not need synchronisation.

Unreachable code is a particular category of redundant code that can be chopped without further ado. Depending on how obvious it is, a compiler or checking tool may be able to warn you about it. For instance, a return statement halfway through a function means that the last half is unreachable without teleport. Such hiccoughs are more likely to be present in functions with a long evolution and a long line count. Sometimes unreachable code can be subtle. The following is from some code I reviewed a while back:

```
if(container.empty())
{
    for(iterator at = container.begin();
        at != container.end();
        ++at)
    {
        ... // many lines of code working through at
    }
}
```

Another category of redundant code is unused code. Featurism, overgeneralisation and changed requirements can cause code to be written but never used in practice. Redundant code such as this fattens source that could be lean; a little source liposuction never goes amiss. If you are concerned that such code might one day become useful, the version control system will remember it for you.

Many companies want to push themselves to the state of the art, but seem bemused by how much state there really is to that art

11. Let the code make the decisions

How do you get your program to do something? You specifically write out the code that performs the task. How do you make your program take alternate actions depending on some context or condition? The obvious answer is to write all the decisions and options out explicitly. Decisions and multiple options encoded as if else and switch statements certainly spell everything out in detail, but sometimes that can be just a little—if not a lot—more detail than is strictly necessary.

Sometimes the decision is already being taken for you by the code you are calling. Checking a pointer against null before deleting it is perhaps the most common C++-specific redundant coding habit:

```
if(cache)
    delete cache;
if(connection)
    delete connection;
if(buffer)
    delete buffer;
```

Nothing more than the obvious is required:

```
delete cache;
delete connection;
delete buffer;
```

Here, in an example where you also *hand redundant code its notice*, the underlying code is already making the decision for you.

Data structures can be organised to eliminate decision making,

leading to code that more accurately and explicitly reflects application concepts and constraints in the runtime structure of your program. Consider an application that holds a number of objects that can be in one of two states: saved or changed. How do you save all the changed objects? A common solution would be to include a bool flag and query function in the objects' class:

```
for(deque<saveable *>::iterator at = all.begin();
    at != all.end();
    ++at)
{
    if(at->changed())
        at->save();
}
```

The Collections for States pattern[4] describes an approach that is more explicit, more efficient and takes decision control flow structures out of the code. An additional container holds items that have been changed:

```
for(deque<saveable *>::iterator at = changed.begin();
    at != changed.end();
    ++at)
{
    at->save();
}
changed.clear();
```

Further refactoring lets the STL carry out all of the loop and grind:

```
for_each(
    changed.begin(), changed.end(),
    mem_fun(&saveable::save);
changed.clear();
```

In this case the other container is the master container and holds all the objects, regardless of their state, but a variation would be to have it hold only items that had been saved, so that the two containers hold complementary states.

A problem with hard-coding some decision structures into your code is that the openness and adaptability of the code may be reduced. It is here that the most common form of implicit decision-making has a role to play. Polymorphism—whether runtime polymorphism through virtual functions and inheritance or compile-time polymorphism through templates[5]—cuts back the need for many cascading if, else if or switch statements.

Another form of decision elimination is to use lookup tables such as arrays, standard containers or customised containers[6, 7, 8]. Imagine if bus timetables were presented in terms of explicit decisions, either through switch or if else, based on time: they would be unreadable. Let the data structures do the work for you, and let the code make the decisions.

12. Prefer code to comments

Comments are a problem in the majority of systems I have reviewed. Not their absence, their presence. Sure, a good comment can be useful, but given the general hit-versus-miss rate across most commercial code that I've seen, such comments seem to be few and far between. The lingering belief that comments are somehow always a good thing seems to be adding drag rather than thrust to a lot of source code. This does not mean that all comments are necessarily bad, just that if you took a production system at random and removed all of the comments from its source, the result would probably be an improvement.

You can *hand redundant code its notice* as far as many comments are concerned. They add nothing to the code, and often detract from it. Comments that say nothing or simply parrot the code are simply a waste of good ASCII. Even worse is the accumulation of comments that are wrong: there is no value—to be precise, there is negative value—in having the source text lie to the reader.

Part of the problem is that comments are utterly non-functional and are bypassed by the compiler. Similarly, because they are non-functional and often incorrect or uninformative, the other main audience of the code, the programmer, often skips them as well. This can be exacerbated by syntax highlighting in editors: having comments in a separate colour makes it easier to ignore them, thus letting sleeping dogs lie. I know of many programmers who have written scripts to filter comments out of files or who switch the comment colour in their editor to be the same as or similar to the background colour to avoid being distracted.

In other words, it's not just a case of preferring good code to bad comments: you should prefer good code to good comments. This recommendation is an easy one to put into practice. Read your comments and if they give you news, not trivia, then keep them. Otherwise, reach for the delete key.

Safe and sound

The 12 recommendations presented in this column and the last are not intended to be exhaustive, but they do frame some principles that cover a lot of ground. For instance, why isn't there a specific recommendation to write short functions? Once you've applied all of the recommendations, that's by and large what you'll be left with. Short functions are a property, not a recommendation.

Like overly jumpy control flow, a piece of logic can sometimes accumulate mass like a snowball

Clearly, not all C++ practices that may be dear to your heart can be included. For example, there is no recommendation to be const-correct, and const-correctness won't just fall out of the practices as a by-product. It is certainly something you would want to fix, but its presence won't necessarily make the massive difference in attempting to eliminate bugs and regain control of the software that eliminating spurious code or twisted logic will. ■

References

1. Kevlin Henney, "Six of the best", *Application Development Advisor*, May 2002.
2. Kevlin Henney, "Making an exception", *Application Development Advisor*, May 2001.
3. Kevlin Henney, "One careful owner", *Application Development Advisor*, June 2001.
4. Kevlin Henney, "Collections for States", *Java Report*, August 2000, available from *www.curbralan.com*, as is the original EuroPLoP '99 paper in C++.
5. Kevlin Henney, "Promoting polymorphism", *Application Development Advisor*, October 2001.
6. Kevlin Henney, "Bound and checked", *Application Development Advisor*, January–February 2002.
7. Kevlin Henney, "Look me up sometime", *Application Development Advisor*, March 2002.
8. Kevlin Henney, "Flag waiving", *Application Development Advisor*, April 2002.