CHECKLIST for CLASS AUTHORS,
from RUMINATIONS in C++, by Andrew Koenig and Barbara Moo
(section headings added; data variable names changed to have trailing underscores)

// ==
// INTRODUCTION
// ==
Every good pilot uses a checklist. This fact sometimes raises naive passengers' eyebrows: Surely someone who knows how to fly the airplane has no need for such a thing! Nevertheless, a pilot who cares about safety will explain that, even though every checklist item is utterly routine, it is still important to have a memory jogger to guard against forgetting anything important during a moment of distraction. Even experienced professional pilots have been known to forget to lower their landing gear until reminded to do so by the warning horn or, worse, by the sound of the propellors destroying themselves on the runway.

A checklist is not a do-list. Its purpose is to help you recall things you might have forgotten, rather than to constrain you. If you blindly follow a checklist without thinking about it, you will probably forget things anyway. With that in mind, here are questions to ask about your classes as you define them. There are no "right" answers to these questions; the point is to get you to think about them, and to make sure that whatever you do is the result of a conscious decision, rather than an accident.

// ==
// SUMMARY
// ==

// ==
  I. DOES your CLASS NEED a CONSTRUCTOR?
// ==
If the answer is "no", you may be reading this chapter at an inopportune time. Some classes don't need constructors because they are so simple that their structure IS their interface. We are concerned here, however, primarily with classes that are complicated enough to require a constructor to conceal their inner workings.

// ==
 II. ARE your DATA MEMBERS PRIVATE?
// ==
Public data members are often a bad idea because the class author has no control over when those members are accessed. Consider, for example, a class that holds a variable-length vector.

```
template <typename T> class Vector {
 public:
     int length_;     // a public data member: a dubious idea
     // ...
  };
```

If the class author makes the length of the vector available as a member variable, the author must ensure that the member variable is kept up to date all the time, because there is no way to tell when the user of the class is going to want to access the information. If, on the other hand, the length is made available through a function such as

```
template <typename T> class Vector {
 public:
     int length() const;    // a much better idea
     // ...
 };
```

then it is unnecessary for the `Vector` class to calculate the length until the user calculates the length function. Even if that is not an issue for the present implementation, it may become one later.

Moreover, using a function instead of a variable makes it easy to block write access while still allowing read access. In the first version of the `Vector` class, there is nothing to stop the user from changing the length! It is true in principle that length could be made a const int, but not if is ever useful to change the length of a `Vector` object after that object has been created. We could use a reference to allow users read access only:

```
template <typename T> class Vector {
  public:             // each constructor _must bind_ length_ to true_length_;
     const int& length;
     // ...
  private:
     int true_length_;
  };
```

Indeed that would forestall the latter objection, but it still does not have the flexibility of making length a function from the start.

Even if the author of the `Vector` class intended to allow users to change the length of a `Vector`, making the length a public data member is not a good way to do so. Changing the length presumably requires allocating and deallocating memory, as well as copying the `Vector's` values somewhere else automatically. If the length is a variable that users can set directly, there is no way to detect immediately that a user has changed the length. By allowing users to change the length only by calling a member function, we make it possible to know immediately every time that the user changes the length.

There is no single "best" style for writing functions that access or change parts of a class. For example, assume that there are two functions to deal with a vector length. Should they have names such as `set_length` and `get_length`, or should they both be named `length` and distinguished by the number of arguments that they accept? Should the function that modifies the `length` return void or a value? If it returns a value, should that be the previous value of the length, or the new length that has just been installed? Or should it return something else entirely, such as an indication of whether it has been successful in dealing with the request? Pick a convention, use it consistently, and document it clearly.

// ==========
III. DOES your CLASS have a CONSTRUCTOR without ARGUMENTS?
// ==========

If a class has any constructors at all, and you want it to be possible to declare objects of that class without explicitly initializing them, then you must explicitly write a constructor that takes no arguments--for example,

```
class Point {
public:
    Point(int p, int q): p_(p), q_(q) {}
    // ...
private:
    int p_, q_;
  };
```

Here we have defined a class with a constructor. Unless this class has a constructor that doesn't require arguments, it will be illegal to say

```
Point  p;      // error: no initializer
```

because there will be no way to figure out how to initialize the object `p`. This may be an intentional part of the design, of course, but don't forget to think about it. Also, remember that if a class demands an explicit initializer, such as the `Point` class above, it is illegal to try to create an array of objects of that class:

```
Point  pa[100];    // error
```

Even if you want to require all instances of your class to be initialized, it is worth thinking about whether to prohibit arrays of objects of that class in exchange for that insistence.

// ==
IV. DOES every CONSTRUCTOR INITIALIZE every DATA MEMBER?
// ==

The purpose of a constructor is to put its object into a well-defined state. The state of an object is reflected by the object's data members. Therefore, it is the responsibility of every constructor to establish well-defined values for all the data members. If any constructor fails to do so, that is a likely sign of trouble ahead.

Of course, this assertion is not always true. Sometimes, classes will have data members that are not given values until their objects have existed for sometime. Remember, though, that the purpose of asking these questions is not to demand particular answers, but rather to encourage you to think about the questions.

// ==
V. DOES the CLASS NEED a DESTRUCTOR?
// ==

Not all classes with constructors need destructors. For example, a class that represents complex numbers probably does not need a destructor, even if it has constructors. If you think about what a class does, it should be obvious whether or not the class needs a destructor. It usually suffices to ask whether the class allocates any resources that freeing the members of the class will not free automatically. In particular, a class that has new-expressions in its constructors will usually have to have corresponding delete-expressions in its destructor--thus, it will probably need a destructor.

// ==
VI. DOES the CLASS NEED a VIRTUAL DESTRUCTOR?
// ==

Some classes need virtual destructors only to say that their destructors are virtual. Of course, a class that is never used as a base class doesn't need a virtual destructor. Virtual functions of any kind are useful only in the presence of inheritance. But suppose that you have written a class called, say, B, and someone else derives a class D from it. When does B need a virtual destructor?

```
class B {};
class D : public B {};
```

When anyone EVER does something like this:

```
D d;
B* b = &d;
delete b;
```

i.e., WHENEVER ANYONE EXECUTES a `delete` EXPRESSION on any OBJECT of TYPE `B*` that ACTUALLY POINTS to an OBJECT of TYPE `D`.

This is true even if neither `B` nor `D` has any virtual functions--for example

```
struct B     { std::string s; };
struct D : B { std::string t; };
```

```
int main() {
    B* bp = new D;  // no problem here, but...
    delete  bp;     // calls wrong destructor unless B's destructor is virtual
}
```

Here, even though B has no virtual functions, and indeed has no member functions at all, the delete will go awry unless B is given a virtual destructor.

```
struct B {    // addendum: no public needed here--structs are default public
    std::string  s;
    virtual ~B()  {}
};
```

Virtual destructors are often empty.

// ==
VII. DOES your CLASS NEED a COPY CONSTRUCTOR?
// ==

The answer is often, but not always, "no". The key question is whether there is any difference between copying an object of this class and copying its data structures and base classes. You need a copy constructor only if there is a difference.

If your class allocates resources in its constructor, then it probably needs an explicit copy constructor to manage these resources. A class with a destructor (except for an empty virtual destructor) usually has that destructor because it needs to free resources that its constructor allocated, which usually implies that it needs a copy constructor as well. A classic example is a String class:

```
class String {
  public:
      String();
      String(const char* s);
      // other member functions
  private:
      char*  data_;
  };
```

Even without looking at the rest of this class definition, we can guess that it needs a destructor, because its data member points to dynamically allocated memory that must be freed with the corresponding object. It needs an explicit copy constructor for the same reason. Without one, copying a String object will be implicitly defined in terms of copying its data_ member. After the copy, the `data_` members of both objects will point to the same memory; when the two objects are destroyed, that memory will be freed twice!

If you don't want users to be able to copy objects of a class, make the copy constructor (and probably also the assignment operator) private:

```
class Thing {
public:
    // ...
private:
    Thing(const Thing&);
    Thing& operator=(const Thing&);
};
```

If you don't use these member functions from other members, it is sufficient to define them as shown. There is no need to define them, because no one calls them: You didn't, and the user can't.

// ==
VIII. DOES your CLASS NEED an ASSIGNMENT OPERATOR?
// ==
If it needs a copy constructor, it probably needs an assignment operator for the same reason. If you don't want users to be able to assign objects of your class, make the assignment operator private.

Assignment for a class `X` is defined by `X::operator=`. Usually, `operator=` should return an X& and end by saying:

```
return  *this;
```

for consistency with the built-in assignment operators.

// ==
IX. DOES your ASSIGNMENT OPERATOR HANDLE CORRECTLY ASSIGNMENT of an OBJECT to ITSELF?
// ==
Self-assignment is mishandled so often that more than one C++ book has gotten it wrong. Recall that assignment always replaces an old value in its destination by a new one. If we free the old destination value before copying the source value, and the source and destination are the same object, then we might wind up freeing the source before we copy it. For example, consider a `String` class:

```
class String {
public:
    String& operator=(const String& s);
    // various other members
private:
    char*  data_;
};
```

It can be tempting to implement assignment this way:

```
// obvious, but incorrect implementation
String& String::operator=(const String& s) {
    delete[] data_;
    data_ = new char[strlen(s.data_) + 1];
    strcpy(data_, s.data_);
    return  *this;
}
```

This approach fails horribly when we assign a String object to itself, because and *this both refer to the same object. The simplest way to avoid the problem is to guard against it explicitly.

```
// correction 1
String&  String::operator=(const String& s) {
    if (&s != this) {
        delete[] data_;
        data_ = new char[strlen(s.data_) + 1];
        strcpy(data_, s.data_);
    }
    return  *this;
}
```

Another possibility is to save the old destination value until after you have copied the source value:

```
// correction 2
String& String::operator=(const String& s) {
    char* newdata = new char[strlen(s.data_) + 1];
    strcpy(newdata, s.data_);
    delete[] data_;
```

```
        data_ = newdata;
        return  *this;
    }
```

// ==
X. DOES your CLASS NEED to DEFINE the RELATIONAL OPERATORS?
// ==

Now that C++ supports templates, general-purpose libraries increasingly contain container classes. The classes provide generic definitions of data structures, such as lists, sets, and graphs. These containers rely on operations from the types they contain. A common requirement for containers is the ability to determine whether a value is less than or greater than some other value.

Thus, if your class logically can support equality, it may be useful to provide `operator==` and `operator!=`. Similarly, if there is some ordering relation on values of your class, you may want to provide the rest of the relational operators. You may need these operators even if you don't expect users to use the relational operations directly. As long as they want to create ordered collections of your type, you'll need to provide the relational operators.

// ==
XI. DID you REMEMBER to SAY `delete[]` (array delete) when DELETING an ARRAY?
// ==

Some early C++ literature suggested that the copy constructor for class X should be `X::X(X&)`. This suggestion is incorrect: it should be `X::X(const X&)`. After all, copying an object does not modify the original! In fact, because it is illegal to bind const to a temporary, using `X::X(X&)` as a copy constructor would not allow the result of any nontrivial expressions to be copied.

The same argument applies to assignment. Use `X::operator=(const X&)`, and not `X::operator=(X&)`.

// ==
XII. WHEN a FUNCTION has REFERENCE PARAMETERS, SHOULD THEY be `const` REFERENCES?
// ==

The only time that a function should have a reference argument without saying const is when it intends to modify the argument. Thus, for example, instead of saying

```
    Complex operator+(Complex& x, Complex& y);
```

you should almost certainly say:

```
    Complex operator+(const Complex& x, const Complex& y);
```

unless you mean to allow adding two `Complex` values to change their values! Otherwise, expressions such as `x + y + z` become impossible, because x + y is not an lvalue, and therefore may not have a non const reference bound to it.

// ==
XIII. DID you REMEMBER to DECLARE your MEMBER FUNCTIONS `const` APPROPRIATELY?
// ==

When a member function is guaranteed not to modify its object, declare it const so that it can be used for `const` objects. Therefore, if we use access functions as in our earlier example:

```
 template <typename T> class Vector {
    public:
        int length();       // get length (wrong)
        int length(int)     // set length, return previous value
        // ...
      };
```

the function that fetches the length without changing it should be declared `const`:

```
template <typename T> class Vector {
  public:
      int length() const;  // get length (right)
      int length(int)      // set length, return previous value
      // ...
    };
```

Otherwise, we will run into trouble in contexts such as this:

```
/* return the larger of n and the length of v */
template <typename T>
int  padded_length(const Vector<T>& v, int n) {
    int k = v.length();    // oops!
    return  k > n ? k : n
}
```

The line flagged OOPS! won't compile unless the aforementioned const appears in the declaration of length, because `v` is a reference to `const`.

// ==
// CONCLUSION
// ==
This list naturally raises the question of why C++ doesn't somehow handle all these things naturally. The main reason is that it's not possible to do so and still be assured of being right.

Continuing our earlier aviation analogy, making these things automatic would be like designing airplanes to lower their landing gear automatically under the right circumstances. People have tried doing that, with limited success. The trouble is "the right circumstances" are hard to define in a way that can be reliably handled mechanically. Therefore, in practice, the pilot has to check the "gear down" light before every landing anyway, so the automation doesn't buy much. Of course, leaving the gear extended all the time would solve the problem, but only by introducing unacceptable overhead.

The same argument applies to programming-language design. For example, giving a class a virtual destructor adds overhead. If the class is tiny and has no (other) virtual functions, that overhead can be significant. Causing all classes to have virtual destructors automatically would violate the C++ philosophy of making people pay for only what they use; it would be analogous to leaving the landing gear extended all the time.

The alternative is to have the compiler somehow figure out when a class should have a virtual destructor, and to supply that destructor automatically. Again, the problem lies in defining exactly when to generate such a destructor. If the definition is not perfect, programmers will have to check it anyway. Checking will be at least as much work as defining the virtual destructor in the first place.

In other words, C++ is biased towards programmers who think for themselves. Using C++ requires thought, in much the same way as does using any professional tool.

<<END>>