EECS 211
C++ Style

Follow the rules below to write clean simple C++ code. For further examples of good style, see Ol' Doc McCann's List of Programming Style Guidelines.

Indent consistently

There are several indenting styles commonly used by C and C++ programmers. See this Wikipedia entry for a list. The book uses the Allman (BSD) style. I use the K & R style. Either is fine but **be consistent**.

In all styles, one cardinal rule of indentation is followed:

> Indentation = nesting

Nested code, e.g., statements in a for-loop, is always indented some constant number of spaces (2, 3, or 4 are typical) from the nesting element, e.g., the start of the for-loop or if-statement. Statements nested at the same level have the same indentation. The following is very bad indentation:

```
   void square_array(int x[], int n, int y[])
{
for (int i = 0; i < n; ++i) {
   int z = x[i];
     y[i] = z * z;
      }
      }
```

The function heading is not nested and should not be indented. The for-loop is nested and should be indented. The `y[i] = ...` line should be indented the same as the `int z ...` line. The close brace for the for-loop should be under the 'f' for the `for`. The close brace of the function should be under the 'v' of `void`. The proper indentation for this code is:

```
void square_array(int x[], int n, int y[])
{
  for (int i = 0; i < n; ++i) {
    int z = x[i];
    y[i] = z * z;
  }
}
```

>

Initialize Variables

When a variable has an initial value, assign it in the declaration of the variable, not in a separate assignment statement. E.g., don't write

```
int main (void)
{
  int a;

  a = 4;
  ...
}
```

write

```
int main (void)
{
  int a = 4;
  ...
}
```

It's shorter, clearer, and prepares you for C++ where, with some data types, it's more efficient.

>

Initialize Class Variables Directly

When initializing a variable holding an instance of a class, write

```
MyClass thing( value );
```

not

```
MyClass thing = value;
```

The second form is equivalent to

```
MyClass thing ( MyClass( value ) );
```

This creates a temporary then copies it. While most compilers will optimize the second form into the more efficient first form, there are times when they can't. So use the first form and be sure.

This does not apply to basic numeric datatypes and such. `int n = 12;` is more typical than `int n( 12 );`.

Locally declare for-loop variables

Instead of

```
int i;

for (i = 0; i < n; ++i) {
  ...
}
```

write

```
for (int i = 0; i < n; ++i) {
  ...
}
```

It's shorter and keeps the scope of `i` as narrow as possible. A code maintainer can tell that this `i` is used only in this loop and nowhere else.

Name Constants

Don't scatter literal values like 12 and 52 and 31 throughout your code. Define named constants at the start of the file, using `const type variable = value;`.

Define A Function for Each Task

It has been said that the major task of a programmer is managing complexity. Code grows quickly and it's important to keep it modular and self-documenting, for easy repair and extension. Well-named modular functions are central to writing good code.

Define a function when

- The same task, e.g., finding something in an array, occurs more than once.
- A function gets more than half a dozen lines long. It almost certainly has at least one, if not several, functions lurking inside it.
- You need to put comments on code to explain what it's doing.

Use Good Names

Nothing is more important to readable code than good names. It's also the case that naming is often done very badly. See the Naming section of Roedy Green's How to Write Unmaintainable Code for examples of what not to do.

Here are some guidelines for good names:

- For a function name, use a verb or a verb-noun to say what a function returns or does.
  - Good: `isLeapYear`, `printDate`, `findLargest`.
  - A name should never say how a function does it, e.g., `printWithLoop`.
- For a variable name, use a noun that describes the kind of data a variable holds.
  - Good: `averages`, `currentDate`, `numGrades`.
  - A name should normally not specify the data form, e.g., `cardArray`.
- For a variable holding generic data, use a standard generic name, such

as

- `i` for a for-loop index, and `j` for a nested for-loop index
- `x`, `y` and `z` for generic numbers, especially floating point
- `n` for a generic integer

Avoid Needless Variables

Declare variables to

- avoid recalculations
- give meaningful labels to intermediate results

If neither of these applies, don't create a variable. For example, in

```
int twoDigits(int number)
{
  int result;

  result = firstDigit(number) + secondDigit(number);
  return result;
}
```

the variable is absolutely useless. Write

```
int twoDigits(int number)
{
  return firstDigit(number) + secondDigit(number);
}
```

Avoid Repeated Code

Never repeat code for anything at all complex. Every copy of such code will have to be updated if you find a bug or want to make an improvement.

The two ways to avoid repeated code are:

- variables, to store calculation results
- functions, to hold common code patterns

Use double not float

`double` is the default floating point type for C++. If you write the number 3.0 in C++ code, it's a double, not a float. Double has the best trade-off between precision and use of space for most code.

Avoid "using" declarations in header files

Note: The Deitel book vioates this rule frequently.

A "using" declaration, such as `using std::string;` can be very useful in avoiding clutter in your code. It lets you write `string str = "foo";` rather than `std::string str = "foo";`.

Such declarations should only appear in implementation code, i.e., `.cpp` files. They

should never appear in header files. "Using" declarations in header files can cause name conflicts. If one header file has `using std::string;` and another has `using astronomy::string;`, then there will be a name conflict when both header files are included by another file. The whole point of namespaces was to avoid such collisions.

"Using" declarations are fine in `.cpp` files, because those files are compiled independently.

Include "header guards" in all header files

Always surround the code in a header with a guard form:

```
#ifndef NAME_H
#define NAME_H
...
...
#endif
```

where *NAME* is the uppercase form of the header file name, e.g., `COMPLEX_H` for the header file `complex.h`.

The guard form prevents the header code from being read more than once, no matter how many files are compiled that `#include` it. This is not just for efficiency. Compiler errors will occur when a header is read multiple times.

Use member initializer lists in constructors

Whenever possible (which is almost always), initialize private data members like this:

```
Complex::Complex(double r, double i) : real(r), imag(i) { }
```

not like this:

```
Complex::Complex(double r, double i)
{
  real = r; imag = i;
}
```

Initializers are not only clearer, but they also avoid creating default instances when data members hold user-defined data structures.

> Initialization is done in the order in which the variables are declared. To avoid warning messages, be sure to list variables in declaration order in the member initialization list.

Declare destructor as virtual if any member function is virtual.

For safety, if a class has any virtual member functions, then make the destructor virtual too, like this:

```
class SomeClass
{
  ...
  virtual ~SomeClass();
  ...
}
```

This is explained in Section 13.9, and in many places online, such as here.

Use typedef's to clarify code using generic containers

The standard library containers are great for producing efficient robust code quickly. But the template notation can clutter up your code pretty quickly. Use **typedef** to define clear names for each container-based object. This will simplify the code overall, and make it much easier to change which container you're using.

So, instead of this

```
set<string> names;
for (set<string>::const_iterator i = names.begin;
     i != names.end();
     ++i)
{
  ...
}
set<string>::iterator found = find(names.begin(), names.end(), "John");
if (found != names.end()) {
  ...
}
```

Write this

```
typedef set<string> List;
typedef List::const_iterator ConstListIter;
typedef List::iterator ListIter;

for (ConstListIter i = names.begin; i != names.end();  ++i)
{
  ...
}

ListIter found = find(names.begin(), names.end(), "John");
if (found != names.end()) {
  ...
}
```

This becomes particularly useful with the map containers. See Chapter 22 in Deitel for more examples.

Comments?  Contact the Prof!