Style in C++

For explanations, see: http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Classes

Basic principles

Readability is the most important attribute of style

Choose good names. The most direct way of explaining what a program is about is by selecting good names for variables, types, procedures. If I'm trying to write some new code, and I can't figure out a good name for a class or method, I'll often be stuck until I have something. Many times, I find that the reason I can't pick a good name is that I don't understand enough about what I'm trying to say; with understanding comes good names, and vice versa.

Code so that changes and understanding will be easier.

Formatting

1. Indent to show organizational structure of code. Two spaces is a good indentation amount. Any larger and you risk needs lots of line wrapping.
2. Blank lines improve readability by setting off sections of code that are logically related. Use blank lines minimally.
3. Do not place multiple statements on the same line.
4. Do not exceed 80 characters on a line.
5. Indent continuation lines.
6. One declaration per line is recommended since it encourages commenting.
7. Comments should always use // so that /* and */ can be used to comment out whole sections of code.
8. Compound statements are statements that contain lists of statements enclosed in braces "`{ statements }`". The enclosed statements should be indented one more level than the compound statement.
9. switch Statements should have the following form:

```
switch (condition) {
case ABC:
    statements;
    /* falls through */

case DEF:
    statements;
    break;

case XYZ:
    statements;
    break;

default:
    statements;
    break;
}
```

Every time a case falls through (doesn't include a `break` statement), add a comment where the `break` statement would normally be. This is shown in the preceding code example with the `/* falls through */` comment.

Naming The most important consistency rules are those that govern naming. The style of a name immediately informs us what sort of thing the named entity is: a type, a variable, a function, a constant, a macro, etc., without requiring us to search for the declaration of that entity. The pattern-matching engine in our brains relies a great deal on these naming rules.

1. Interface/Class Names
    a. Begin with an uppercase letter and have a capital letter for each new word, with no underscores: `MyExcitingClass`, `MyExcitingEnum`.
    b. Each class should be in a separate file named using the *ClassName*.
    c. Every class should have a separate .h file.
    d. Keep your class names simple and descriptive.
2. Methods:
    a. Method names should be verbs, in mixed case **with the first letter lowercase**, with the first letter of each internal word capitalized. For example, bestSoFar or totalIncome. [Another choice is to separate words with an underscore, best_so_far or total_income.]
    b. Write methods that only do ``one thing''. This not only aids reusability, but it makes documentation easier.
    c. Avoid overloading methods on argument type.
    d. Methods to get and set an attribute variable V have names getV and setV.
    e. A method to return the length of something should be named length
    f. A method that tests a boolean condition V should be named isV. Example: isInterrupted.
    g. A method that converts its object to a particular format F should be named toF. Example: toString
    h. Whenever possible, base names of methods in a new class on names in an existing class that is similar. This makes it easier for the reader to remember.
    i. Abbreviations and acronyms must **not** be uppercase when used as name exportHtmlSource(); *// NOT: exportHTMLSource();*.
    j. Complement names must be used for complementary operations: get/set, add/remove, create/destroy, start/stop, insert/delete, increment/decrement, old/new, begin/end, first/last, up/down, min/max, next/previous, old/new, open/close, show/hide, suspend/resume
3. Constant Names: all names must be explained via a comment
    a. Constant names: There are two choices here
        i. Use a `k` followed by mixed case: `kDaysInAWeek`.
        ii. Use SOLID caps with an underscore between words: DAYS_IN_WEEK
    b. Floating point constants should always be written with decimal point and at least one decimal.
    c. Enumeration constants can be prefixed by a common type name. This gives additional information of where the declaration can be found, which constants belongs together, and what concept the constants represent.
       enum Color { COLOR_RED, COLOR_GREEN, COLOR_BLUE };
4. Local Names & Parameters: all names must be explained via a comment
    a. Variables should have names which are nouns or noun phrases.
    b. Plural form should be used on names representing a collection of objects.
       Use abbreviations sparingly. cp may mean "copy" to you, but the reader may spend considerable effort trying to decide what the abbreviation stands for.
    c. Use descriptive parameter names. Parameter names should be descriptive enough that the name of the parameter and its type can be used to determine its meaning in most scenarios.
    d. Avoid giving a variable the same name as one in a superclass as this is usually an error.
    e. Boolean variable names should contain *Is* or other words which implies Yes/No or True/False values, such as IsFinished. Often names can be chosen either in a positive (e.g., isValid or isComplete) or negative (isInvalid) form. Always use the positive form.
    f. C++ pointers and references should have their reference symbol next to the type rather than to the name.

```
float*  x;    // NOT: float *x;
int&  y;    // NOT: int &y;
```

Methods

1. Always validate method parameters. Validating parameters is the first task that should be performed by the procedure.
2. Document cases where the return value of a called method is ignored. These are typically errors. If it is by intention, make the intent clear. A simple way to do this is: int unused = obj.methodReturningInt(args)
3. Each method must have a few lines of comments which explain what parameters are needed and what the method accomplishes.

Comments

1. Pay attention to punctuation, spelling, and grammar; it is easier to read well-written comments than badly written ones.
2. Every file should have a comment at the top describing its contents.
3. Every class definition should have an accompanying comment that describes what it is for and how it should be used.
4. Every function declaration should have comments immediately preceding it that describe what the function does and how to use it

```
// Returns true if the table cannot hold any more entries.
 bool IsTableFull()
```

5. In general the actual name of the variable should be descriptive enough to give a good idea of what the variable is used for. In certain cases, more comments are required. However, I would encourage you to add a comment for almost every variable.

```
//Rotate grid north
//column: column which is rotating
 void Board::rotateNorth(int column){
   if (column <0||column >=SIZE) return;
   int wrap = board[0][column];  //value on board that wraps around
   for (int i = 0; i <SIZE-1; i++)
           board[i][col] = board[i+1][column];
   board[SIZE-1][col] = wrap;
 }
```

6. In your implementation you should have comments in tricky, non-obvious, interesting, or important parts of your code.
7. Note that you should *never* describe the code itself. Assume that the person reading the code knows C++ better than you do, even though he or she does not know what you are trying to do:

```
// Now go through the b array and make sure that if i occurs,
// the next element is i+1.
...        // Geez.  What a useless comment.
```

8. When you pass in a null pointer, boolean, or literal integer values to functions, you should consider adding a comment about what they are, or make your code self-documenting by using constants.

```
bool success = CalculateSomething(interesting_value,
                                  BASE_VALUE,
                                  false,  // Not the first time we're calling this.
                                  NULL);  // No callback.
```

Readability

1. Declare all constants (except for 0, 1, and 2) as const. Unnamed numeric constants, termed *magic numbers*, hinder maintenance as well as readability. This means you cannot use the literal ``5'' (for example) in your code. The reader asks, ``Why 5?'' Defining a constant with the value of 5 allows the programmer to explain why the value of 5 is used.
2. It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems.

3. Avoid assignments (``=") inside `if` and `while` condition. These are often typos and (even when correct) hinder readability.
4. Try to make the structure of your program match the intent. Example:

   ```
   if (booleanExpression) {
   return true;
   } else {
   return false;
   }
   ```

   should instead be written as

   ```
   return booleanExpression;
   ```

5. To increase readability, an empty for statement should have the following form:

   ```
     for (initialization; condition; update)
          ;
   ```

6. Use names you can pronounce. People talk about programs. It's easier to talk about code if you can pronounce the words inside of it.
7. The use of break and continue in loops should be avoided.
8. The form while(true) should be used for infinite loops.
9. Complex conditional expressions must be avoided. Introduce temporary (meaningful) boolean variables instead

   ```
   bool  isFinished = (elementNo < 0) || (elementNo > maxElement);
   bool  isRepeatedEntry = elementNo == lastElement;
   if (isFinished || isRepeatedEntry) { …}

   // NOT: if ((elementNo < 0) || (elementNo > maxElement)|| elementNo == lastElement) { … }
   ```

10. Loop variables should be initialized immediately before the loop.

    ```
    isDone = false;
    while (!isDone) {
    ```

Class Organization

1. In general, every `.cpp` file should have an associated `.h` file. Correct use of header files can make a huge difference to the readability, size and performance of your code.
2. All header files should have `#define` guards to prevent multiple inclusion.
3. Define functions inline only when they are small, say, 10 lines or less.
4. Place a function's variables in the narrowest scope possible, and initialize variables in the declaration.
5. If your class defines member variables, you must provide an in-class initializer for every member variable or write a constructor (which can be a default constructor). If you do not declare any constructors yourself then the compiler will generate a default constructor for you, which may leave some fields uninitialized or initialized to inappropriate values.
6. Only very rarely is multiple implementation inheritance actually useful. We allow multiple inheritance only when at most one of the base classes has an implementation; all other base classes must be pure interface classes tagged with the `Interface` suffix.
7. Do not overload operators except in rare, special circumstances
8. Use C++ casts like `static_cast<>()`. Do not use other cast formats like `int y = (int)x;` or `int y = int(x);`.
9. Use `0` for integers, `0.0` for reals, `nullptr` (or `NULL`) for pointers, and `'\0'` for chars.