Style in C++

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, methods. Finding a good name is sometimes challenging. Often it is a question of understanding. With understanding comes good names, and vice versa.

Code so that changes will be easier.

Formatting

1. Indent to show organizational structure of code. Four or less spaces is a good indentation amount. Any larger and you risk lots of line wrapping.
2. Blank lines improve readability by setting off sections of code that are logically related.
3. Two blank lines should always be used in the following circumstances:
   a. Between sections of a source file
   b. Between class and interface definitions
4. One blank line should always be used in the following circumstances:
   a. Between methods
   b. Between the local variables in a method and its first statement
   c. Before a block or single-line comment
   d. Between logical sections inside a method to improve readability
5. Do not place multiple statements on the same line.
6. Do not exceed 80 characters on a line. Indent continuation lines.
7. One declaration per line is recommended since it encourages commenting.
8. Comments should always use // so that /* and */ can be used to comment out whole sections of code.
9. 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.
10. There are several appropriate styles for placing brackets and indenting. Pick a style you like and be consistent.

K&R style

```c
int main(int argc, char *argv[]) {
    ...
    while (x == y) {
        something();
        somethingElse();
        if (someError)
            doCorrect();
        else
            continueAsUsual();
    }
    finalThing();
```

The One True Brace Style

// ...
if (x < 0) {
    printf("Negative");
    negative(x);
} else {
    printf("Positive");
    positive(x);
}

Allman Style
while(x == y)
{
    something();
    somethingElse();
}
finalThing();

11.switch Statements should have a version of the following form:

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

1. Interface/Class Names
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.
   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 complement 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. Names should be sequence of one or more words all uppercase, components separated by underscore. Examples: MIN_VALUE, MIN_RADIX. Underscores are shunned by many (as they are more difficult to type), but when using solid caps, underscores are needed to separate the words.
   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 };
   d. Local Names & Parameters: all names must be explained via a comment
   a. Fields 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 so 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.

Readability
1. Declare all constants (except for 0, 1, and 2) as const. Mysterious 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:

```java
if (booleanExpression)
    return true;
else
    return false;
```

should instead be written as

```java
return booleanExpression;
```

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

```java
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 boolean variables instead

```java
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.

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