Style Guidelines for C++

Following a clean and clear coding style is very helpful for the readability and maintainability of your code. Some aspects of a coding style refer to what needs to be commented and how to format those comments, and also what coding patterns should and shouldn't be used. Style-guidelines like these are often very important for the maintainability of code, and they often have very good reasons behind them.

By the same token, other aspects of coding-style are more aesthetic in nature, and they really don't have as much to do with correctness. However, these style-guidelines still can improve the maintainability of your code, because it will make your code easier to read and understand. This can be especially important when you are working on large coding projects with multiple programmers. In such cases, a consistent coding style can make it much easier to read and understand other people's code. It is like formatting a document - if the whole document follows the same formatting conventions, it is just easier to read than if each part has its own format!

People also usually end up settling on a "favorite" coding style. This is nice when you have the luxury of controlling the coding standard, but in many circumstances there will already be a coding standard in place. In those cases, it's important to be flexible.

1. **Use spaces after commas and semicolons. Don't use spaces before commas and semicolons.**

```
for (int i = 1 ;i < 5;i++) {
  foo(x1,x2 ,x3);
}
```

   vs.

```
for (int i = 1; i < 5; i++) {
  foo(x1, x2, x3);
}
```

2. **Use spaces around binary operators, except ->. Don't use spaces with unary operators.**

```
for (int i=1; i<5; i++) {
  x=y+z*15;
  foo->callFunction();
}
```

   vs.

```
for (int i = 1; i < 5; i++) {
  x = y + z * 15;
  foo->callFunction();
}
```

3. **Use a consistent placement-style for curly-braces..**

   There are two common ways to place curly-braces:

```
for (int i = 1; i < 5; i++) {
  //code here;
}
```

   or

```
for (int i = 1; i < 5; i++)
{
  //code here;
}
```

   Choose one way and stick with it.

4. **Use 2, 3, or 4 spaces for each indentation level.**

   Indent only with spaces, not tabs.

   Don't use hard-tabs for 8 spaces. Most text editors provide a way to configure whether 8 spaces are converted to a "hard tab", or whether they are left as spaces ("soft tabs"). When you mix spaces and tabs, it can get ugly!

   Be consistent about applying indentation to nested blocks of code. A good editor will make this much easier.

5. **Make code fit to 80-character column widths.**

   Even if you have a wide screen, other people might not. It also makes it much easier to format your code for printing, which often occurs for the purposes of debugging or code-review.

   Put line breaks at appropriate points to help make code easier to read.

6. **Use parentheses to clarify precedence.**

   You can assume that the reader of your code understands the basic rules of precedence, like addition and multiplication, or logical ANDs and ORs. But if you have more complicated expressions, or you are using less common operators (e.g. bitwise logical operators, or the ? : operator, etc.), use parentheses to clarify the precedence.

```
int x = y * z - p * q >> 2;
if (something == TRUE && anotherthing == FALSE) {
  //do something
}
```

   vs.

```
int x = (y * z - p * q) >> 2;
if (something == TRUE && anotherthing == FALSE) {
  //do something
}
```

7. **Follow a consistent name-capitalization pattern.**

   C++ allows any capitalization for all names. However, please do not capitalize the first letter of variable names. Do capitalize the first letter of class names.

   There are any number of rules for names that contain multiple words, such as camelCase, UpperCamelCase, using_underscores, etc. Again, pick one style and use it consistently within your code.

8. **Use blank lines to separate units of functionality.**

   Don't be afraid to use blank lines to help make code blocks more readable. In much the same way that you break pieces of thought into paragraphs in a writing class, do the same with code within a function. Comments often help delineate these "paragraphs" as well.

   For example, use one blank line to separate units of computation within a function body. Use two blank lines to separate function definitions, or class declarations, etc.

```
void foo() {
  int x = fribbery;
  y.attack(frabbet(x));
  z.attack(frobbet(x));
  bejooger[0].scramboozle(y);
  bejooger[1].scramboozle(y);
  bejooger[2].scramboozle(z);
  bejooger[3].scramboozle(z);
  if (curdleblogpod.getAlive() <= 5) {
    //Yikes!
  }
}
```

vs.

```
void foo() {

  //Start the attack on the fribs.
  int x = fribbery;
  y.attack(frabbet(x));
  z.attack(frobbet(x));

  //Now send the bejoogers out...
  bejooger[0].scramboozle(y);
  bejooger[1].scramboozle(y);
  bejooger[2].scramboozle(z);
  bejooger[3].scramboozle(z);

  //Make sure the curdleblogs are still alive.
  if (curdleblogpod.getAlive() <= 5) {
    //Yikes!
  }
}
```

9. **Use brackets if nested blocks are longer than one physical line.**

   C++ doesn't require a pair of brackets around an enclosed block if it is only one "statement" long -- so you can nest a `for`-loop inside of an `if` statement, for example, with no enclosing brackets. However, don't take advantage of this unless the enclosed block of code is only one physical line long, and even then, only if that line is "simple."

```
if (foo)
  for (int i = 0; i < 3; i++) {
    cout << "Hello there, " << i << endl;
    doSomething(i >> 2);
  }
```

vs.

```
if (foo) {
  for (int i = 0; i < 3; i++) {
    cout << "Hello there, " << i << endl;
    doSomething(i >> 2);
  }
}
```

10. **Put spaces after keywords like `for`, `if`, `while`, etc.**

```
if(foo) {
  while(true) {
    //do some stuff
  }
}
```

vs.

```
if (foo) {
  while (true) {
    //do some stuff
  }
}
```