# C++ Reference Material
# Programming Style Conventions

What follows here is a set of reasonably widely used C++ programming style conventions. Whenever you move into a new programming environment, any conventions you have been using previously may or may not agree with the ones you are required to use in the new environment, so you should examine the new conventions closely to see what is the same, and what is different, from what you are used to. For the sake of brevity, only a few specific examples to illustrate the style conventions described here are given on this page, but many code and formatting examples are collected in a series of separate code examples. You should take the style conventions you are required to use very seriously, because your instructor and your marker(s) (and, eventually, your employer) will surely do so.

## Part A: Program Readability (Source Code Readability)

### The "Big Three" of Programming Style (according to Scobey)

1. Name things well, using the specified style conventions.

2. Be consistent.

3. Format continually, and re-format whenever necessary.

*The importance of choosing a good name for each programmer-named entity, thereafter spelling and capitalizing that name appropriately and consistently, and continuing to position these names properly within well-formatted code, cannot be overemphasized.*

### Rules for naming and capitalization of programmer-chosen identifiers

#### Naming

1. Names must be meaningful within their given context whenever possible, which is most of the time. Two exceptions to this rule are:

   - The use of single-letter loop control variables such as `i` in those situations where no particular meaning attaches to the variable.

   - Use of a generic variable name such as `x` for reading in values (real numbers, say) to which no particular meaning applies.

2. Variables and value-returning functions are generally noun-like, as in:

```
3.   length
4.   numberOfStudents
```

```
5.   averageScore()
```

Boolean variables and functions, however, are usually adjective-like, as in:

```
valid
isFinished()
```

Special (sometimes global, if permitted) boolean variables that act as a switch to turn testing or debugging on or off begin with the word `testing` or the word `debugging`, as appropriate:

```
testingTextItems or testingTextItemsGLOBAL
debuggingDisplayOutput or debuggingDisplayOutputGLOBAL
```

Appending the all-caps GLOBAL to indicate a variable acting globally as a switch to turn testing or debugging on or off is yet another way to be as informative as possible with a variable name. When one is faced with the choice between long variable names and clearer code, or short variable names and less clear code, the choice should be obvious.

6. The name of every `void` function (both free functions and member functions) *must* begin with a *verb*, as in:

```
7.   GetInputFromUser()
8.   DisplayOutput()
9.   time.incrementHours()
10.  counter.display()
```

Occasionally value-returning functions *may* also begin with a verb. One particular example of this is any "getter" member function that appears in a class:

```
time.getHours()
```

**Capitalization**

Names must always be capitalized consistently, according to whatever conventions are being used for each category of name. Here are some examples that illustrate our capitalization conventions:

1. Constants use all uppercase letters, with the words in multi-word names separated by underscores, as in:

```
2.   const int SIZE = 100;
3.   const double TAX_RATE = 0.15;
```

4. Variables use "camel" notation, which means that names start with a lowercase letter, and all subsequent words in the name, if any, begin with an uppercase letter, as in:

```
5.   double cost;
6.   int numberOfGuesses;
```

7. Macro definitions used in header files to avoid multiple inclusion are a special case, and their capitalization is illustrated by this example for a file (`menu.h`, say) containing the specification of a `Menu` class:

```
8.   #ifndef MENU_H
9.   #define MENU_H
10.    ...
11.  #endif
```

12. Free functions start with a lowercase letter, except for void functions, which start with a capital letter (and, of course, with a verb, as we pointed out above). Some examples:

```
13.  double average();
14.  int numberOfDigits();
15.  void Swap();
16.  void DisplayMenu();
```

17. Member functions names *always* start with a lowercase letter, as in:

```
18.  int time.getHours();
19.  void time.setHours();
20.  void time.display();
```

21. Type names start with an uppercase letter, use the "camel" notation described above, and names for classes should be nouns that are usually singular (but not always, as in `TextItems`}. Some examples:

```
22.  Menu
23.  TextItems
24.  RandomGenerator
25.  String80
```

## Rules for indentation and alignment

1. Use an indentation level of 4 spaces, and *always* use actual spaces, *never* the TAB character.

2. Indent and align each of the following:

   a. Statements in the body of a function (with respect to the corresponding function header)

   b. Statements in the body of a loop (with respect to the loop keyword(s))

c. Statements in the body of an `if` and, if the `else` is present, in the body of the `else` (The statements in the body of an `if` should always align with those in the body of the corresponding `else`, and the `if` and `else` must themselves align, unless the entire `if..else` construct is a short one on a single line.)

d. Field definitions in the definition of a `struct`

e. Member definitions in the definition of a `class`

3. Each level of nesting requires another level of indentation.

4. Align enclosing braces vertically with each other, except in very rare cases. [A short array initialization comes to mind as an exception.] Situations where braces *must* be aligned vertically include:

   a. The braces enclosing a function body with the function header

   b. The braces enclosing a loop body in a `while`, `do..while` or `for` loop

   c. The braces enclosing the body of an `if`, `else` or `switch` with the corresponding `if`, `else` or `switch`

   d. The braces enclosing the body of a `struct` or `class` with the keyword `struct` or `class` and with the access specifier(s) `public`, `private` and/or `protected` (if present)

5. Align comments with the thing commented (unless, of course, it is an in-line comment.

6. If a statement extends over more than one line, indent and align the second and any subsequent lines for readability (which may sometimes mean ignoring the 4-space-indentation rule for those lines).

**Use of whitespace**

1. Use vertical spacing to enhance readability. For example, use one or more blank lines to separate logically distinct parts of a program. In general, don't be afraid to use more vertical whitespace, rather than less.

2. Use horizontal spacing to enhance readability. For example, it is usually a good idea to place a blank space on each side of an operator such as `<<` or `+`, though this rule can be relaxed in a long complex expression where the enclosing spaces may be removed from some of the operators to show more clearly which operands are associated with particular operators. A typical situation where this might be done is to use

```
3.   for (int i=start; i<=finish; i++) ...
```

rather than

```
for (int i = start; i <= finish; i++) ...
```

for example, where it might be argued that the first is (in this case) slightly more readable.

4. Some programmers prefer to use a style that is almost "functional" in nature in their writing of selection and looping statements, by omitting the space between the selection or looping keyword and the following left parenthesis, as in

```
5.   if(condition) ...
6.   while(condition) ...
7.   for(int i=0; ... )
```

and so on. This is also permissible, but if used at all it must (of course) be used consistently.

**Comments**

1. When commenting your code, you should strive to be informative without being excessive. Your goal is to have readable code, and too many comments can often be nearly as bad as, and sometimes worse than, too few.

2. At the beginning of a file, always have a comment containing the name of the file, and a comment containing the purpose of the code in that file. These two comments can often be one line each. You may also want, or need, additional comments in this location, such as the code author's name, the date of the current version, the current version number, and perhaps even a list of modifications included in the current version. Exactly what these initial comments are to contain will generally be determined by your instructor, or the coding conventions of your employer.

3. For each function, include the following information in the given order (and note that in the case of class files, we require the information to be present in the specification (header) file, but we allow it to be omitted in the implementation file, since the appearance of the same information in two different locations generally leads to a maintenance nightmare):

   o One or more comment statements describing in summary form (or more detailed form if necessary) what the function does.

   o The return value of the function.

- A list of the function's parameters in the same order as they appear in the parameter list, with a description of each.
- The function's pre-conditions and post-conditions.
- A list of any exceptions the function may throw, with a description of each.

In the actual parameter list of any function, always place comments (`//in`, `//out`, or `//inout`) to indicate the conceptual nature of each parameter with respect to the direction of information flow. For class member functions, these comments must appear in both the class header file and the class implementation file.

4. Be sure to follow whatever rules may be given for inserting the necessary comments into your source code to identify correctly any submission for evaluation. This may require putting much of your commenting in a form required by some kind of external documentation-generation system, such as Doxygen. If so, the details will be given elsewhere.

**Program Structure**

1. When your entire program is contained in a single source code file, that file should have the structure obtained by placing the following code sections in the given order:

   a. Comments indicating the name of the file, purpose of the code in that file, and anything else that may be required in these initial comments for the particular occasion

   b. The necessary "includes" for the required header files (Also, each group of includes that requires a using declaration or one or more using statements, must be followed by the necessary using declarations/statements.)

   c. Definitions for any global constants and global data types, and declarations of any ***explicitly permitted*** global variables.

   d. Prototypes for any required functions, grouped in some intelligent way that will depend on the nature of the program.

   e. The `main` function.

   f. Definitions for each of the functions whose prototypes appeared before main, and in the same order as the corresponding prototypes.

2. When you have a multi-file program, each file should follow the above guidelines, except of course that only one file will have a `main` function in it, and function prototypes are generally in a different file from the corresponding function

definitions. The contents of any one file should form a logical grouping of some kind within the context of the overall program.

### Maximum Length of Source Code Lines

Choose a maximum length for the lines of text in your source code file in order to ensure that your lines of code do *not* "wrap" in an unsightly fashion, either on the screen or on the printed page. It is your responsibility to choose and insert good line breaks to prevent this from happening, while at the same time ensuring that your code remains as readable as possible. A maximum line length value between 72 and 76 characters is recommended, but whatever value you choose must be stricly less than 80.

### No TAB Characters in Source Code

Your source code must *not* contain any TAB characters. Most modern editors have an easy way to ensure that this requirement is met. You should find out what the necessary procedure is in your case, and use it consistently. Having only spaces in your source code file means that its indentation and aligment will remain the same, no matter where that file may be printed.

### Miscellaneous

1. Put each program statement on a separate line, unless you can make a very good argument to explain why you didn't. One such argument might be that you used a one-line-but-multi-statement C++ "idiom" of some kind.

2. Use named constants whenever appropriate.

3. Always protect header files, and other files to be included, against multiple inclusion.

## Part B: User Interface and Program Output

### Self-identification and programmer identification

Each program should identify both itself and its author(s) when it runs. Exactly how this is to be done for submissions for evaluation will be specified by your instructor.

### Self-description and/or on-line help

Each program should at a minimum describe briefly what it does when it runs. Such information may appear automatically when the program runs, in simple cases. In more complex programs this information and much else may be available via an on-line help system of some kind. Once again, for submissions for evaluation, your instructor will provide details on the kind of information to be displayed.

## Prompting for input

Every program must prompt properly for all input that it requires. Note in particular that there should alwalys be a space at the end of prompt (between the prompt itself and the input entered in response to that prompt) in those situations where the input is entered on the same line as the prompt.

## Echoing input in the output

It is always good programming style to have a program echo its input somewhere in its output, so the user can verify that what was thought to be entered was in fact entered.

## Organization and formatting of output

All output must be organized, aligned, and formatted for maximum readability by the user.

## Spelling and punctuation

The appearance of misspellings and bad punctuation in program output is not only bad style, but is bound to raise questions in the mind of a user about the care taken in other aspects of program development.