# Chapter 1: Preamble

## 1.1 Commandments

A nationally-known expert has said that C++ is a coding monster that forces us to use a disciplined style in order to tame it. This leads to a fundamental rule for skilled C++ programming:

> **Can is not the same as should.**

The style guidelines below are motivated by years of experience writing and debugging C and C++ . None of them are arbitrary. I expect you to read them, understand them, and follow them.

**Commandment 1.** Use C++ , not C for all work in this course. The biggest difference is how one does input and output.

**Commandment 2.** The use of global variables for any purpose will not be tolerated. If your code contains a global variable declaration, your work will be handed back ungraded.

**Commandment 3.** Test every line of code you write. It is your job to prove to me that your entire program works. If I get a program without a test plan and output, I will assume that it does not compile. A program with a partial or inadequate test plan will be assumed to be buggy.

## 1.2 Style

If you want to be a professional, learn to make your work look professional. Start now. Read these guidelines and follow them. Your willingness, diligence, and ability to write clean code in my prescribed style will determine my willingness and ability to write a strong job recommendation for you.

### 1.2.1 General Rules

1. Learn how to include and use a local library, how to create, organize, and compile a multi-module program, and how to use the C preprocessor for conditional compilation.
2. Wherever possible, use **symbolic names** in your code. For example, use quoted characters instead of numeric ASCII codes.
3. Use only the **standard language**. Do not use proprietary extensions and language additions such as `clrscr()` and `conio.h`. Also, do not use system commands such as `pause`. Learn how to do these things within the standard language.
4. **Whitespace**. In general, put spaces around your operators and either before or after each parenthesis. Do not write a whole line of code with no spaces!
5. **Comments**. Use // and keep comments short. Do not repeat the obvious.
6. **Blank lines**. Use blank lines to group your code into "paragraphs" of related statements. Put a blank line after each group. DO NOT put a blank line between every line of code. Do not use more than one blank line. Do not put a blank line before an opening or closing brace. Do not separate the first line of a function from its body.
7. **Simplicity** is good; complexity is bad. If there are two ways to do something, the more straightforward or simpler way is preferred.

8. **Locality** is good; permitting one part of the program to depend on a distant part is bad. For example, initialize variables immediately before the loop that uses them, not somewhere else in the program.

9. Avoid writing **useless code**. For example, don't initialize variables that don't need initialization. Do not leave blocks of code in your program that you have commented out. Remove them before you send me the code.

10. Avoid writing the **same code** twice. For example, don't call strlen(s) twice on the same string. Don't write the same thing in both clauses of an if statement. Don't write two functions that output the same thing to two different streams. This way, when you correct an error once, it is corrected. You don't need to ask how many times the same error occurs.

11. File system **path names** are not portable. If you must write one in your program, put it in a `#define` at the top of main so that it can be found easily and changed easily.

### 1.2.2 Naming

1. Please **do not use** `i`, `I`, `l`, or `O` as a variable name because `i` and `l` look like 1 and `O` looks like 0. Use `j`, `k`, `m`, or `n` instead.

2. **Long, jointed names and short, meaningless names** make code equally hard to read. Try for moderate-length names.

3. **Local variables and class members** should have short names because they are always used in a restricted context. Global objects should have longer names because they can be used from distant parts of the program.

4. Names need to be **different enough** to be easily distinguished. The first part of a name is most important, the last letter next, and the middle is the least important. Any two names in your program should differ at the beginning or the end, not just in the middle.

5. Do not use the same name for two purposes (a class and a variable, for example). Do not use two names that differ only by an '`s`' on the end of one. Do not use two names that differ only by the case (for example `Object` and `object`).

6. I should be **able to pronounce** every variable name in English, and no two names should have the same pronunciation.

7. Learn to use the names of the various **zero-constants** appropriately. *NULL* is a pointer to location 0, `'\0'` is the null character, *""* is the null string, *false* is a bool value, and 0 and 0.0 are numbers. Use the constant that is correct for the current context.

8. Use names written entirely in UPPER CASE for `#defined` **constants** but not for other things. This is a long-honored `C` custom.

9. To be consistent with `Java` usage, the name of a class should start with an Upper case letter and use camel-case after that: `MyClassName`. Variable and function names should always start with a lower case letter: `myVar, myFunction()`.

### 1.2.3 Usage

1. **Use the `C++` language as it is intended**. Learn to use ++ (instead of +1) and if. . . break instead of complex loop structures with flags.

2. Please do not misuse the **conditional operator** to convert a *true* or *false* value to a 1 or a 0:

```
write this    (a < b)
instead of    (a < b) ?  1 :  0
```

3. Use the `->` **operator** when you are using a pointer to a struct. Example:

   ```
   write this    p->next->data
   instead of    *((*p).next).data
   ```

   Using * and . for this purpose leads to code that fails the simplicity test: it is hard to write, hard to debug, and hard to read.

4. Use pointers for sequential **array access** and subscripts for random access.

5. C has **subscripts**. . . use them. They are easy to write, easy to debug, and easy to read. Use pointer arithmetic rarely, if at all. Example: suppose that `ar` is an array of MAX ints and `p` is a pointer to an array of ints.

   ```
   write this                                 ar[n]    or    p[n]
   instead of                                 *(ar+n)  or    *(p+n)
   sometimes you should write this, however   int* ar_end = ar + MAX;
   ```

   The last line in this example is the ordinary way to set a pointer to the end of the array

### 1.2.4 Indentation

1. Never let **comments** interrupt the flow of the code-indentation.

2. Break up a long line of code into parts; **do not let it wrap around** to the left edge of the paper on your listings.

3. The **indentation style** preferred by experts is this:

   ```
   while ( k < num_items) {
       cout << age[k];
       k++;
   }
   ```

   This style is space-efficient and helps avoid certain kinds of common errors. Note that the lines within the loop are **indented a modest amount**: more than 1 or 2 spaces and less than 8 spaces. The opening bracket is on the end of the line that starts with the keyword and the closing bracket is directly below the first letter of the keyword. Eclipse/CDT uses this style by default.

4. Put the indentation into the file when you type the code originally. If you indent one line, most text editors will indent the next line similarly.

5. The other two generally approved styles are shown below. Randomly indented code is unprofessional and inappropriate for a graduate student. For CPSC 427, please use the style in paragraph 3 above.

**Brackets aligned on the left:**

```
while ( k < num_items)
{
    cout << age[k];
    k++;
}
```

**Brackets aligned with the indented code:**

```
while ( k < num_items)
    {
    cout << age[k];
    k++;
    }
```

### 1.2.5 Function Definitions

1. If a function is simple and fits entirely on **one line**, write it that way. Example:

   ```
   bool square_sum( double x, double y ) { return x*x + y*y; }
   ```

2. Otherwise, each function should **start with a whole-line comment** that forms a visual divider. If the function is nontrivial, a comment describing its purpose is often helpful. If there are preconditions for the function, state them here.

```
// ------------------------------------------------------------------
// If needed, put description of function here. ----------------------
void
print( Stack* St )             // Print contents of stack, formatted.
{
    char* p = St->s;           // Scanner and end pointer for data.
    char* pend = p + St->top;

    printf( "The stack %s contains: -[", St->name );
    for ( ;  p < pend;  ++p)  printf( " %c", *p );
    printf( " ]>" );
}
```

3. Write the **return type** on a line by itself immediately below the comment block then write the name of the function at the beginning of the next line, as in the above sample. Why? As we progress through this language, return types become more and more complex. It is greatly helpful to be able to find the function name easily.

4. In general, try to stick to **one return statement per function**. The only exception is when a second return statement substantially simplifies the code.

5. In a well-designed program **all function definitions are short**. Brief comments at the top of the function are often enough to explain the operation of the code. If the function code has identifiable sections or phases, there is a good chance that each phase should be broken out into separate functions. When a function cannot be broken up, each phase should be introduced by a whole-line comment.

### 1.2.6 Types, type definitions and struct

1. As this course progresses, a clear concept of the type of things is going to become essential. Regardless of examples you see in other contexts, when writing the name of a pointer or reference type, write **the asterisk or ampersand as part of the type name**, not as part of the object name. Example:

```
cell* make_cslist( cell& item );    // write it this way
cell *make_cslist( cell &item );    // not this way.
```

2. Use an *enum* **declaration** to give symbolic names to error codes. The name of the error then becomes adequate documentation for the line that uses it. Suppose you want to define codes for various different kinds of errors. Write it this way in C++ :

```
enum error_type { size_OK, Too_small, Too_large };
```

Declare and initialize an error_type variable like this:

```
error_type ercode = size_OK;
```

3. When using **structured types**, use a class declaration (in C++ ). Please DO NOT use the `struct` declaration or the `struct` keyword this term.

### 1.2.7 Using the Tools Library

All programs for this course must be done as projects and all must use the `banner()` and `bye()` functions and the `Fatal()` exception class from the tools library.

1. Establish a directory named cs427 on your hard disk for all the work in this course.
2. From the course website, please download a small library called "tools". It has two files: a source code file `tools.cpp` and a header file `tools.hpp`. On some browsers, you must download the code by using your mouse to copy the code and paste it into a new, empty, file. (Otherwise, the browser may insert HTML tags into the code file.) The ideal place to put your two tools files is at the top level of your cs427 directory. Alternatively, you can copy them directly from the Zoo directory `/c/cs427/code/tools/`.
3. Within the cs427 directory, create a separate subdirectory for each programming assignment plus a subdirectory called "`submit`". Put your source code files, input file, and output files into this directory. If you are using an IDE (integrated development environment) such as Eclipse or Xcode, create a project file using the IDE's menu system, and copy the tools files into the project.
4. Before submitting, copy the files to be submitted into `submit`, make sure they meet the requirements of the assignment, and submit them from there according to the submission procedures for the course.
5. To use the tools, put `#include "tools.hpp"` in your main program and in any `.hpp` files you write yourself. Do not include `tools.cpp` anywhere but be sure to copy it into your project so that it will get compiled and linked in with your code.
6. Various useful functions and macros are included in the tools library; please look at tools.hpp and learn about them. You will need to use *banner()*, *bye()*, *Fatal()*, *flush()*, *DUMPp*, and *DUMPv*. You may also need *say()*, *today()*, *oclock()*,
7. Additional functions may be added during the term, so please check before each new assignment to see if the master copy has changed.
8. Look at the first several lines of `tools.hpp`; note that it includes all the useful C library headers and several of the C++ headers. When you include the tools, you do not need to include these standard header files.
9. Personalization. Before you can use the tools, you must put your own name in the `#define NAME` line of the file "tools.hpp" in place of the dummy name "Ima Goetting Closeur".
10. Start each program with a call on *banner()* or *fbanner()*. This will label the top of your output with your name, etc. End each program with a call on *bye()*, which prints a "Normal termination" message.
11. If you need to abort execution, use *throw Fatal(format, ...)*. This throws a *Fatal* exception with embedded error message created from the arguments to *Fatal()*. The embedded error message can be retrieved in the `catch` block using the member function `what()`. Do not use *assert()* because it does not give adequate user feedback. The syntax for *Fatal()* is exactly like C's *printf()*. Example:

```
f_in = fopen( "data.in", "r" );
if (f_in == NULL) throw Fatal( "Input file cannot be opened." );
```