PPP Style Guide

Bjarne Stroustrup

Introduction
All major real-world software projects have a “house style” (“coding guide-lines”, “project standard” or whatever they are called) to ensure clarity and consistency. This can be a nuisance because no style is perfect, different projects have different styles, and we’d rather just write what “looks good on the day.” However, professionals follow the approved style for a project (e.g. see §25.6). This document outlines the style we use with Programming: Principles and Practice using C++. More examples can be found in the model solutions to exercises posted on the web (though the comments on those can be a bit chatty).

Stick to the simple rules outlined here. It’ll help you to read your own code and that of others and keep you out of some minor programming problems. Your TAs are encouraged to take off points for egregious departures from this style guide (as well as to gently guide you to improve your style – you won’t master all immediately).

“Code as if you really mean it.” Most real-world code “lives” for a long time (years or decades) and is read and modified repeatedly by others. Make their job more manageable by using good style. Remember, one of those “other people” will be you.

Naming
See §3.7 for a discussion. Use a single capital letter to start a type name, e.g. Table and Temperature. Names of non-types are not capitalized, e.g. x and var. We use underscores for multi-part names, e.g. initial_value, symbol_tbl. Use meaningful names. Don’t overuse acronyms. Don’t use excessively long names, such as remaining_free_slots_in_symbol_table. Be careful when using letters and digits that are easily misread: 0Oo1I. Don’t use ALL_CAPS.

C++ identifiers are cases sensitive, so Val is different from val.

Indentation
Indent as done in the book. For example:

```cpp
// if statement:
if (a==b) {
    // ...
}
else {
    // ...
}
```

~ 1 ~
// loop:
for (int i=0; i<v.size(); ++i) {
    // ...
}

// Switch statement:
switch(a) {
case a:
    // ...
    break;
case b:
    // ...
    break;
default:
    // ...
}

// function:
double sqrt(double d)
{
    // ...
}

// class or struct:
class Temperature_reading {
    public:
        // ...
    private:
        // ...
};

Note the placement of the braces ({ and }). We consider that placement is significant. I use tab characters for indentation. This can be a problem when changing editors. As long as you are consistent, you could use spaces for indentation instead (with a minimum of 3 spaces per indentation).

That style is known as “K&R Style” or “Kernighan and Ritchie style” after the people who popularized it for C and even “Stroustrup style” in the context of C++. It preserves vertical space.

Comments
See §7.6.4. Use comments to explain what you cannot state directly in code. Comments are for you and your friends. The compiler doesn’t understand them.

Do not imitate the comment in the book explaining what a language feature does – by the time you use a feature you are supposed to know that.
Don’t say in comments what can be said clearly in code. Code is good at saying exactly what is done (in minute detail and even if it wasn’t what you wanted it to do). Comments are good for

1. Stating intent (what is this code supposed to do)
2. Strategy (the general idea of this is …)

If the comments and the code disagree, both are most likely wrong.

You are encouraged to use intelligible English in your comments; not (say) SMS-lingo. Keep an eye on your grammar, spelling, punctuation, and capitalization. Our aim is professionalism, not “cool.”

Start every program file (.h or .cpp) with a comment containing your name, the date, and what the program is supposed to do. For example:

```cpp
/*
   Joe Q. Programmer
   Spring Semester 2010 (Jan 31)
   Solution for exercise 6.5.
   I use the technique from PPP section XX.Y.ZZ
*/
```

For each non-trivial function and for each non-trivial piece of code, write a comment saying what it is supposed to do:

```cpp
// Bjarne Stroustrup 1/15/2010
// Chapter 5 Exercise 11

/*
   Write out Fibonacci numbers.
   Find the largest Fibonacci number that fits in an int
*/

#include "std_lib_facilities.h"

void fib()
{ /*
     Compute the series and note when the int overflows;
     the previous value was the largest that fit
    */
   int n = 1; // element n
   int m = 2; // element n+1

   while (n<m) {
      cout << n << "\n";
      int x = n+m;
      n = m; // drop the lowest number
    }
}
```
m = x;     // add a new highest number
}

cout << "the largest Fibonacci number that fits in an int is " << n << '\n';
}

In these comments, I assume that the reader knows what a Fibonacci sequence is. If not, a look at the reference to the book will tell. Comments should not substitute for reference material.

**Declarations**

Use one line per declaration. In most cases, add a comment saying what that variable is supposed to do:

```cpp
int p, q, r, b;     // No! Also: not very mnemonic names, and where are the initializers?
const int max = v.size()/2;     // maximum partition size
int nmonths = 0;     // number of months before current date
```

Note that function arguments are usually on a single line (if you need multiple lines, you probably have too complicated functions:

```cpp
int find_index(const string& s, char c);     // find c’s position in s (-1 means ‘not found’)
```

**Variables and constants**

Always initialize your variables. Don’t declare a variable or constant before you have an appropriate value with which to initialize it. For example:

```cpp
vector<int> make_random_numbers(int n)
    // make n uniformly distributed random numbers
{
    if (n<0) error("make_random_number: bad size");
    vector<int> res(n);
    // ...
    return res;
}
```

The point is that we don’t want to initialize `res` until we have checked to see that its initializer `n` is acceptable for our use. So, don’t use this alternative:

```cpp
vector<int> make_random_numbers(int n)
    // make n uniformly distributed random numbers
{
    vector<int> res;     // why define res when you don’t yet have a size for it?
    if (n<0) error("make_random_number: bad size");
    res.resize(n);
    // ...
    return res;
}
The latter is more work and in real code putting a distance between a variable definition and its proper initialization is an opportunity for errors to creep in.

We accept one common and important (apparent) exception to the “always initialize” rule: A variable that is immediately used as the target for an input operation need not be explicitly initialized. For example:

```cpp
vector<string> vec;
string buf;
while (cin>>buf) vec.push_back(buf);
```

Don’t use “magic constants”:

```cpp
for (int i=1; i<32; ++i) {
    // process a month
}
```

Why 32? The size of a `vector`? The number of days in a month plus one? Better:

```cpp
const int mmax = 32;  // here we explain what 32 is and why
// ...
for (int i=1; i<mmax; ++i) {
    // process a month
}
```

```cpp
for (int i=1; i<months.size(); ++i) {  // oh! Now it’s obvious
    // process a month
}
```

**Expressions and operators**

Avoid overly long and complicated expressions: If you use more than three to four operators on the right-hand side of an assignment, consider if what you are saying is clear. Don’t try to be clever with notation. Prefer prefix `++` (e.g. `++count`) to postfix `++` (e.g. `count++`) and prefer either to the less concise `count=count+1`. If in doubt parenthesize, but you are supposed to know the most basic precedence rules: `a*b+c/d` means `(a*b)+(c/d)` and `i<0 || max<i` means `(i<0) || (max<i)`. Don’t “hide” assignments in the middle of expressions: `z=a+(b=f(x))*c`.

**Feature use**

Use the set of features presented in the course so far. Every exercise in a chapter is meant to be solvable using the language and library facilities presented in that chapter or before. You do not get extra credit for using advanced features not yet presented in the course. On the contrary, you may be asked to re-do the work or get points taken off. In particular, note that use of arrays (rather than standard-library `vector`s and `string`s) is not acceptable before Chapter 17. Similarly, don’t use pointers before the book does. Array and pointer use (especially clever uses) correlate strongly with bugs and long painful debugging sessions
– do yourself a favor and avoid that. Don’t use casts (explicit type conversion) until the book introduces it, and then only in emergencies.

You may very well be used to something different, but you are not here to learn “the old way” you are here to learn something new.

**Line length**
Understand that you read lines (usually on the screen). Lay out your code so that it fits into a reasonably-sized window (e.g. 100 characters wide) or on paper (about 80 characters wide). Don’t rely on automatic line wrap: Decide how you want your code to look and do it. For example:

```
// bad:
cout << iten_name << "": unit = " << unit_count << "; number of units = " << number_of_units << "; total = " << unit_count*number_of_units << ‘\n’;
```

```
// better:
cout << iten_name
<< "": unit = " << unit_count
<< "; number of units = " << number_of_units
<< "; total = " << unit_count*number_of_units
<< ‘\n’;
```

You read code more often than you write it.

**Error handling and reporting**
See §5.1. Unless we specifically say otherwise, we will assume that your program

1. should produce the desired results for all legal inputs
2. should give reasonable error messages for all illegal inputs
3. need not worry about misbehaving hardware
4. need not worry about misbehaving system software
5. is allowed to terminate after finding an error

Unless otherwise specified, your program is not required to recover from errors. If your program detects an error, such as an illegal input, it may exit by a call of `error()` (see §5.6.3) from `std_lib_facilities.h` (remember to catch `runtime_error` to write out the error message).

**Compiler errors and warnings**
A program should compile cleanly; that is, it should compile without errors (or it won’t run) and warnings (most warnings point to a potential problem). We accept one exception; you may ignore the `signed/unsigned` warning for this:
for (int i=0; i<v.size(); ++i) // …

where v is a standard library container (such as, vector or string). On a normal PC, this warns of something that can only happen if a vector has more than $2^{31}$ (more than 2,000,000,000) elements. For an explanation of this problem and what to do if you need larger vectors, see §25.5.3.

**Foreign style**

When you learn a new language, it is almost impossible to avoid using idioms and style from your previous language. This is true for programming languages as well as natural languages. However, the ultimate aim is always to master the idioms of the new language: “to speak it as a native.” For programming languages, that’s important for maintenance (other programmers do better with styles they are familiar with) and can be important for conciseness of expression, correctness, and performance. Style matters!

**For Java programmers**

Java and C++ code can be very similar – especially where computations are done mostly with built-in types, such as char, int, and double. Try to remember that in C++ not everything is or should be a class. Free-standing functions are fine (§9.7.5). Not every member function (“method”) should be virtual and by default they are not. There is no universal base class (Object) in C++. Parameterization of classes and functions is central to C++ (think “generics on steroids;” Chapters 19-21). Namespaces (§8.7) can take the role of highest level classes (use a namespace where you might have used a class with only static members). Use const to indicate immutability (§4.3.1 §9.7.4). Instead of serialization, look to iostreams (Chapters 10-11).

**For C programmers**

Almost every C construct is also C++ and has the same meaning in C++. C lacks direct support for user-defined types, such as string and vector, so their equivalents must be simulated using lower-level facilities. Similarly, the creation of an object using new must be synthesized by lower-level facilities (new provides free-store allocation and initialization). For a C programmer learning C++, the aim must be to avoid those lower-level facilities where reasonable. Chapter 27 explains how in detail. Avoid malloc(), casts, pointers, and arrays at least until Chapters 17-18 where their proper role in C++ is outlined and don’t use macro substitution (§27.8, §A.17). Don’t simply use new where you would have used malloc(); think about what you are trying to achieve. Typically, letting sting, vector, or some other container (Chapter 20) do the work is simpler, safer, and equally efficient.

**Feedback**

Comments on this document and suggested improvements are welcome. Send email to bs@cs.tamu.edu.