PPP Style Guide

Bjarne Stroustrup
www.stroustrup.com/programming
www.stroustrup.com

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. This document outlines the style we use with *Programming: Principles and Practice using C++* (sometimes referred to as "PPP").

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" might be you.

There can be no one true style that is best for everybody everywhere, but what we recommend here is better than anything a novice can cook up in a hurry. This is not a comprehensive style guide for major industrial use, but it is still better than some we have seen claimed to be that.

Industrial-strength coding guidelines can become quite detailed and cover topics that a novice don't encounter. Don't look at them just yet, but if/when you need a set of guidelines that's helpful "at scale", I recommend The C++ Core Guidelines.

Naming

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** and **symbol_tbl**. Use meaningful names. Don't overuse acronyms. Don't use excessively long names, such as **remaining_free_slots_in_symbol_table**. The length of a name should be roughly proportional to the size of its scope.

Be careful when using letters and digits that are easily misread: **0Oo1lL**. Don't use **ALL_CAPS**.

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

Indentation

Indent as done in the book. For example:

```
if (a==b) {         // if statement
        // …
}
else {
        // …
}

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

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

double sqrt(double d)    // function
{
        // …
}

class Temperature_reading  {    // class or struct
public:
        // …
private:
        // …
};
```

Note the placement of the braces (**{** and **}**). We consider that placement 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. The point about preserving vertical space is to fit logical entities (e.g., a function definition) on a single screen to ease comprehension.

Whitespace

We don't have really strong opinions on the use of whitespace beyond the use of indentation, but we have found that the following rules of thumb ease reading. We use vertical whitespace (empty lines) between functions, between classes, and to separate logically different sections of declarations of code. For example:

```
void fct1()
{
        vector<string> v;               // to be used in the whole of the function

        string s;                       // input
        while (cin>>s)
                v.push_back(s);

        for (string& s : v) {           // processing
                // …
        }
}

int fct2()
{
        // something

        // something else
}

class X {
        // …
};
```

The book is full of examples. Sometimes, we use two blank lines between two functions, but don't overdo vertical whitespace: doing so will limit what you can fit on a screen for simultaneous viewing.

Never place two or more statements or two or more declarations on a single line:

```
int x = 7; f(x);  ++x;      // don't
```

It is too easy to miss something important in such dense text.

To ease reading, we place a control statement and a controlled statement on separate lines.

```
        while (cin>>s)
                v.push_back(s);
```

We use a space after a **for**, **switch**, **if**, or **while** before the **(**.

We do *not* use a space between the function name and the **(** in a declaration or a call.

We use a space between class name and the **{** in a class declaration.

We don't usually insert spaces in expressions, but when we do it's to emphasize meaning (operator binding):

```
if (x<0 || max<=x)
        // …
cin>>s;
int a = z+y*z;
```

We don't use spaces in function argument lists, but we do use them in lists of argument types:

```
void f(int,  const string&,  double);
f(1,"2",3.4);
```

In the ever-popular discussion of where to put spaces near the "pointer to" declarator operator, *****, we use the conventional C++ style:

```
int* p;          // do it this way
int  *p;         // don't
int  * p;        // don't
int*p;           // don't
```

And when you are defining a variable, remember to initialize:

```
const auto perfect = 6;
```

If you use a "smart" editor, it will have its own style, which you may or may not be able to influence.

Comments

Use comments to explain what you cannot state directly in code. Comments are for you and your friends. Compilers don't understand them.

Do not imitate comments in the book that explain 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 …)
3. Stating invariants, pre- and post-conditions

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, .cpp**, or **module**) handed in during a course with a comment containing your name, the date, and what the program is supposed to do. For example:

```
/*
        Joe Q. Programmer
        Spring Semester 2011 (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:

```
// Bjarne Stroustrup 1/15/2010
// Chapter 4 Exercise 11

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

#include "PPP.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, we assume that the reader knows what a Fibonacci sequence is. If not, a look at the reference to the book (or a web search) will tell. Comments should not substitute for reference material. If an extensive discussion is needed a comment can instead refer to reference material.

A comment stating a pre-condition, post-condition, or an invariant is not a substitute for code appropriately checking a condition (e.g. argument validation in functions and constructors). For example:

```
class vector {    // vector of double
        /*
        invariant:
                for 0<=n<sz elem[n] is element n
                sz<=space;
                if sz<space there is space for (space-sz) elements after elem[sz-1]
        */
        int sz;                 // number of elements
        double* elem;           // pointer to the elements (or 0)
        int space;              // number of element plus number of free slots
        // …
};

vector::vector(int s)
        :sz(s), elem(new double[s]), space(s)
{
        If (s<0)                // size must be non-negative
                throw Bad_vector_size();
        for (int i=0; i<sz; ++i)
                elem[i]=0;      // elements are initialized
}
```

Or use **expect()** to express assertions.

Declarations

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

```
int p, q, r, b;             // No! Also: not very mnemonic names;  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:

```
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:

```
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:

```
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:

```
int x;
cin>>x;
```

Even in this case, we often use an initializer so that **x** has a defined value if the input operation fails. Note that objects of some types, such as **string** and **vector** are implicitly initialized. This example has no uninitialized variables:

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

Don't use "magic constants'':

```
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:

```
const int mmax = 32;            // here we explain what 32 is and why

// …

for (int i=1; i<mmax; ++i) {
        // … process a month …
```

Better still:

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

And better still when you traverse a whole container:

```
for (in x : months) {       // Even more obvious
        // … process a month …
}
```

Use **const** if you don't plan to ever change the value of an object.

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**.

Language feature use

When doing a drill or an exercise, 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. Also, note that use of built-in 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) only in emergencies, and there are very few emergencies.

Don't use macros.

Avoid global variables.

Avoid naked **delete**s and naked **new**s. The implication of following this advice is to keep resources owned by objects that handle their release implicitly. Following this advice – and extending it to resource management in general gets you most of the way to exception-safe and leak-free code.

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 << item_name << ": unit = " << unit_count << "; number of units =  " << number_of_units
<< "; total = " << unit_count*number_of_units <<  '\n';

// better:
cout << item_name
        << ": unit = " << unit_count
        << "; number of units =  " << number_of_units
        << "; total = " << unit_count*number_of_units
        <<  '\n';
```

You read a piece of code much more often than you write it. Layout should reflect and emphasize the logical structure of code.

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).

Error handling and reporting

See Chapter 4. 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()** from **PPP.h** (remember to catch **runtime_error** to write out the error message).

Return an error indicator for errors that are not exceptional, and an immediate caller can be expected to handle. For example:

```
ifstream is {ifile};
if (!is)
        error("couldn't open",ifile);
```

Throw an exception if an error is exceptional (shouldn't happen in a normal execution of a program) or an immediate caller cannot be expected to handle it. An example would be an out-of-range error.

Do not try to handle exceptions everywhere. Rely on RAII to allow exceptions to percolate up to a caller than is prepared to deal with errors. If nothing else, let **main()** give a reasonable error message. Having many **try**-**catch** clauses scattered all over a program is a sign of poor design.

Use some form of invariant/contract to express expectations of arguments and state. For example, use **expect()** to check preconditions.

Generic programming

When writing a template, specify concepts to express the requirements on template parameters. If you can't immediately think of a suitable concept, use a placeholder:

```
template<typename T> concept Element = true;  // to be elaborated later

template<Element T> class My_container {
        // …
};
```

Foreign style

When you learn a second 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 (and similar languages) 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. Non-member functions are fine. 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;"). Namespaces 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. Instead of serialization, look to iostreams. Don't use **new** to simulate local variables:

```
void f1(int i)      // awkward, "alien" style
{
        vector<int>& v = new vector<int>;
        // …
        delete &r;
}
```

This is more verbose, more error-prone, and less efficient that the colloquial:

```
void f2(int i)
{
        vector<int> v;
        // …
}
```

When returning a collection of data from a function, return a container, not a pointer to a container:

```
vector<int> * f3(int i)    // awkward, "alien" style
{
        vector<int>* v = new vector<int>;
        // …
        return r;
}
```

Instead, return a container:

```
vector<int> f4(int i)      // returning a container
{
        vector<int> v;
        // …
        return v;
}
```

That's simpler code. Optimizations or move semantics will make returning a container at least as efficient as the pointer style, and you can't forget a **delete.**

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. PPP2 Chapter 27 – available on the PPP website – explains how in detail. Avoid **malloc()**, casts, pointers, and arrays at least until Chapters 15-16 where their proper role in C++ is outlined and don't use macro. Avoid C-style string manipulation (e.g. **strcmp**(), **strcpy()**, and especially **gets()**). Don't simply use **new** where you would have used **malloc()**; think about what you are trying to achieve. Typically, letting **string**, **vector**, or some other container, do the work is simpler, safer, and equally efficient.

Feedback

Comments on this document and suggested improvements are welcome.