Autumn 2003 Shensheng Zhang

C Programming Style for CS 111: Programming Methodology

From material by Robert Plummer, Julie Zelenski, and Harry Lai

As we will stress all the time, a working program is only half the challenge—constructing an elegant and well-engineered solution is your ultimate goal. Developing a good sense of style takes practice and work, conveniently we've got plenty of both coming up in this course. Your section leader's feedback on your code will be invaluable in helping you learn how to design and organize code in a clear, readable manner that works for you. As with any complex activity, there is no one "right" way that can easily be described, and there is no easy-to-follow checklist of do's and don'ts. But that said, this handout will try to identify some qualities that can contribute to readable, maintainable programs to give you an idea what we're working toward.

Why are we concerned about programming style?

When your program is digested by a C compiler, style makes little difference. The comments, the well-chosen names, and the careful alignment of the code have no effect on the ultimate execution of the program. So why bother?

The effort you put into achieving good style in your program is clearly for human consumption, and the most important person who is going to read your code is you. A program written with good style is more likely to be correct than one without, and it will be easier for you to debug and modify as you move it to its final form.

That last point is worth noting. When you develop software in the "real world", and even for the assignments in the remainder of this course, it is rarely the case that you write the entire program in one attempt. It is far more likely that you will get the essential "core" of the program running, and then add the remaining features. Large problems are simply too complex to tackle any other way.

The result of this approach is that you will spend some time looking at your own code, asking questions like "What was I trying to do at this particular point," or "Where is the best place to add a new feature?" It is at this point that you will realize the value of an investment in good code.

Here is a simple example. Which of the following lines of code would you rather work on?

```
m = s * h;

distance = rate * time;
```

The first line gives no hint as to its meaning. If you are lucky, you remember what quantities the variables hold. If you don't, or if someone else wrote the code, you will have to spend some time figuring out what is going on. The second line holds no such mysteries.

Now consider the following attempt to determine a commission based on sales:

```
if (salesAmt <= 50.0) commission = 0.0;
if (salesAmt > 50.0) if (salesAmt <= 100.0) commission = 0.02 * salesAmt;
if (salesAmt > 100.0) commission = 0.03 * salesAmt;
```

Though it produces the correct answer, the code above is not nearly as easy to understand or modify as the following equivalent version:

```
if (salesAmt <= 50.0)
{
        commission = 0.0;
}       else if (salesAmt <= 100.0)
{
        commission = 0.02 * salesAmt;
}       else
{
        commission = 0.03 * salesAmt;
};
```

These two examples should make the point. Writing with clarity is as important for software as it is for literature.

Choosing Names

The first step in documenting code is choosing meaningful names for things. For variables and constants the question is "What is it?" For functions, the question is "What does it do?" A well-named variable or function helps document all the code where it appears. Here are a few examples:

Bad Names

sum
L P
b

Good Names

RunningTotal
lastPayment
balance

Good Names in Context

x
y
i
n

The bad names do not convey enough information, while the good ones explain what they are in terms that relate to the problem. If you are doing graphics or geometry, x and y are good names; if you are doing a payroll, they are probably not. You can sometimes use i as a generic loop index variable, and n as a generic counter variable, but be careful and don’t overuse them.

You should not have much difficulty deciding whether your names contribute to readability. Here are a few guidelines:

Name Guidelines

- Use lower/upper case	lastPayment
- Use nouns for variables	diameter
- Use verbs for functions	PrintPayroll
- Use uppercase for constants	#define MAX_VALUE 1000
- Use moderate length	
numberOfPeopleOnOlympicTeam	too long
ntm	too short
numTeamMembers	about right

- The name should suggest the concept

The last point refers to the situation where we have a concept in mind, and ask ourselves what variable name it suggests. That approach is actually backwards. If other people read your program, they don't know your concept – all they have is the variable name. So it is the name that should suggest the concept, not the other way around.

Avoid content-less, terse, or cryptically abbreviated variable names. Names like "a", "temp", "nh" may be quick to type, but they're awful to read. Choose clear, descriptive names: "average", "height", or "numHospitals". In general, names should mean something. If a variable contains a list of floats that represent the heights of all the students, don't call it list, and don't call it floats, call it heights. There are a few variable naming idioms that are so prevalent among programmers that they form their own exception class:

i, j, k	Integer loop counters.
n, len, length	Integer number of elements in some sort of aggregation
x, y	Cartesian coordinates. May be integer or real.

The uses of the above are so common, that the apparent lack of content is acceptable.

Names of #define-d constants should make it readily apparent how the constant will be used. "MAX_NUMBER" is a rather vague name: maximum number of what? "MAX_NUM_STUDENTS" is better, because it gives more information about how the constant is used.

Function names should clearly describe their behavior. Functions which perform actions are best identified by verbs, e.g. "FindSmallest()" or "DrawTriangle()". Predicate functions and functions which return information about a property of an object should be named accordingly: e.g. "IsPrime()", "StringLength()", "AtEndOfLine()".

Making your code self-documenting

The following line of code adds one to the variable days in certain circumstances:

```
if (((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0))
{
        days++;
};
```

Unless you are familiar with the formula on the right, figuring out the intent might take some effort. Now consider the following alternative:

```
leapYearResult = ((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0);
if (leapYearResult )
{
        days++;
};
```

Big difference! By adding an extra variable, the programmer has made the computation completely clear[1]. No comments are required; no reference manual is needed. The code provides its own documentation by way of the variable names. Of course, the formula still has to be correct. Just saying it checks for leap year doesn't mean that it does.

Using comments effectively

The motivation for commenting comes from the fact that a program is read many more times that it is written. A program must strive to be readable, and not just to the programmer who wrote it. A program expresses an algorithm to the computer. A program is clear or "readable" if it also does a good job of communicating the algorithm to a human. Given that C is a rather cryptic means of communication, an English description is often needed to understand what a program is trying to accomplish or how it was designed. Comments can provide information that is difficult or impossible to get from reading the code. Some examples of information you might find in comments:

- General overview. What are the goals and requirements of this program? This function?
- Design decisions. Why was a particular data structure or algorithm chosen? Were other strategies tried and rejected?
- Error handling. How are error conditions handled? What assumptions are made? What happens if those assumptions are violated?
- Nitty-gritty code details. Comments are invaluable for explaining the inner workings of particularly complicated (often labeled "clever") paths of the code.
- Planning for future. How might one make modifications or extensions later?
- And much more... (This list is by no means exhaustive)

At the top of each file, it is a good convention to begin with an overview comment for the program, interface, or implementation contained in the file. The overview is the single most important comment in a program. It's the first thing that anyone reading your code

---

[1] Another equally valid design is to make a separate function called IsLeapYear. It would take in a year (as an integer) and return TRUE or FALSE.

will read. The overview comment explains, in general terms, what strategy the program uses to produce its output. The program header should lay out a roadmap of how the algorithm works— pointing out the important routines. The overview should mention the role of any other files or modules that the program depends on. Essentially, the overview contains all the information which is not specific or low-level enough to be in a function comment, but which is helpful for understanding the program as a whole.

Each noteworthy function is often preceded by a comment that contains the function's purpose, a description of the parameters, and details of the function's return value, if any. It's a good idea to mention if the function relies directly on any #define-d constants. Additionally, you should describe any special cases or error conditions the function handles (e.g. "...prints out an error message if divisor is 0", or "...returns the constant NOT_FOUND if the word doesn't exist").

Comments should correctly match the code; it's particularly unhelpful if the comment says one thing but the code does another thing. One adage to keep in mind is that *"Misleading comments are worse than no comments."* It's easy for such inconsistencies to creep in over the course of developing and changing a function. Be careful to give your comments at once-over at the end to make sure they are still accurate.

Inside your functions, you will not need too many comments. Look at it this way. If your variable and function names are descriptive, if your code is well laid out, and if you have avoided obscure, "tricky" code, what else is there to say? The best program is one that is so clear from the outset that few comments are actually needed.

The audience for all commenting is a C-literate programmer. Therefore you should not explain the workings of C or basic programming techniques. Useless over-commenting can actually decrease the readability of your code, by creating muck for a reader to wade through. For example, the comments

```
int counter; /* declare a counter variable */
i = i + 1; /* add 1 to i */
while (index < length)... /* while index is less than length */
num = num + 3 - (num % 3); /* add 3 to num and subtract num mod 3 */
```

do not give any additional information that is not apparent in the code. Save your breath for important higher-level comments! Only illuminate low-level details of your implementation where the code is complex or unusual enough to warrant such explanation. A good rule of thumb is: *explain what the code accomplishes rather than repeat what the code says*. If what the code accomplishes is obvious, then don't bother.

Here are a few commenting guidelines:

- Comments should make the code accessible to the reader
- Explain the code's intent in the heading
- Keep the comments up to date (if you update the code, update the comment)
- Don't comment bad code--fix it
- Avoid useless comments

Attributions

All code copied from books, handouts or other sources, and any assistance received from other students, section leaders, fairy godmothers, etc. must be cited. We consider this an important tenet of academic integrity. For example,

```
/* IsLeapYear is adapted from Eric Roberts text,
* _The Art and Science of C_, p. 200.
*/
```

or

```
/* I received help designing the Battleship data structure, in
* particular, the idea for storing the ships in alphabetical order,
* from Joe Smith, a section leader, on Tuesday, Nov. 11, 2000.
*/
```

Blank lines

Including blank lines to separate sections of code can improve readability whether there are comments or not. Without even reading the following code, it is easy to see that this block does three jobs:

```
{

    structSize = sizeof(CHOOSE_COLOR);
    hWndOwner = hWnd;
    windowFlags = CC_RGBINIT;

    ChooseColor(newColor);

    if (rgbResult != inputSampleRef)
    {
        inputSampleBrush = CreateSolidBrush(inputSampleRef);
        InvalidateRectangle(hWnd, NULL, TRUE);
    };
}
```

Booleans

Boolean expressions and variables seem to be prone to redundancy and awkwardness. Replace repetitive constructions with the more concise and direct alternatives. A few examples:

```
if (flag == TRUE)          is better written as          if (flag)

if (matches > 0)           Is better written as          found = (matches > 0);
{
    found = TRUE;
}   else
{
    found = FALSE;
};

if (hadError == FALSE)     is better written as           return (!hadError);
{
    return TRUE;
}   else
{
    return FALSE;
}
```

Constants

Avoid embedding magic numbers and string literals into the body of your code. Instead you should #define a symbolic name to represent the value. This improves the readability of the code and provides for localized editing. You only need change the value in one place and all uses will refer to the newly updated value.

#define-d constants should be independent; that is, you should only need to change one #define to change something about a program. For example,

```
#define RECT_WIDTH 3
#define RECT_HEIGHT 2
#define RECT_PERIMETER 10 /* WARNING: problem */
```

is not so hot, because if you wanted to change RECT_WIDTH or RECT_HEIGHT, you would also have to remember to change RECT_PERIMETER. The correct way is:

```
#define RECT_WIDTH 3
#define RECT_HEIGHT 2
#define RECT_PERIMETER (2 * RECT_WIDTH + 2 * RECT_HEIGHT)
```

Parentheses should be placed around any constant definition that involves an expression. This insures that the constant will be evaluated as intended, and that the result will not be altered by operator precedence in the expression in which the constant is used.

Decomposition

Decomposition does not mean taking a completed program and then breaking up large functions into smaller ones merely to appease your section leader. Decomposition is the most valuable tool you have for tackling complex problems. It is much easier to design, implement, and debug small functional units in isolation that to attempt to do so with a much larger chunk of code. **Remember that writing a program first and decomposing after the fact is not only difficult, but prone to producing poor results**. You should decompose the *problem*, and write the program from that already decomposed framework. In other words, you are aiming to decompose problems, not programs!

The decomposition should be logical and readable. A reader shouldn't need to twist her head around to follow how the program works. Sensible breakdown into modular units and good naming conventions are essential. Functions should be short and to the point.

Strive to design functions that are general enough for a variety of situations and achieve specifics through use of parameters. This will help you avoid redundant functions—sometimes the implementation of two or more functions can be sensibly unified into one general function, resulting in less code to develop, comment, maintain, and debug. Avoid repeated code. Even a handful of lines repeated are worth breaking out into a helper function called in both situations.

Formatting and Capitalization

One final point about style is that you should develop a consistent approach. Doing so improves readability and adds information to the code. For example, if you always use all caps and underscores for constants, then when you see a line like this:

```
ChoosePlayers (MAX_TEAM_SIZE);
```

you know right away that MAX_TEAM_SIZE is a constant and that it is defined somewhere in a #define statement. If sometimes your constants use mixed case, and sometimes your variables are all caps, then you can't tell for sure and you've lost one small chance to make your code more understandable.

One convention you will have to decide on is the placement of braces. Your textbook presents the "classic" method for C, and another method (vertically aligning braces with indented code inside) has been shown in class. For the purpose of simple, let’s all keep the form we used in class (vertically aligning braces with indented code inside).

For capitalization schemes, we will typically capitalize each word in the name of a function, variables will be named beginning with abbreviation on the data type, #define constants will be completely uppercased, etc.

下面将详细叙述同学们在作业中要遵循的编程规范及风格：

（1） 程序结构：

① 逻辑上属于同一个层次的语句互相对齐；逻辑上属于内部层次的推到下一个对齐位置。

② 在每个操作符、表达式前后至少留一个空格。

③ 将相关的语句进行分组（grouping），每一组前后留一个空白行。

④ 一行代码中只写一个声明、定义和简单语句。

⑤ 所有的语句（简单和复合）均以‘ ；’结束。

⑥ 所有条件和循环语句都认为是复合语句；所有复合语句的起始标识符‘ { ’均另起一行。

如：

```
if    (condition) // do not place '{' here although it is perfectly OK in many textbooks
{     // using combound statement brackts even it is a simple statement.
      statements;
};    // end with ';' although it is all right without it.
```

（2） 标识符（变量名、函数名、符号名）包括所有的命名：

① 基本原则是一个标识符应能明确表达其含义和用途，可以直接应用于程序的说明文档中（self-documenting），可以让读者方便地理解。

② 除了循环块体的循环变量，如：i, j, k 和一些有明显物理意义的变量，如图形学中坐标点的 x, y, z 外，不使用含义不清的变量。

③ 除另有说明外，作业中不允许定义全局变量。

④ 常量：全部大写；尽量用全名，不用缩写；单词之间用“ _ ”隔开；

⑤ 函数名：每个单词词头大写；尽量用全名，不用缩写；

⑥ 采用 windows 编程统一的命名规范。每个标识符由一个至多个 identifier segment 组成，第一个 segment 是小写的表示该变量的数据类型，其后每个 segment 的首字母大写。当一个 segment 较长时，可以运用适当的简写。一个具有良好风格的简写应该遵循下述两点

- 移走单词中的元音，除非其位于单词的开头；
- 用一个辅音代替两个辅音。

表 1.数据类型及其缩写规则

Data type	Prefix	Example
Boolean	bln	blnFound
Byte	byt	bytRasterData
Date (Time)	dtm	dtmStart
Double	dbl	dblTolerance
Error	err	errOrderNum
Integer	int	intQuantity
Long	lng	lngDistance

String	str	strFName
Point	pnt	pntArray
FilePoint	fpt	fptInFile
Char	chr	chrDummy
User-defined type	udt	udtEmployee

示例：intWidth　　intStudentCount

（3） 函数定义：

① 一个函数内的代码行数应控制在一至两个屏幕之内。一个函数就做一件事情。

② 不使用 goto 语句。不在循环中使用 continue 和 break 语句。

③ 在函数的定义及使用中，需注意下述问题：

- 总是明确地标明返回类型。
- 在每一个函数的开始，放置一个明显的标识（见示例）并加以注释，说明其目的，解释其包含的参数以及返回值。
- 所有的函数都以 return 语句终止，在一个函数中避免使用多个 return 语句。

示例：

```
/* **************< Function Name >******************
      < function purpose >
      Pre <explanation of all parameters>
      Post <explanation of all output and return values >
*/

int function (int p1, float p2 )
{
      // Local Definitions
      // Statements
      return value;
}

/* ************** End of Function ****************** */
```

（4）main 函数中定义的是需要在子函数之间传递的数据，一般讲，在 main 中可以包括以下三类语句：

① 程序开始时的初始化工作

② 函数调用

③ 程序结束时的收尾工作。

示例：

```
int main ( void )
{
      // Local Definitions
      // Start of program
            < function calls only >
      // End of program
      return 0;
}
```