Chapter 1: Style and Program Organization

A program is a detailed set of instructions read by both a human and a machine. The computer reads only the code, while the human concentrates on the comments. Good style pertains to both parts of a program. Well-designed, well-written code not only makes effective use of the computer, it also contains careful constructed comments to help humans understand it. Well-designed, well-written code is a joy to debug, maintain, and enhance.

Good programming style begins with the effective organization of code. using a clear and consistent organization of the components of your program you make them more efficient, readable, and maintainable.

Program Organization

Good computer programs are organized much like good books. This can seen especially well with technical books, in which the structure is very clear.

People have been writing books for hundreds of years, and during that time they have discovered how to organize the material to efficiently present their ideas. Standards have emerged. For example, if I asked you when this book was copyrighted, you would turn to the title page. That's where the copyright notice is always located.

The same goes for code. In fact, we can make the parallels quite explicit.

Any technical book can be analyzed into standard components. So can program. These components correspond quite closely as the following table shows.

Book	Program
Title Page	Heading
Table of Contents	Table of Contents
Chapter	Module
Section	Function
Paragraph	Conceptual Block
Sentence	Statement
Word	Variable
Index	Cross Reference
Glossary	Variable Declaration Comments

These components really do serve the same purposes.

- **Title Page**

  A book's title page contains the name of the book, the author, and the publisher. On the reverse of the title page is the copyright page, where you find things like the printing history and Library of Congress information.

At the beginning of every well-documented program is a section known as the heading. It is, in effect, the title page of the program. The heading consists of a set of boxed comments that include the name of the program, the author, copyright, usage, and other important information. The heading comments are fully discussed in Chapter 2.

- **Table of Contents**

  Every technical book has a table of contents. It lists the location of all the chapters and major headings, and serves as a road map to the rest of the book.

  A program should have a table of contents as well, listing the location of each function. This is difficult and tedious to produce by hand, however it can be produced quite easily by a number of readily available tools, as discussed later in this chapter.

- **Chapters**

  Technical books are divided into chapters, each one covering a single subject. Generally, each chapter in a technical book consists of a chunk of material that a reader can read in one sitting, although this is not a rule: Donald Knuth's highly regarded 624-page *Fundamental Algorithms* (Addison-Wesley, Reading, MA, 1968) contains only two chapters.

  Similarly, a program is divided into modules, each a single file containing a set of functions designed to do some specific job. A short program may consist of just one module, while larger programs can contain 10, 20, or more. Module design is further discussed later in this chapter.

- **Sections**

  Each chapter in a technical book is typically divided into several sections. A section covers a smaller body of information than a chapter. Sections in this book are identified by section heads in bold letters, making it easy for a reader to scan a chapter for a particular subject.

  Just as a book chapter can contain several sections, a program module may contain several functions. A function is a set of instructions designed to perform a single focused task. A function should be short enough that a programer can easily understand the entire function.

- **Index**

  A technical book should have a good index that lists every important subject or keyword in the book and the pages on which it can be found. The index is extremely important in a technical book because it provides quick access to specific information.

  A program of any length should have a cross reference, which lists the program variables and constants, along with the line numbers where they are used. A cross reference serves as an index for the program, aiding the programmer in finding variables and determining what they do. A cross reference can be generated automatically by one of the many cross reference tools, such as *xref*, *cref*, etc.

- **Glossary**

  A glossary is particularly important in a technical book. Each technical profession has its own language, and this is especially true in computer programming (e.g., set COM1 to 1200,8,N, I to avoid PE and FE errors). A reader can turn to the glossary to find the meaning of these special words. Every C program uses its own set of variables, constants, and functions. These names change from program to program, so a glossary is essential. Producing one by hand is impractical, but as you'll see later in this chapter, with a little help from some strategically placed comments, you can easily generate a glossary automatically.

Rule 1-1:

Organize programs for readability, just as you would expect an author to organize a book.

Automatic Generation of Program Documentation

Some of the program components described above can be generated automatically. Consider the table of contents, for example. On UNIX systems, the *ctags* program will create such a table. Also, there is a public domain program, called *cpr*, that does the job for both DOS and UNIX.

A cross reference can also be generated automatically by one of the many cross reference tools, such as *xref*, *cref*, etc. However, you can also generate a cross reference one symbol at a time. Suppose you want to find out where total-count is located. The command grep searches files for a string, so typing:

```
grep -n total_count *.c
```

produces a list of every use of *total_count* in all the C files. (The *-n* tells grep to print line numbers.)

The command *grep* is available both on UNIX systems and in MS-Windows with Borland C++ and Borland's Turbo-C.

Also in UNIX, the command:

```
vi ‘grep -l total_count *.c‘
```

invokes the *vi* editor to list the files that contain the word `total_count`. Then you can use the *vi* search command to locate *total_count* within a file. The commands next (*:next*) and rewind (*:rew*) will flip through the files. See your *vi* and UNIX manuals for more details.

Borland C++ and Borland's Turbo-C++ have a version of *grep* built in to the Integrated Develop Environment (IDE). By using the command Alt-Space you can bring up the tools menu, then select *grep* and give it a command line, and the program will generate a list of references in the message window. The file corresponding to the current message window line is displayed in the edit window. Going up or down in the message changes the edit window. With these commands, you can quickly locate every place a variable is used.

You can also partially automate the process of building a glossary, which is a time-consuming task if performed entirely by hand. The trick is to put a descriptive comment after each variable declaration. That way, when the maintenance programmer wants to know what `total_count` means, all he or she has to do is look up the first time `total_count` in mentioned in the cross reference, locate that line in the program, and read:

```
    int total_count;   /* total number of items in all classes */
```

So we have a variable (`total_count`) and its definition: “Total number of items in all classes” — in other words, a glossary entry.

Module Design

A module is a set of functions that perform related operations. A simple program consists of one file; i.e., one module. More complex programs are built of several modules.

Modules have two parts: the public interface, which gives a user all the information necessary to use the module; and the private section, which actually does the work.

Another analogy to books is helpful here. Consider the documentation for a piece of equipment like a laser printer. This typically consists of two manuals: the Operator's Guide and the Technical Reference Manual.

The Operator's Guide describes how to use the laser printer. It includes information like what the control panel looks like, how to put in paper, and how to change the toner. It does not cover *how* the printer works.

A user does not need to know what goes on under the covers. As long as the printer does its job, it doesn't matter how it does it. When the printer stops working, the operator calls in a technician, who uses the information in the Technical Reference Manual to make repairs. This manual describes how to disassemble the machine, test the internal components, and replace broken parts.

The public interface of a module is like an Operator's Guide. It tells the programmer and the computer how to use the module. The public interface of a module is called the “header file.” It contains data structures, function definitions, and **#define** constants, which are needed by anyone using the module. The header file also contains a set of comments that tells a programmer how to use the module.

The private section, the actual code for the module, resides in the *c* file. A programmer who uses the module never needs to look into this file. Some commercial products even distribute their modules in object form only, so *nobody* can look in the private section.

Rule 1-2:

Divide each module up into a public part (what's needed to use the module) and a private part (what's needed to get the job done). The public part goes into a .h file while the private part goes into a .c file.

Libraries and Other Module Groupings

A library is a collection of generally useful modules combined into a special object file.

Libraries present a special problem: How do you present the public information for a library? Do you use a single header file, multiple header files for the individual modules, or some other method?

There is no one answer. Each method has its advantages and disadvantages.

Multiple header files

Because a library is a collection of modules, you could use a collection of header files to interface with the outside world. The advantage to this is that a program brings in only the function and data definitions it needs, and leaves out what it doesn't use.

The X Windows system, for example, has a different header file for each module (called a tool kit in X-language).

A typical X Windows program contains code that looks like this:

```
#include <Xll/Intrinsic.h>
#include <Xll/Shell.h>
#include <Xm/Xm.h>
#include <Xm/Label.h>
#include <Xm/RowColumn.h>
#include <Xm/PushB.h>
#include <Xm/Separator.h>
#include <Xm/BulletinB.h>
#include <Xm/CascadeB.h>
```

As you can see, this can result in a lot of **#include**s**.** One of the problems with this system is that it is very easy to forget one of the **#include** statements. Also, it is possible to have redundant **#include**s. For example, suppose the header file *XmILabel.h* requires *XmISeparator.h* and contains an internal **#include** for it, but the program itself also includes it. In this case, the file is included twice, which makes extra, unnecessary work for the compiler.

Also, it is very easy to forget which include files are needed and which to leave out. I've often had to go through a cycle of compile and get errors, figure out which include file is missing, and compile again.

Therefore, the advantages of being compact must be balanced against the disadvantages of being complex and difficult to use.

One header does all

One way of avoiding the problems of multiple header files is to throw everything into a single, big header file. Microsoft Windows uses this approach. A typical Windows program contains the line:

```
#include <windows.h>
```

This is much simpler than the multiple include file approach taken by X Windows System. Also, there is no problem with loading a header file twice because there is only one file and only one **#include** statement.

The problem is that this file is 3,500 lines long, so even short 10-line modules bring in 3,500 lines of include file. This make compilation slower. Borland and Microsoft have tried to get around this problem by introducing “precompiled” headers, but it still takes time to compile Windows programs.

Mixed approach

Borland's Turbo Vision library (TV) uses a different method. The programmer puts **#define** statements in the code to tell the TV header which functions will be used. This is followed by one **#include** directive.

```
#define Uses_TRect
#define Uses_TStatusLine
#define Uses_TStatusDef
#define Uses_TStatusItem
#include <tv.h>
```

The file *tv.h* brings in additional include files as needed. (The **#define**s determine what is needed.) One advantage over multiple include files is that the files are included in the proper order, which eliminates redundant includes.

This system has another advantage in that only the data that's needed is brought in, so compilation is faster. The disadvantage is that if you forget to put in the correct **#define** statements, your program won't compile. So while being faster than the all-in-one strategy, it is somewhat more complex.

Program Aesthetics

A properly designed program is easy to read and understand.

Part of what makes books readable is good paragraphing. Books are broken up into paragraphs and sentences. A sentence forms one complete thought, and multiple sentences on a single subject form a paragraph.

Code paragraphs

Similarly, a C program consists of statements, and multiple statements on the same subject form a conceptual block. Since “conceptual block” is not a recognized technical term, you may just as well call them paragraphs. In this book, paragraphs are separated from each other by a blank line. You can separate paragraphs in C in the same way.

Omitting paragraphs in code creates ugly, hard-to-read programs. If you’ve ever tried reading a paper without paragraphing, you realize how easy it is to get lost. Paragraph-less programming tends to cause the program to get lost:

```
/* Poor programming style */
void display(void)
{
    int start, finish;  /* Start, End of display range */
    char line[80]:      /* Input line for events */
    printf("Event numbers? ");
    start = -1;
    finish = -1;
    fgets(line, sizeof(line), stdin);
    sscanf(line,"%d %d", &start, &finish);
    if (start == -1)
        return;
    if (!valid(finish))
        finish = start;
    if (valid(start))
        display2(start, finish);
}
```

Now, see how much better the same code looks after adding some whitespace to separate the function into areas:

```
/* good programming style
void display(void)
{
    int start, finish;  /* Start, End of display range */
    char line[80]:      /* Input line for events */

    printf("Event numbers ? ");
    start = -1;
    finish = -1;

    fgets(line, sizeof(line), stdin);
    sscanf(line,"%d %d", &start, &finish);

    if (start == -1)
        return;

    if (!valid(finish))
        finish = start;

    if (valid(start))
        display2(start, finish);
}
```

Note that the paragraphs here are not defined by the syntax of the language, but by the semantics of the program. Statements are grouped together if they belong together logically. That judgement is made by the programmer.

Rule 1-3:

Use white space to break a function into paragraphs.

Statements

Good paragraphing improves the aesthetics, hence the readability, of a program. But there are also aesthetic issues at the level of the sentence; or in C, the statement. A statement expresses a single thought, idea, or operation. Putting each statement on a line by itself makes the statement stand out and reserves the use of indentations for showing program structure.

```
/* Avoid this style of programming */
void dump_regs()
{
    {int d_reg_index;for(d_reg_index=0;d_reg_index<7
     d_reg_index++)printf("d%d 0x%x\n",
     d_reg_index, d_reg[d_reg_indexl);););
    {int a_reg_index;for(a_reg_index=0;a_reg_index<7
     a_reg_index++)printf("a%d 0x%x\n",
     a_reg_index, a_reg[a_reg_indexl);););
}
```

Figuring out this code is like extracting a fossil from a rock formation. You must take out your hammer and chip at it again and again until something coherent emerges. This kind of programming obscures the control flow of the program. It hides statement beginnings and endings and provides no paragraph separations.

Simply reformatting this code gives us a clearer understanding of what it does.

```
/* Better style */
void dump_regs()
{
    {
        int d_reg_index = 0; /* Data register index */

        for (d_reg_index = 0;
             d_reg_index < 7;
             ++d_reg_index)
            printf("d%d Ox%x\n",
                   d_reg_index, d_reg[d_reg_indexl);
        }
    }
    {
        int a_reg_index; /* Index of the address reg */

        for (a_reg_index = 0;
             a_reg_index < 7;
             ++a_reg_index)
            printf("a%d Ox%x\n",
                   a_reg_index, a_reg[a_reg_indexl);
        }
    }
}
```

(Better yet, add comments after the `d_reg_index` and `a_reg_index` declarations to explain the variables.)

Rule 1-4:

Put each statement on a line by itself

In clearly written English there are limits on the optimum length of a sentence. We've all suffered through the sentence that runs on and on, repeating itself over and over; or, through a structure whose complexity demonstrates more the confusion than the cleverness of the author (although it should be noted that, as in the present example, a demonstration of confusion can be the whole point), just get all bollixed up.

Likewise, a clearly written C statement should not go on forever. Complex statements can easily be divided into smaller, more concise statements. For example:

```
/* Poor practice */
ratio = (load * stress - safety_margin -
         fudge_factor) / (length * width * depth -
         shrinkage);

/* Better */
top = (load * stress - safety_margin - fudge_factor);
bottom = (length * width * depth - shrinkage);

ratio = top / bottom;
```

Rule 1-5:

Avoid very long statements. Use multiple shorter statements instead.