CS Department
Program Style Guide

Last revised: January 1998

Programming projects will consist, usually, of programs to be written by you. While you can and indeed should work with others on the design of a homework programming assignment, the implementation should be yours alone.

Good programming style is an important feature of any good program. Gary Litvin of Skylight Software gives the following explanation on why a working program with poor programming style is not good enough:

> First, a programmer's product is not an executable program but its source code. In the current environment the life expectancy of a "working" program is just a few months, sometimes weeks. On the other hand, source code, updated periodically, can live for years; to be of any use, it must be readable to others. Second, bad or unconventional style is uncool. It immediately gives away an amateur, a kind of social misfit in the software developers' culture. As nerdy as this culture might be, it has its own sense of esthetic pride.
>
> [from *The 32 Bits of Style*, http://www.concentric.net/~skylit/style]

Jump directly to:

Guidelines

- Structure, simplicity, and even neatness are crucial considerations for the production of working, high-quality programs.
- For a program to be used, it must be well-organized, coherent, and easy to understand. Since real application programs are frequently modified, future users need to be able to read a program easily.
- A well-written program reflects a well-conceived design; such a program is almost always simpler and more efficient than a hastily written or jumbled one.

Documentation

See the Documentation Standards document for a description and example of program headers and for a description of code documentation.

In addition to the comments inside a program, you should create a User's Guide which gives the name of the program, directions for using it, a short description of the purpose of the program, your name, the date when you wrote the program, and dates and comments on any subsequent modifications you wrote.

Program Format

Formatting refers to the way statements are placed in a program -- indentation, blank lines, etc. Good formatting can make a program far easier to understand.

White space is essential for clarity. This includes spaces around identifiers, operators, at the beginning and end of comments, as well as blank lines, especially to separate blocks and different sections of blocks.

Consistent indentation is essential for readability. Indentation should be used to *indicate that the indented statements are under the control of the previous non-indented statement*. Use this as a test when deciding to indent. Always indent inside the { } bracket pair. Examples in C or C++:

- Inside a block:

```
{
    statement 1;
    statement 2;
    etc.
}
```

- Inside a **for** or **while** or **do while** loop:

```
for (k = 1; k < = MaxSize; k++)
{
    Sum = Sum + k;              // under the for's control
    Sum2 = Sum2 + k * k;
}
```

- With an **if then else** construct:

```
if (k == 10)
    printf ("Error in ignition system.\n")
else
    printf ("All systems are GO.\n");
```

Coding Style

Keep statements reasonably short. Try to avoid statements that take more than one line. Keep the number of lines in a block down so that a block fits on one screen (about 21 lines), excluding initial documentation. Then the block can be scanned easily for content, coherence, use of variables, etc.

Choose names for your variables and other identifiers that are meaningful in the context of their use.

Avoid the use of constants within program statements. Rather declare them as named constants so that a future user can change their values easily for different applications.

Capitalization

Here are the capitalization rules you are expected to use in your programs for homework, tests, and projects:

1. In general, use lower case. C and C++ are case-sensitive; reserved words must be in lower case.
2. In C or C++, capitalize macro names.

3. Capitalize the first letter of user-defined types (including classes) as well as variables and constants of global scope.
4. Use mixed case in a reasonable fashion for the names of identifiers, programs, procedures, and functions. For example, a variable that contains the number of words in a sentence might be named numWords or num_words.

Program Grading

Your programs will be graded according to the following four components:

1. **Program correctness:** Each program should conform to specifications stated in the problem statement. A program should demonstrate correct handling of ordinary input, special cases, and error conditions. Test data should include typical values, out-of-range values, boundary values and special case values.
2. **Program design:** Your program should be modularized into small coherent independent functions or classes, with weak coupling and strong cohesion.
3. **Style and documentation:** Your program should be easy to read and understand. This involves program indentation, modular design, variable names, user interface and program comments. Appropriate pre-conditions and post-conditions should be included for each function.
4. **Efficiency:** Algorithms chosen should be efficient in regards to both time and space. Algorithms should be written cleanly and clearly rather than written using brute-force. You should be prepared to justify your choice of algorithms.

CS Department Documentation Standards

Last revised: September 2000

NOTE: Due to an unfortunate error, this document currently describes appropriate program documentation for Java programs only. The previous C and C++-related version of this file will return sometime soon.

Synopsis

The following is a description of expected program documentation. See also the Program Style Guide. Further documentation and style information is available at http://www.javaranch.com/coop.html

Comments

Comments allow us to write in natural language to clarify the intent of the program. Comment programs internally with clear, coherent statements in concise English, using good grammar, punctuation, and spelling. You might try writing the comments first, then the program to comply with the comments.

What comments should always be in a program?

Header Comments

At the beginning of anyfile you should have at least:

- A *descriptive* title for the program, function(s), or class(es) in the file.
- Your name and the date.
- Dates and comments on subsequent changes. If the person making the modifications is not the original author, leave the original author's name but identify the name of the modifer as such.

You should also include class JavaDoc documentation immediately preceding the class declaration. The class JavaDoc documentation must include:

- brief synopsis
- detailed description
- example code segment using the class
- author list using the JavaDoc @author tag

Note that because of a "feature" in JavaDoc, you must precede all example code with an asterisk on each

line to preserve your indenting.

The following format for the comments at the beginning of a program or function is recommended:

```
/*
    IntVector Class (Dynamic Array of Ints)
    Author: Justin Case
        with assistance from: Adam Baum
    Creation Date: 12 September 2000
        Modified:    15 September 2000    Jeanette Winterson
            Changed implementation of append()
*/
/** A vector class optimized for working with ints. <p>
    Like the Vector object, except rather than tracking a dynamic
    array of pointers to different objects, this is simply a
    dynamic array of ints.  The advantage is speed and memory
    savings.<p>

    Example:
    <pre>
    *
    *        // report longest lines
    *        TextFileIn f = new TextFileIn("blather.txt");
    *        IntVector v = new IntVector();
    *        int longestLine = 0 ;
    *        boolean done = false ;
    *        while ( ! done )
    *        {
    *            String s = f.readLine();
    *            if ( s == null )
    *            {
    *                done = true ;
    *            }
    *            else
    *            {
    *                int sLength = s.length() ;
    *                if ( sLength > longestLine )
    *                {
    *                    longestLine = sLength ;
    *                }
    *                v.append( sLength );
    *            }
    *        }
    *        f.close();
    *        System.out.println("The longest lines are on line
    numbers:");
```

```
    *        for ( int i = 0 ; i

    @author Adam Baum
    @author Justin Case
*/

public class IntVector
{
    ...
}
```

The method JavaDoc documentation must include:

- brief synopsis
- detailed description (if anything can be added to the brief synopsis)
- parameter list using the javadoc @param tag (if there are any parameters)
- return value using the javadoc @return tag (if anything is returned)
- exception list using the javadoc @exception tag (if there are any exceptions thrown)

This information should be immediately above each function implementation. Here is an example:

```
/** Get a copy of one int. <p>
    Retrieve an int relative to the index provided.<p>

    @param Index Which int (0 is the first int).<p>
    @return The retrieved int or zero if Index is outside of
    0..length.<p>
*/
public int get( int Index )
{
    ...
}
```

http://www.javaranch.com/coop.html

Variable Declaration Comments

In the declaration section, provide a short description of the logical role played by each variable. Organize this section as a legend. For example, in a Java program you might have:

```
    double weight;           // The package's weight in kilograms
    Address destination;     // The address to which the package
    should be sent
```

Comments for Code Sections

Each **section that performs a task** should be preceded by a comment explaining the **purpose** of the

section and the **algorithm** being used to carry it out. Individual statements that are complex or non-intuitive should also be preceded by comments (or followed by a comment if it fits on the same line). **An end brace after a long block** should have a short comment connecting it to its respective beginning brace. You should also include such comments for end braces when you have a number of nested blocks. Here is an example in Java:

```
// CALCULATE THE AVERAGE by summing the values and
// then dividing by the number of values.
sum = 0;
for ( i = 0; i <  max; i++ )
{
        sum = sum + values[i];          // accumulate sum
        // other statements if necessary ...
}       // end of loop calculating sum
avg = sum / max;
```

Use blank lines to separate different sections of the program (see the Program Style Guide).

The 32 Bits of Style

by
Gary Litvin

Skylight Publishing
9 Bartlet Street, Suite 70
Andover, MA 01810
(978) 475-1431
http://www.skylit.com
e-mail: support@skylit.com

This document is available on the Internet:
http://www.skylit.com/style

To Maria, who appreciates style, Aaron, who has a keen sense of it,
and Marg, who has spent hours fixing my grammatical
and stylistic blunders.

The language is perpetually in flux: it is a living stream, shifting, changing, receiving new strength from a thousand tributaries, losing old forms in the backwaters of time. To suggest that a young writer not swim in the main stream of this turbulence would be foolish indeed, and such is not the intent of these cautionary remarks. The intent is to suggest that in choosing between formal and the informal, the regular and the offbeat, the general and the special, the orthodox and the heretical, the beginner err on the side of conservatism, on the side of established usage."

William Strunk Jr. and E.B. White, *The Elements of Style*

Preface

Style is a crucial component of professionalism in software development. Clean code that follows stylistic conventions is easier to read, maintain, and share with colleagues. Programmers who code in good style are less likely to have silly bugs and will actually spend less time developing and debugging their code. Finally, good style in programs is a concern because for us, humans, style and esthetics are a concern in everything we do.

As far as teachers are concerned, programs written in good style are easier to grade, and a program will look better if, perchance, it wins a prize in a computer science contest and is published in the organizer's newsletter. There is also a negative side, of course: it is harder to recognize plagiarism when all student programs look alike.

Following stylistic conventions is easy, and, after a little practice, becomes second nature. Occasionally, an especially independent-minded student resists all conventions, arguing, "But it works!" This attitude is supported by some teachers' grading criteria and by the rules of APCS exams and programming contests. But a teacher who believes that style is important can usually persuade even the most obstinate "free-thinkers" by pointing out two things. First, a programmer's product is not an executable program but its source code. In the current environment the life expectancy of a "working" program is just a few months, sometimes weeks. On the other hand, source code, updated periodically, can live for years; to be of any use, it must be readable to others. Second, bad or unconventional style is uncool. It immediately gives away an amateur, a kind of social misfit in the software developers' culture. As nerdy as this culture might be, it has its own sense of esthetic pride. If all other means of persuasion fail, the teacher can start taking off lots of points for bad style a couple of weeks into the course.

In *The Elements of Style*, William Strunk wrote:

> "There is no satisfactory explanation of style, no infallible guide to good writing, no assurance that a person who thinks clearly will be able to write clearly, no key that unlocks the door, no inflexible rule by which the young writer may shape his course."

These words, which come in the book's closing chapter, describe (and illustrate) the deeper meaning of style — after the straightforward rules of correct usage have been discussed and illustrated in the previous chapters. In programming, too, style has a deeper meaning: it is that elusive quality which makes one person's code elegant and easy to follow and another person's convoluted and obscure although in line with all the

superficial stylistic conventions. The mystery is not as wide open in writing code as in creative writing: there are many firm design principles, and "a person who thinks clearly" usually is able to code clearly. Still, some mystery remains. In this brief paper I can only discuss the superficial stylistic conventions, leaving the "deeper meaning" alone.

A final disclaimer: stylistic conventions vary and keep changing. Every textbook, tutorial, and manual follows a certain style, and every programmer has his or her own stylistic idiosyncrasies. The generation of C/C++ programmers who learned C from K&R (Kernighan and Ritchie), for example, may be accustomed to a slightly different C++ style than the younger generations. Some of the specific advice given here reflects my own taste, not universal convention. Still, the overwhelming majority of programmers agree that style is important, and the main elements of style are universally accepted by software professionals.

1. Head over Heels

The source code of a program proceeds in certain order. This order is mostly determined by C++ syntax: things have to be defined before they can be used. But some of it is a matter of style. In any case, the guidelines discussed in this chapter are fairly standard.

We will use a simple program, Grades, (Figure 1) as an example. The program prompts the user to enter a number of student grades and displays the average grade.

Bit 0. At the top of each source module, put in a comment that states the purpose of the program or the module, its author, date of completion and other pertinent information, such as a copyright message, special instructions on how to run the program, data files that your program reads or creates, and a history of revisions.

Bit 1. Place all `#include` directives near the top of your source module. Start with system header files (in angular brackets) followed by your own or your organization's header files (in double quotes).

Bit 2. Place system- or compiler-dependent comments and directives near the top of your module.

Bit 3. Place global definitions, such as `typedef`s, `enum`s, definitions of structures and classes, global constants, and so on at the top of your program, after the `#include`s.

Normally, these definitions would be placed in your own header file, but there is no need to complicate things in the school computer lab, especially at the beginning.

Bit 4. Avoid C-style `#define` macros. Use `const` definitions and `inline` functions instead. (*Inline* functions are not a part of the AP subset.)

C++ constants and inline functions are better because they enforce data types.

Bit 5. Functions can be placed either before `main()` or after it, depending on your preference and the overall design. If you place your functions after `main()`, at least some prototypes are required, and it is better to provide prototypes for all your functions. Group these together near the top of the module, after the user-defined data types and constants (or put all the prototypes in your header file).

The code of very short, self-explanatory inline functions may belong with the prototypes. Serious OOP projects avoid free-standing functions altogether: programmers use classes that are usually placed in separate modules, with only one free-standing function, `main()`.

```
// GRADES.CPP
//
// This program prompts the user to enter a student's grades and
//   displays their average.
//
// Author: Mark Avgerinos
// Date 12-31-1999
//

#include <iostream.h>
#include <iomanip.h>
#include <ctype.h>

// Comment out the line below if your compiler has
//   a built-in bool type.
#include "bool.h"
#include "apvector.h"

const int MAXGRADES = 20;

// Function Prototypes:

int Enter(apvector<int> &grades);
double Average(int n, const apvector<int> &grades);

//*************************************************************
//*****************            Main           *****************
//*************************************************************

int main()

{
    int n;
    apvector<int> grades(MAXGRADES);
    char more;

    // Set output formatting to display one digit after decimal point.
    cout << setprecision(1);
    cout.setf(ios::fixed | ios::showpoint);

    do
    {
        n = Enter(grades);
        if (n > 0)
            cout << endl << "The average grade is "
                 << Average(n, grades) << endl;

        cout << endl << "Another student? (y/n): ";
        cin >> more;
        cin.ignore(1000, '\n');

    } while (tolower(more) == 'y');

    return 0;
}
```

```
//****************************************************************
//******************          Functions        ******************
//****************************************************************

int Enter(apvector<int> &grades)

// Prompts the user for the number of grades and the student's grades
//   and places grades into the grades array.
// Returns the number of student grades entered.

{
    int i, n = 0, len = grades.length();

    cout << "Number of grades: ";
    cin >> n;
    if (n <= 0)
        return 0;
    if (n > len)
    {
        cout << "*** The maximum allowed number is " << len << " ***\n";
        n = len;
    }

    cout << "Enter " << n << " grades: ";
    for (i = 0;   i < n;   i++)
        cin >> grades[i];

    return n;
}

//****************************************************************

double Average(int n, const apvector<int> &grades)

// Assumes n > 0.
// Returns (a[0] + a[1] + ... + a[n-1]) / n

{
    double sum = 0.;
    int i;

    for (i = 0;   i < n;   i++)
        sum += grades[i];

    return sum / n;
}
```

Figure 1. The Grades program.

Bit 6. Use arguments' names, not just their data types, in function prototypes.

For example,

```
void test(int,int,int);
```

is not very informative.

Bit 7. Separate different sections of your program with blank lines and separator comment lines.

Bit 8. Split your code into "paragraphs" that represent meaningful steps or actions in your program by inserting blank lines and, if necessary, comment lines. For example:

```
...
int i, n = 0, len = grades.length();

cout << "Number of grades: ";
cin >> n;
if (n <= 0)
    return 0;
if (n > len)
{
    cout << "*** The maximum allowed number is " << len << " ***\n";
    n = len;
}

cout << "Enter " << n << " grades: ";
for (i = 0;   i < n;   i++)
    cin >> grades[i];

return n;
...
```

2. Spaces and Braces (and Parentheses)

The formatting of program text is governed only by the rules of style. With the exception of preprocessor directives, literal strings in double quotes, and single-line comments, C++ compilers do not require spaces around operators or punctuation and do not care how the program text is split between lines. Code with arbitrary spacing and line breaks (Figure 2) will compile with no problems.

```
const int MAXGRADES=20;int Enter(apvector<int> &grades);double Average(int
n,const apvector<int> &grades);int main(){int n; apvector<int>
grades(MAXGRADES); char more;
cout << setprecision(1);cout.setf(ios::fixed|ios::showpoint);  do{
n = Enter(grades); if(n>0)cout<<"\nThe average grade is "
<< Average(n, grades)<<"\n";cout<<"\nAnother student? (y/n): ";
cin >> more;cin.ignore(1000,'\n');}while (tolower(more) == 'y');
return 0;}int Enter(apvector<int> &grades){int i,n=0,len=grades.length();
cout<<"Number of grades: ";cin>>n;if (n <= 0)return 0;if(n>len){
cout << "*** The maximum allowed number is " << len << " ***\n";n=len;
}cout<<"Enter "<<n<<" grades: ";for (i=0;i<n;i++)cin>>grades[i];return n;}
```

Figure 2. A fragment from the Grades program with no formatting — which works fine.

This is a little extreme, but some real world examples are not too far from it. Figure 3 shows a fragment from an American Computer Science League 96-97 Senior Division's "perfect ten" program, published in the ACSL booklet:

```
#include <stdio.h>
ints[8][8];
int pent(char r,int xc,int uc,int k)
{   int x,y,c=1,d[8][8];
    for(x=0;x<=7;x++) for(y=0;y<=7;y++) d[x][y]=s[x][y];
    d[xc][yc]++;
    if((r==65)     &&(xc>5||xc<0||yc>7||yc<2)) c=0;
    ...
```

Figure 3. A fragment from a "perfect ten" ACSL competition program.

Bit 9. Place each statement on a separate line. Indent.

Code should be indented, usually by two-four character positions, in the body of a function, within braces, and under `if`, `else`, `switch`, and `for`, `while`, and `do-while` loops . The same indentation offset should be used throughout your code (see Figure 1). Four spaces is a commonly used indentation step.

Indentation is useful only for human readers: compilers don't care. Suppose you write:

```
if (date == 13)
    if (weekday == 5)
        cout << "Friday, the 13th\n";
else
    cout << "An uneventful day\n";
```

It will actually be compiled as:

```
if (date == 13)
{
    if (dayOfWeek == 5)
        cout << "Friday, the 13th\n";
    else
        cout << "An uneventful day\n";
}
```

You must use braces to express your intentions correctly:

```
if (date == 13)
{
    if (dayOfWeek == 5)
        cout << "Friday, the 13th\n";
}
else
    cout << "An uneventful day\n";
```

Bit 10. Keep braces visible.

There are different styles of placing braces. In the K&R style, the opening brace is placed at the end of the previous line, and the closing brace on a separate line. Some people prefer to place both the opening and the closing braces on separate lines:

```
if (n > len)
{
    cout << "*** The maximum allowed number is " << len << " ***\n";
    n = len;
}
```

This way it is easier to see the opening brace and to cut and paste blocks of code. But the program listing is a bit longer, and it is easier to slip in an extraneous semicolon before the opening brace — a bug that is hard to see. For example:

```
while (getline(dataFile, line) && n < MAXLINES);
{
    data[n] = line;
    n++;
}
```

There is no need to clutter the code with redundant braces. For instance,

```
    if (!dataFile)
        return false;
```

looks as good to me as

```
    if (!dataFile)
    {
        return false;
    }
```

But always use braces with a `do-while` loop, even if the body of the loop has only one statement. For example:

```
do
{
     x *= 3;
} while (x < limit);
```

Never place braces between `else` and `if` on the same line:

```
...
else { if (...)      // Bad style!
     }
```

Bit 11. Use spaces liberally, do not cram things together.

There are no clear-cut rules on where to add spaces — different people have different tastes.

I use spaces on both sides of the assignment operator and with most other binary operators: arithmetic, logical, relational, `<<` and `>>` operators, bit-wise operators, and so on. For example, I find

```
for (i=0;i<n;i++)
    sum+=grades[i];
```

less readable than

```
for (i = 0;   i < n;   i++)
    sum += grades[i];
```

I add two or three spaces after each semicolon in a `for` loop because I find it easier to read, especially if the initialization, condition, and/or increment parts are long. For example:

```
for (i = 0, j = n - 1;   i < j;   i++, j--)
    ...
```

Occasionally I omit spaces around the multiplication operator in short expressions to make them look more like in algebra. For example:

```
a[2*i + 1] ...
```

With longer names, spaces become more important. For example:

```
taxAmount=saleAmount*taxRate;
```

appears too crammed to me. I prefer

```
taxAmount = saleAmount * taxRate;
```

Unary operators, such as `-` (negation) , `++`, `--`, `!`, `~`, `&` (address of), are usually attached to their operands. For example:

```
if (! match)     // Too fancy
    count --;
```

is too fancy. People normally write:

```
if (!match)
    count--;
```

Most people do not use spaces around brackets, the scope resolution operator `::`, or the "dot" or `->` structure or class member access operators. Believe it or not,

```
msg . text = ...
```

compiles correctly, but you won't see it in programs too often. We simply write

```
msg.text = ...;
```

Some people like to leave spaces around parentheses in expressions and in function prototypes and calls. For example, they write:

```
int Enter ( apvector<int> &grades );
double Average ( int n, const apvector<int> &grades );
```

or

```
int Enter (apvector<int> &grades);
```

I don't mind either, but I find the former version a little showy. I do leave a space before the opening parenthesis when the function's name or the argument list is long. I also leave a space before the opening parenthesis in `if`, `for`, `while`, and `switch` statements. For example:

```
if (n > len)
{
    ...
}

for (i = 0;   i < n;   i++)
    cin >> grades[i];
```

Newcomers to C or C++ often complain about its cryptic syntax. One of the reasons for this, I think, are the multiple uses of the `*` and `&` symbols. The asterisk is used as the multiplication operator, in declarations of pointers, and as the pointer-dereferencing operator. The ampersand is used in `&&` as the logical "AND," as the bit-wise "AND" operator, as the "address of" operator, and in declarations of references. (The "address of" and bit-wise "AND" are not in the AP subset.)

I add spaces on both sides of `*` and `&` when they are used as a binary operators (multiplication and bit-wise "AND," respectively). For example:

```
z = x * y;
status = mask & statusBit;
```

On the other hand, I attach `*` and `&` to the variable's name when they are used to declare pointers or references, in dereferencing, or as the "address of" operator:

```
TREENODE *root;
void Swap(int &x, int &y);
*this = rhs - *this;
```

In the AP Big Integer Case Study and in the AP classes, you can see spaces on both sides of the `&` in declarations of reference variables and reference arguments. For example:

```
Optype StringToOp(const apstring & s);
```

It is hard for me to get used to this because I am so "programmed" to perceive an isolated `&` as the bit-wise "AND" operator. This may be not a problem for others, because bit-wise operators are not part of the AP subset.

My general approach to these details is lax: use whatever seems most readable and try not to be too fancy or dogmatic.

Bit 12. Omit needless parentheses.

C++ uses a well-defined order of precedence among operators. All unary operators have higher rank than (are applied before) all binary operators. Among the binary operators, arithmetic operators have higher rank than relational operators, which in turn apply before logical operators. There is no need to clutter your code with redundant parentheses. For example, instead of

```
if ((!match) && (i < (length - 1)))...
```

you can write simply:

```
if (!match && i < length - 1)...
```

But you have to be a little more careful with && ("AND") and || ("OR"): && has a higher rank than ||. For example, you need to keep the parentheses in

```
if (year % 4 == 0 && (year % 100 != 0 || year % 400 == 0))...
```

Occasionally, redundant parentheses facilitate reading long logical expressions:

```
if ((x > -3 && x < - 1) || (x > 1 && x < 3))...
```

3. Comments

Bit 13. Comment each function and operator: state its purpose, arguments, assumptions ("preconditions"), results ("postconditions"), return value, and side-effects, if any.

Bit 14. Avoid redundant comments.

```
if (a[i] >= 0)
    count++;   // Increment the count
```

Bit 15. Self-explanatory code is usually better than heavily commented code.

For example, instead of

```
if (s == "MA" && a < 21)     // If state is Massachusetts and
                             //   age is less than Massachusetts
                             //   legal drinking age...
    ...
```

it is better to write:

```
const int massDrinkingAge = 21;
if (stateCode == "MA" && age < massDrinkingAge)
    ...
```

Prompts and error messages often serve as "comments." For example:

```
if (n > len)
{
    cout << "*** The maximum allowed number is " << len << " ***\n";
    n = len;
}

cout << "Enter " << n << " grades: ";
for (i = 0;   i < n;   i++)
    cin >> grades[i];
```

Bit 16. Explain your intentions ahead of time in difficult algorithms.

```
// Find the smallest element in the right subtree
//   (it must be the leftmost element in that subtree):
node = root->right;
while (node->left)
    node = node->left;
```

Bit 17. Comment recursive functions.

```
void TowersOfHanoi (int fromPeg, int toPeg, int nRings)

// This function solves the "Towers of Hanoi" puzzle.

{
    int spare;

    if (nRings == 0)   // If nothing to do...
        return;

    // Move the top n - 1 rings from fromPeg to spare:
    spare = 6 - fromPeg - toPeg;             // 6 = 1 + 2 + 3
    TowersOfHanoi (fromPeg, spare, nRings - 1);

    // Move the remaining ring from fromPeg to toPeg:
    cout << "Move from " << fromPeg << " to " << toPeg << endl;

    // Move n - 1 rings from spare to toPeg:
    TowersOfHanoi (spare, toPeg, nRings - 1);
}
```

Bit 18. Comment obscure code or unusual usage.

In the following example the binary representation of an `int` variable is used to generate all bit patterns of nine 0's or 1's:

```
    unsigned int mask, bit;
    int i, sum;

    // Generate all bit patterns from 000000000 to 111111111 (9 bits):
    for (mask = 0;   mask <= 0x1FF;   mask++)
    {
                                  // hex 1FF is binary 111111111
        bit = 0x001;              // hex 001 is binary 000000001
        sum = 0;
        for (i = 0;   i < 9;   i++)
        {
            if (mask & bit)       // If this bit is set...
                sum += a[i];      //   then add a[i] to the sum
            bit <<= 1;            // Shift left by one (to the next bit)
        }
        ...
```

(Hex constants, bit-wise "AND," and shift operators are not in the AP subset.)

4. Names

Apt naming is crucial to making your program readable and to avoiding silly bugs.

Bit 19. Use meaningful names.

```
tot = a * (1 + r);
```

is better written as

```
totalAmt = saleAmt * (1 + taxRate);
```

But overly long COBOL-style names will clutter your code:

```
totalAmountDue = totalSaleAmount * (1 + salesTaxRate);
```

In C++, function names do not have to be too long. The same name may be used for functions that take arguments of different types without a conflict (a feature known as *function overloading*). Member functions in different classes may have the same name, too. In fact, the object-oriented programming approach encourages the use of the same name for functions that perform semantically similar tasks. The argument's name or the class object's name often work together with the function name to create a readable combination. For example:

```
n = Enter(grades);
Display(image);
Display(msg);
Insert(tree, lastName);
list.Insert(lastName);
vendingMachine.Insert(change);
```

Bit 20. Use simple names for loop-control variables. Declare these variables locally in each function and use the same name where appropriate.

There is nothing wrong with using names like `i` or `k` for a subscript or for a loop control variable. Note that longer names, such as `index`, `subscript`, `counter`, `lcv`, or `loopControlVariable` are no more meaningful than `i`.

If possible, use the same name for similar purposes. Note, for example, how the same local variable names `r` and `c` are used throughout the `Marine Biology` case study code to represent the row and column in the matrix of fish:

```
for (r = 0; r < NumRows(); r++)
{
  for (c = 0; c < NumCols(); c++)
  {
      ...
```

Bit 21. Use the upper and lower case consistently.

C++ is case sensitive, and all reserved words are lowercase. Other than that, there are no syntax rules for using the upper or lower case; stylistic conventions vary, too. It is still useful to set up a convention for yourself, your group, or your organization. One possibility:

- Start all variable names with a lowercase letter and capitalize subsequent words:
  `carry`, `totalAmt`, `fileName`
- Start all free-standing function names with a capital letter:
  `Average(…)`, `Enter(…)`
- Start all member function names in user-defined classes with a capital letter:
  `IsNegative(…)`, `NumDigits(…), AddSigDigit(…)`
- Use all caps for global constants and `enum` list elements;
  `BASE`, `enum COLOR {RED, GREEN, BLUE}`;
- Use all caps for user-defined types;
  `typedef unsigned char BYTE`, `class BIGINT`

... and so on.

Bit 22. If you have to use global constants or variables, give them unique, conspicuous names.

Programmers often use all caps for global constants — a style that goes back to K&R. For example:

```
const int MAXGRADES = 20;
const int BASE = 10;
```

Some programmers use all caps for user-defined data types, too:

```
struct MESSAGE {...
enum SIGN {...};
```

But others only capitalize them:

```
class BigInt {
enum Sign {...};
```

Bit 23. Use a standard prefix in names of class data members.

The data members of a class act more or less like global variables within the class's scope, and their names often clash with reasonable names for arguments and local variables in member functions. This is not only annoying but may cause nasty bugs. To avoid the conflicts and to make class data members more visible in the code, programmers often start all data members' names with a standard prefix, for instance, "m" or "my." For example:

```
class apstring
{
    ...
    int myLength;              // Current length of the string
    int myCapacity;            // Capacity of the character array
    char *myCstring;           // Pointer to the character array
};
```

5. Style or Substance?

The following simple rules are on the fuzzy border line between style considerations and sound design principles.

Bit 24. Avoid global variables.

Loop control variables and other general-purpose variables with names like `i`, `count`, `sum`, `amt`, declared globally, look especially ridiculous and are also dangerous (Figure 4).

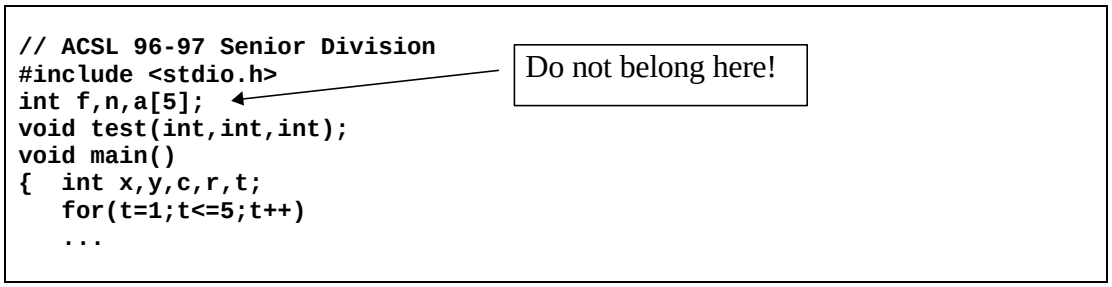

Figure 4. A fragment from an ACSL "perfect ten" program with global variables

Bit 25. Pass structures and class objects to functions by reference. Add the `const` keyword if the function does not change the object.

For example:

```
int Enter(apvector<int> &grades);
double Average(int n, const apvector<int> &grades);
```

Passing by reference is simply a matter of efficiency.

Bit 26. Do not repeat the same calculations or function calls.

```
if (x*x + y*y < 1)
    d = 1. / sqrt(x*x + y*y);
```

is less efficient than:

```
double r2 = x*x + y*y;
if (r2 < 1)
    d = 1. / sqrt(r2);
```

And

```
if (Average(n, grades) >= 92)
    letterGrade = 'A';
else if (Average(n, grades) >= 85)
    letterGrade = 'B';
else if ...
```

is unacceptable. Instead, write:

```
double avgGrade = Average(n, Grades);
if (avgGrade >= 92)
    ...
```

Bit 27. Function calls in expressions are okay.

A novice is often afraid to use function calls in expressions. Notwithstanding the previous rule, there is nothing wrong with using a function call in an expression as long as you do not make the same call repeatedly.

For example,

```
int na = a.ToInt();
cout << "a = " << a << " as int = " << na << endl;
double xa = a.ToDouble();
cout << " as double = " << xa << endl;
```

is an overkill. More concisely and just as clearly, try:

```
cout << "a = " << a << " as int = " << a.ToInt() << endl;
cout << " as double = " << a.ToDouble() << endl;
```

Bit 28. It is okay to use logical expressions as values of the bool type.

```
if (count >= 3)
    return true;
else
    return false;
```

may be written more concisely as:

```
return (count >= 3);  // Parentheses optional
```

Bit 29. Do not scatter input and output statements all over your program. Either use dedicated functions for data input and output, or keep most of the user interface in the `main()` function.

In the Grades program, for example, it would be bad style to print out the average grade inside the `Average(…)` function that calculates it. The function is more general if it returns the average value to the calling program; the calling program then prints it out, together with other appropriate values or messages, using a desired format. This way, later programs that need to calculate averages for non-printing purposes can re-use the `Average(…)` function.

On the other hand, the `Enter(…)` function is dedicated to entering grades into an array. A function of this type displays all the pertinent prompts and error messages, handles data input, and places the data into the specified place, in this case into the `grades` array. It is bad style to mix data entry tasks with serious computations in the same function.

Bit 30. Code defensively.

```
while (n-- > 0)
{
    ...
}
```

may save one line of code, but it may also lead to a bug if `n` is used inside the loop.

```
while (n > 0)
{
    ...
    n--;
}
```

is safer.

Bit 31. *Non Sibi* ("Not for Self") is the motto of Phillips Academy in Andover. Keep these words in mind when writing code.