Guidelines for C++ Program Style

Comp 314 Fall 1997

Introduction

There is no fixed set of rules according to which clear, understandable, and provable programs can be constructed. There are guidelines, of course, and good ones at that; but the individual programmer's style (or lack of it), his clarity of thought (or lack of it), his creativity (or lack of it), will all contribute significantly to the outcome.

-- Peter J. Denning

It is a major goal of our programming courses that you not only learn to produce correct programs, but that you learn to use a good style of programming. There are, of course, many different opinions about exactly what good style is. In these notes and in our classes, we will try to guide you to write *C* ++ programs in a particular style that we believe is good. It is the style we should use and the style you should use in all your *C* ++ programming courses at Rice.

We do not believe these notes reflect an *ideal C* ++ programming style. There may be several aspects of style presented here that differ from what is described in textbooks. Programming style can be a matter of strong and diverse opinion, and proponents of other styles are not necessarily wrong for disagreeing with ours.

In some cases, the choice of one stylistic convention over another may seem completely arbitrary. Why, then, declare one convention good and another reasonable one bad? For the same reason that, among two equally good spellings for the word describing a feline, *cat* is better than *kat* . Both convey the same information, but the eye jerks to a halt on the word *kat* for an instant to interpret it, while it glides over *cat* without a pause. For whatever reason, *cat* is the spelling people expect to see. To communicate with a community of programmers, including those who grade your programs and help you with programming problems, it is best to speak fluently the language of that community. When everyone writes loops the same way, anyone can look at a looping piece of code and think *loop* instantly. When everyone uses their own unique style to write loops, each component must be decoded before the revelation *loop!* occurs.

Programming style primarily concerns human, rather than computer, efficiency. A well-styled program is to be preferred for three reasons. It is

easier, and therefore cheaper, to debug;

1. easier to test and show correct; and
2. easier to modify if the problem changes

Computer efficiency and good programming style sometimes come into conflict. The *C* family of languages, in particular, allows terse, cryptic, machine-imitating statements that may translate directly to efficient hardware instructions but leave the programming novice, or even the programming expert, baffled. Some of these constructions have become part of the idiom of C programmers, and so would be understood in a glance by an experienced hacker. Ultimately, though, when computer and human efficiency cannot coexist, a well-styled program is preferred to an incomprehensible one.

File Structure and Program Organization

A *C* ++ program is a collection of functions and variables in several files, with a function called *main* at the highest level. The files are generally of two types, *definition* files and *implementation* files. A definition file has a name ending with *.hh* and describes, for both the computer and human reader, how to use a corresponding implementation, or *.cc* , file. A *.cc* file that implements functions not intended for use by other computer programs may have no corresponding *.hh* file, and a *.hh* file that defines only simple data types and defines only inline functions may have no corresponding *.cc* file.

A good program design begins with an analysis of what the major components of the solution must be and how the components are related. For each of these components, there should be a file or collection of files that contain the functions that solve the particular problem component. The relations between the components should be made explicit, both in a Makefile and in *#include* directives at the start of programs. A *Makefile* instructs the *make* utility on how to produce an executable program from related source files. An *#include* directive makes available to one set of functions the definition file of another.

The name of a file should relate to its purpose. The names of corresponding implementation and definition files should match, except for the suffix. When several implementation files are required that correspond to a single definition file *Foo.hh* , the files should have names like *Foo-tweedle.cc* , *Foo-dee.cc* and *Foo-dum.cc* .

The *C* ++ programming language offers not only the usual procedural abstraction, but also data abstraction. The top-level program design might well include a description of fundamental object types that the program is to manipulate. A formal description of the members of a set and the operations that can be performed upon them defines an *abstract data type* . The *class* mechanism in *C* ++ allows the convenient use of abstract data types in programs. The implementation and definition files for an abstract data type should have the same name, up to the conventional suffix, as the type itself.

Variables, as well as functions, may be placed in implementation files. Such variables are *global* , accessible to several functions in the file and possibly to functions in other files as well. Global variables are not completely forbidden; however, a global variable should be as truly integral to the design of the program as a global function. Global variables appear most often as an indication of programmer laziness rather than program structure. The graders therefore regard them with great skepticism.

Functions

On the lines below a function heading, there should be two assertions, called pre- and post-conditions, that describe *what the procedure does* . The precondition states what assumptions the function implementer can make (and, conversely, what conditions the caller must ensure) just prior to the execution of the function. Similarly, the postcondition states what assumptions the caller can make (and, conversely, what conditions the function implementer must ensure) just after the execution of the function. Generally, these two assertions describe facts about the formal parameters and, possibly, global variables. Remember that, in its purest form, an assertion is simply a Boolean expression. Clarity and specificity, however, are more important than formality: it's better to use constructs, even English, that are less formal than Boolean expressions if you need to. If a precondition is trivial (i.e., simply *TRUE*), then it can be omitted. If a postcondition is trivial (i.e., no change of state occurs), then it can be omitted. You may, additionally, have other comments that describe something special about the implementation, such as its performance or method.

For example:

```
// The position of name within the roster

int position(const SortedStringList & roster, const String & name)

// Preconditions:

// For all 0 <= i < roster.n -1,

// roster.member[i] <= roster.member[i+1]

// Postconditions:

// Let pos denote returned value. Then

// For all 0 <= i < k, roster.member[i] < name and

// for all k <= i < roster.n, name <= roster.member[i]

// Method:

// Binary search of the sorted array

{

<function body>;

}
```

Values may be passed to functions either through value or reference parameters. Value parameters provide a function with local variables, initialized to the values of the actual parameters. Changes to these local variables do not affect the actual parameters. Reference parameters provide a function with the actual parameters directly, so that a change to a reference parameter changes the corresponding variable in the calling program. When the intent of a function is to modify its parameters, reference parameters are preferred. In particular, the C style, which faked reference parameters by passing-by-value pointers to the parameters to be modified, is not acceptable.

When the actual parameters of a function are not intended to be changed by the function, value parameters are preferred. However, the greater efficiency of reference parameter passing makes call-by-value unappealing for parameters that are large class objects. In such cases, it is acceptable to pass by reference, but the *const* keyword must be added to the formal parameter description, to check that the parameter is not unintentionally modified.

A function can produce values for the calling program in three ways: with a *return* statement, via output parameters, or by modifying the value of a global variable. A function that computes a value and does not alter its input parameters should use the *return* statement to provide a value, rather than using an output parameter. A function that produces several values cannot *return* them all separately, but can return a class object, where the data members of the class are the different values. If that is inconvenient, a function may return its values in reference parameters. However, a function that can alter its reference parameters should be of type *void* , and not return a value through the usual *return* mechanism. It can be very confusing to see a function call returning a value without realizing that it also changes some of the values provided to it. The output reference parameters should precede all the other parameters in the argument list of a function.

Global variables should not be used as a lazy way to return arguments to a function. However, when a global variable is appropriate, it is best to make it a class variable and use member functions and member procedures to access its value or to modify it.

In a similar vein, class methods should either return a value, or change the state of an object, *but not both* . If you want to do two things, use two lines (and don't worry about the overhead of making two function calls; see the section on efficiency). Some practitioners feel it is okay to violate this is during input: *while (f.read() != EOF_CHAR) {...}* . In such a case, consider the alternative *for (f.read(); !f.eof(); f.read()) {...}* ; this reduces the need for sentinel values, at the cost of having two points of control for *f.read()* .

The object-oriented nature of *C* ++ allows a function that manipulates the representation of an item of some class to be defined either as a friend to the class or as a member of the class. When writing a function that does something *to* a class object, make it a member function. When writing a function that tells something *about* a particular class object, use a member function. When writing a function that does something *with* a class object, make it a friend function.

The *C* ++ programming language also allows overloading of the standard C operators. This can enhance program readability; for example, using *s1 + s2* to form the concatenation of two strings is very natural. (Do you expect this to change the contents of *s1* ?). The usual arithmetic operators should only be used for functions that correspond naturally to their meanings in arithmetic, as with + above. Binary operators, whose operands are equally important, are better expressed as friend functions than as methods of the first operand. If you overload + , you should document whether or not you also overload += , and consider whether to overload ++ , - , -= , etc. Non-intuitive uses of operators should always be avoided.

There is one operator which should be overloaded for every class, and is probably one of the first methods you write in any class, for debugging purposes: << . You should not overload << and >> for anything but I/O. As for overloading some of the more obscure operators, it is a matter of personal style: is *mySet* <<= *i* particularly easier to read than *mySet.add(i)* ?

Flow of Control

The principal looping constructs of *C* ++ are *for* and *while* . The only difference between the two is that a *for* loop provides a convenient location for initializing and updating loop variables. Use a *for* loop when there is a variable or two that exist only to step through the loop. Such a variable could be counting the elements of an array or stepping through a linked list or down a tree path. Do not change an index variable of a *for* loop anywhere but in the heading of the loop. Do not initialize the index variable anywhere but in the immediate vicinity of the loop heading. The traditional way to count through a loop n times is *for (i = 0; i < n; ++i)* , so use this style in standard *for* loops.

Use a *while* loop, or the related *dowhile* loop, when there is no variable that serves as an index variable for the loop. A *while* loop requires of the programmer less discipline. For example, only a *while* loop should contain *break* or *continue* statements. These statements terminate all iterations or a single iteration of the loop. When one of these statements is used, it must be the only command on its line and must be followed by blank lines, so that it is easy to find. If a loop requires some of these early exit statements, try to keep the number to a minimum; they make programs much harder to understand and debug. The same warning applies to the use of the *return* statement to effect an early exit from a function.

Avoid the *goto* statement. The *C* ++ programming language has such powerful, undisciplined control structures already that it is difficult imagine a set of circumstances that truly requires a *goto* . If you believe that you have found such a circumstance, be prepared to defend your position against great skepticism. This subject still raises heated debate, and has ever since Dijkstra's famous article *GOTO considered harmful* . We agree with the article. (Perhaps someday C++ will incorporate Java's labelled break.)

Loop invariants should be stated in comments near the top of complex loops in the same style used for function header pre- and post-condition comments. A loop invariant is a statement about what can be assumed at the start of every iteration of the loop. If it is difficult to write the loop invariant for a particular loop, that loop is not likely to do what you intend.

Magic Numbers and Magic Types

A general principle of program style is that your code should mirror whatever it is modeling. Everybody agrees that a program dealing with cards should have a *class Card* , but beginners tend to use raw numbers like 50 to represent certain trick values in the midst of their code, and integer tags to represent suits (e.g. 3 to represent Hearts).

First, magic numbers: The presence of numeric or character constants in the body of a program is almost always confusing. The number 100 in a bridge-playing program could refer to the number of points for holding suit honors, for going down vulnerable undoubled, or for making a redoubled contract. Only by studying the program can one determine which meaning 100 has in a particular context. If the respective occurrences of 100 were replaced with *SUIT_HONOR_BONUS* , *VUL_UNDERTRICK_PENALTY* , and *REDOUBLE_INSULT_PENALTY* , then the meaning would be clear.

Program modification is also made simpler by using meaningful constant names instead of magic numbers. A few years ago, the value of *REDOUBLE_INSULT_PENALTY* changed from 50 to 100. In a well-styled program, it took modification of one line to update the program. In a poorly-styled one, it took several minutes of study to find which 50s should be changed to 100s. (The allowed magic numbers are 0 and 1.)

Related constants should be gathered together into *enum* types. An *enum* is most useful when a set of categories with unique identifying labels is required and the values of the labels is unimportant. For example, it doesn't matter which suits go with which integers in a bridge program, but hearts and spades should surely have different integer labels. An appropriate type definition might be

```
enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES};
```

All the advantages stated above for named constants over literal constants apply also to named types over primitive types. Even programmers who know not to use magic numbers in a program will use the *int* type to represent all kinds of unrelated quantities in a program, or use the *float* type to represent both temperatures and distances and weights. When the program needs to change to allow a higher precision representation for distances only, each float variable must be considered a candidate for conversion to double, and a process which should take a minute takes instead days. The simple *typedef* mechanism may be enough to define a type, but if a complex set of type conversion rules applies to the types of a program, the *class* mechanism of *C* ++ allows the legal type combinations to be specified. The *C* ++ *inline* specifier allows these strongly type-checked operations to be as efficient as the less secure operations they replace.

The overloading of the *int* type can make programs quite unclear. It is possible to use an *int* as an array of bits to represent subsets of some small set, with shifting, anding and oring to test for set membership, add a member to a set, or remove a member from a set. Such techniques are clever and efficient and perfectly acceptable. However, it should be made clear that the entity being manipulated is a set, and is not supposed to represent an integer. Use typedefs and inline functions definitions to define the set types and set operations. Similarly, an *int* is often used to provide a unique label for some object. In such applications, labels are only compared and assigned, not added or multiplied, and the program would be clearer if the labeling variables were of type *ObjectLabel* rather than *int* . It goes without saying that you should use *bool* s *true* and *false* rather than the deplorable C-style conflation with *int* s 1 and 0; you should use *NULL* rather than the *int* 0 when discussing pointers.

Identifier Names

The *C* ++ programming language places no restrictions on the length of identifier names, allows underscores in names, and distinguishes between upper- and lower-case letters in names. Using this capability to the greatest possible extent could easily make programs iLleGIblE, so use it instead to compose easily read identifierNames. The names given to functions that return *void* should generally be verb phrases, since these are functions that *do something* . Types are analogous to nouns, so their names should usually be nouns. Boolean variables and functions are statements of fact, so their names should be brief assertions. Other variables, constants, and functions that return values are like proper nouns or adjective-noun phrases. In the following program fragment, it is clear which names refer to types, which to variables, and which to procedures:

```
Point rayEndPoint, windowCenter;

Window graphicsWindow;

if (! graphicsWindow.contains(rayEndPoint))

rayEndPoint =

graphicsWindow.segmentIntersection(windowCenter, rayEndPoint);

graphicsWindow.drawFrom(windowCenter);

graphicsWindow.drawTo(rayEndPoint);
```

There are capitalization conventions for each nameable entity:

- Constants use all capital letters, with component words separated by underscores.
- Types use capital letters in the first position of each component word and nowhere else.
- Variables use capital letters in the first position of each component word but the first, and nowhere else.
- Optional: procedures, *i.e.* void-returning functions, use lowercase letters with component words separated by underscores.

These rules are unavoidably violated for some built-in types like *int* and *char* , and conventionally violated for a few other common types like *bool* .

It is obviously best to choose names that describe the purpose or meaning of the entity being named. Sometimes, this can be difficult. For example, in a function that computes the length of a String, there is a temptation to call the formal parameter of the function *string* . It is a bad idea to have two names in a program that differ only in capitalization, so if there are no meaningful alternative names, a name like *theString* might do.

Very short names are permissible in some contexts. The names *i* , *j* , and *k* are traditionally used as utility variables for loop indexing if no better name suggests itself. In code that implements an abstract mathematical algorithm, names like *x* and *y* might be appropriate. For example, in a program to solve quadratic equations, you probably *should* name the coefficients *a* , *b* , and *c* , and the solutions *x1* and *x2* . The discriminant, however, should not be named *d* ; using *disc* is better, and *discriminant* is best, since no confusion can now arise.

Never choose a name solely because it reduces the number of characters you have to type. Don't overly abbreviate words; for example, *numberOfSpades* is a better variable name than *nbrSpds* . Certain terms that have become part of programming jargon, like *flag* , *mask* , *array* , *temp* , *count* , *index* , and *pointer* can be used as components of names, but not as entire variable names. Such words describe implementation details, not the meaning of an entity of the program. Some of these notions have conventional abbreviations, so *spadePtr* is an acceptable abbreviation of *spadePointer* and *nSpades* is a reasonable substitute for *numberOfSpades* . Use such abbreviations consistently; a program should not contain variables called *nSpades* , *numHearts* and *clubCounter* , for example.

Variable Declarations

Variables that are used in only a small part of the program should be declared close to the first place they are used. It should not be necessary to search back through the program to find the declaration and explanatory comment for a variable unless that variable has been used in several parts of the program. Constants and types should be defined near the start of the program; if they are well named, it should be unnecessary to look back to the top of the file for clarification, and it makes changing the program easier when some of the constant values change.

Comments should describe the role of variables being declared. This should be useful information in addition to the meaningful identifier which should be used to name the variable. In fact, the best comment is usually a precise definition of the value of the variable; in its purest form, this would be a data invariant, but less formal comments may be just as helpful. This comment should appear to the right of the identifier. Such comments should be left-justified for easy reading. Utility variables used only as loop indices and subscripts may not require commenting if none would be helpful. Variable declarations should be grouped, not by attribute, but rather by their logical use within the program. If this is done, one comment may be sufficient for describing the use of several variables. For example:

```
float rainfall[50]; // Rainfall[0..numDays-1] is

int numDays; // table of daily precipitation.

int maxRainDay; // Rainfall[maxRainDay] is

// max of rainfall[1..numDays].

float avgRain; // Mean of rainfall[0..numDays-1].
```

Comments are not needed for the declaration of the parameters of a function. The use of the parameters should be adequately described in the comment for the function.

Machine-Level Programming

Although *C* ++ has some features of high level languages, it offers access to some machine operations at a very low level. While these operations are necessary for writers of device drivers1, they can greatly obscure the meaning of programs. Here are some common C tricks:

- *An int is just a bit pattern.* Most computers represent data in a binary form, and most number types use a two's-complement representation. Therefore, shifting a number using the C >> or << operators is the same as halving or doubling it. However, a good compiler will generate the same machine instructions for " *number >>= 1* " as for " *number /= 2* ", and the latter can be understood without knowing how computers work. (Program high-level, not machine-level.)

Similarly, as discussed above, an *int* can be treated as an array of bits. Such uses can be unclear and create unforeseen machine dependencies, so other type names should be used. Operations on *int* s should not rely on certain properties of the representation of *int* s.

- *TRUE and 1 are really the same.* An unfortunate decision by the designers of C allowed expressions like

```
trickScore = 20 + (bidSuit > 2) * 10;
```

which suggests a useless multiplication operation, to replace

```
trickScore = (bidSuit > 2) ? 20 : 30;
```

or, even better,

```
trickScore = (bidSuit > DIAMONDS) ? MAJOR_TRICK_SCORE : MINOR_TRICK_SCORE;
```

(N.B. The first two versions actually contain a bug: the magic number 2 corresponds to *HEARTS* , not *DIAMONDS* . An unintended error written by an experienced bridge player and C programmer!) There is no reason to multiply by truth values, and rarely any reason to perform arithmetic on them. If you do want to add or multiply with *TRUE* and *FALSE* , explicitly cast them to the type you intend.

- *Some machine instructions can change two variables at once.* The combination of a pointer dereference and a pointer assignment is a single operation on some computers, and *C* programs are full of statements like *a = b[i++]* and **--stackTop = item* . They have become *C* idiom. Nonetheless, it is clearer to do one thing per line: *a = b[i]; i++;* and *--stackTop; *stackTop = item;* both avoid bugs involving pre-and post-decrement, and any compiler produces equally efficient target code. Never use the increment and decrement operators as a sub-part of a statement.
- *If you can point to it, you can change it.* Pointer types are necessary for the construction of sophisticated data structures, but pointers can also be considered harmful. Non- *const* pointers should not be made to point to local variables, and non- *const* reference variables should not be made to reference local variables. It can be most disconcerting to have the value of a local variable change mysteriously not as the result of an assignment to it, but as the result of an assignment to a variable that knows the address of the local variable. Optimizing compilers can't do reasonable

flow analysis on such programs. The use of const pointers and const references to read one value under several different names is occasionally acceptable.

Comments and Blank Lines

Your goal is to produce perfectly clear, readable programs, and if this ideal could be met, no comments would be needed. In other words, you should attempt to minimize the need for comments by writing clear programs. However, this goal can seldom be reached, partly due to deficiencies in *C* ++.

Besides the comments mentioned above, you will need *statement comments* and *implementation comments* . A statement comment is meant to read just like a statement in the program, but in a higher level language than *C* ++. Statement comments should be placed in the program to describe logical units. They should begin with a verb, and describe what that program segment is doing, not how it is done. Comment groups of statements, not individual statements whose meaning is clear. The *C* ++ statements that refine a particular statement comment should appear indented beneath that comment in a block delimited by braces. The block should be indented as if the statement comment were the heading of a *for* or *while* loop.

Obscure or unusual statements should be avoided, but when necessary, an implementation comment should be used for clarification. These comments should appear to the right side of the statements they describe.

For example:

```
cents = (cents+50) / 100 * 100; // Round cents to nearest dollar.
```

Do not comment that which is already clear. For example:

```
cout << grossSales << '\n'; // Print the gross sales amount.
```

It is very possible to obscure a program by over-commenting. More is not necessarily better. Assume the reader of the program knows *C* ++ at least as well as you. (Comments should not be used to explain how *C* ++ works.) Note how the comments start capitalized and end with punctuation; this is helpful especially to distinguish multi-line comments from several one-line comments. (Using a sentence fragment is fine.)

The comments must agree with the program: incorrect comments are worse than none at all, because they lead the reader into a false sense of security.

Blank lines should separate logical sections of the program. Since such sections are generally preceded by a comment, one blank line should precede this comment. Blank lines should not be used randomly or excessively. Use them to improve the program's appearance and readability, and as visual logical separators. Optional: If you don't want your comment crowding your code by being directly above or below the pertinent code, you can have a blank commented line to link the comment with its antecedent. See the use of pre/post condition comments in the next section.

Because terminals and paper are sometimes limited to 80 columns width, program lines should not exceed 80 columns.

Indentation

Consistent and logical indentation, or paragraphing, is an extremely important aid in clarifying the structure of a program. The choices in an *if* statement and the bodies of loops become visible at a glance, which can greatly help the reader. There are many different good ways to indent, and most rules are relatively easy to learn, as are these. In a plain text editor, indenting can make editing slightly more difficult, but the benefits in program clarity outweigh the cost. In an editor like *GNU* Emacs, automatic indentation modes make possible the quick reformatting of an entire program. The default emacs C++ mode provides one widely-accepted indentation style. The style shown below can be achieved by some slight modifications[2]. Although you could enter your programs with haphazard indentation and reformat later, using the automatic indentation features can locate missing braces and semicolons before compilation errors occur. Even in writing programs on paper, indent consistently, as it is no more difficult, and it will help you to understand your program.

The general rule of indentation is simple: if one statement is logically a sub-part of another, it is indented (say) two spaces from its containing statement. For example:

// FunctionIdentifier

```
// preconditions

// postconditions

//

returnType FunctionIdentifier(argtype arg) {

const <constant-declarations>;

<local variables>;

// Aligned comments

// for local variables.

<body of function>

}
```

Consecutive statements executed sequentially should start in the same column, one above the other. Some people prefer to have matching braces occur at the same level of the construct they work for (in the case of the above function, both aligned beneath the "r" of "returnType"). The rationale for the alternative brace-positioning shown here is that the next text at the same indentation of (say) the function delcaration is the next statement/declaration. You may use either the default emacs style or this style, or any reasonable consistent hybrid. A more detailed example:

```
// comment describing action of following statements

statement1;

statement2;

// Comment describing action of following while.

//

while (condition) {

// Loop invariant, if present, would go here.

statementA;

statementB;

}
```

An *if* statement without an *else* clause (conditional execution) is written:

```
if (cond) {

statement1;

statement2;

}
```

An *if* statement with an *else* clause (selection of two alternatives) is written:

```
if (cond) {

statement1;

statement2;

}

else {

statementA;

statementB;

}
```

An *if* statement with *else if* clauses (selection of many alternatives) may be written (although the use of a *switch* statement should be investigated to see if it is appropriate) as follows:

```
if (cond1)
```

```
statement1;
else if (cond2) {
statement2a;
statement2b;
}
else if (cond3)
statement3;
else if (condY)
statementY;
else
statementZ;
```

The *switch* statement is quirky, both in its *C* ++ definition and in its default *GNU* Emacs indentation. We recommend the following:

```
switch (expression) {
case 'a':
statement;
break;
case 'b':
statement;
break;
default:
break;
}
```

There should be no more than one statement per line, with possible exception of short *related* assignment statements. If a statement is longer than one line, the continuation lines should be indented at least two spaces, and the statement should be broken in logical places, not just when you reach the end of the line. Even better is to align related parts of complex statements to make error detection and pattern matching easier for you. Compare the ease with which an error is detected in these two lines:

```
triangle(a*x1*x1+b*x1+c, a*x2*x2+b*x2+c, a*x3*x3+c*x3+c);
```

```
triangle( a*x1*x1 + b*x1 + c,
```

a*x2*x2 + b*x2 + c,

a*x3*x3 + c*x3 + c);

Observe how the use of spaces helps group related parts of the expression. Examine the sample programs given out in our classes to see how these indentation rules are used in real programs. If we fail to follow these conventions on handouts, call it to our attention; we're not perfect. Note also that there are many other sets of good indenting rules, and that these rules are somewhat different from those in the text.

Efficiency

Computers are very fast and have very large memories, but they are far from being infinitely fast or large, and it is not difficult (with only mild effort) to produce programs that would overwhelm even the most sophisticated modern computer. The best rule-of-thumb is to first program for correctness, following the most straightforward algorithm which you might follow were you doing the program by hand. A well-decomposed problem usually leads to not-inefficient solutions. After you have a correct program, if you run it and realize that speed is a problem, then go back and look for ways to increase efficiency.

Consider writing a program to calculate tails of binomial distributions:

$$Tail(n,m) = \sum_{k=m}^{n} \binom{n}{k} p^k (1-p)^{n-k}$$

Calling n-choose-k involves three calls to factorial (each of which is a loop); there is a lot of cancellation; each call to n-choose-k and p^k and $(1-p)^{n-k}$ could re-use results for the previous k. Clearly, a lot of optimization can be made. Moreover, this was using Scheme's bignum package for n=100, so each call to n-choose-k involves dividing numbers of well more than 100 digits. Myself, I was anxiously ready to start writing the fancy code for this. But since I always advise others to write the simple implementation first, I steeled myself to first whip out a correct version (to help debug my later, fancy version). So the first implementation re-calculated factorials, immediately cancelled most of the numbers just multiplied, and repeatedly found large powers of p. The result? This most simple-minded approach possible ran in about a second; I'd been wasting my time even thinking about how macho my code could be. On the other hand, if I had needed to compute this for n=1000, the original code would have taken to long, and optimizing would have been in order.

There are two general guidelines to follow when coding for efficiency. First, focus on code which is executed a lot. Compare function f (which accounts for 1% of the program's runtime) and a function g (which accounts for 30% of the program's runtime). Would you rather spend two days getting f to run 10 times faster, or a couple of hours getting g to run twice as fast?

Second, before you work on tweaking individual lines to get (say) a constant factor speed-up, concentrate on using a different algorithm with a better big-Oh time or space efficiency. Replacing an $O(n^3)$ algorithm with an $O(n^2)$ algorithm is going to be a better win than getting the original algorithm to make only half as many memory references. Also, a half-decent optimizing compiler can often do at least as well as a human in twiddling individual lines for increased performance, cache utilization, etc.

Some examples: To compute the sum of the squares of the elements in an array, do not create a new array for the squares before adding. To find the largest element in an array which has been sorted ascending, do not examine each element; simply select the last one. Unnecessary arrays, especially large and/or multi-dimensioned ones, can quickly use up all available memory space. Unnecessary loops, especially when nested, can waste an enormous amount of time. Complicated expressions which do not change should be moved out of loops, if possible, so that they are computed only once.

Summary

Any programmer who fails to comply with the standard naming, formatting or commenting conventions should be shot. If it so happens that it is inconvenient to shoot him, then he is to be politely requested to recode his program in adherence to the above standard

--Michael Spier, The Typset-10 Codex Programaticus,

Digital Equipment Corporation, Maynard, Mass., 1974

A good program reads like a good book. The same principles you use in writing a paper for an English course should be applied to writing a program. In both cases your purpose in writing is to communicate your ideas. Clarity of organization, paragraphing, and attention to details of spelling and punctuation all contribute greatly to the quality of the finished product, independently of the quality of the ideas behind it.

---

1. When was the last time you met somebody who has ever needed to write a device driver?

2. To customize emacs to the style shown here, use "M-x c-set-offset" to set variables such as "block-close" and "class-close" to the value "+". (Do "M-x info" for full emacs documentation.) You can automate this by adding to your ".emacs":

```
;; Customize my C++ indentation.

;;

(add-hook 'c++-mode-hook

'(lambda ()

(c-set-offset 'access-label 0) ; Private/public labels.

(c-set-offset 'block-close '+)

(c-set-offset 'defun-close '+)

(c-set-offset 'class-close '+)
```

(c-set-offset 'inline-close '+)

(c-set-offset 'brace-list-close '+)))