Programming With Style

Theo Norvell. MUN

1 The point of programming style

A computer program has many purposes. One is, of course, to instruct a computer as to what it should be doing. But more importantly a computer program communicates to other programmers what is going on. Here is some of what Turing Award winner Donald Knuth has to say in his essay "Why I must write readable programs."

> Computer programs are fun to write, and well-written computer programs are fun to read . One of life's greatest pleasures can be the composition of a computer program that you know will be a pleasure for other people to read, and for yourself to read.
>
> Computer programs can also do useful work. One of life's greatest sources of satisfaction is the knowledge that something you have created is contributing to the progress of welfare of society.
>
> Some people even get paid for writing computer programs! Programming can therefore be triply rewarding—on aesthetic, humanitarian, and economic grounds.
>
> ...
>
> The computer programs that are truly beautiful, useful, and profitable must be readable by people. So we ought to address them to people, not to machines. All of the major problems associated with computer programming—issues of reliability, portability, learnability, maintainability, and efficiency—are ameliorated when programs and their dialogs with users become more literate.

So the main point of programming style is to make your programs easily readable. There are many ways to accomplish this. I will discuss just a few:

- Comments
- Naming
- Layout
- Simplicity

1.1 Rules

The fundamental rule is the Golden Rule of programming:

> *Write your programs and documentation as you would wish that others would write programs and documentation for you.*

In this course your programs will be marked on style as well as correctness and other technical aspects.

The rest of this paper contains some guidelines for good style.

2 Commendable Comments

Comments are used for many purposes. You should address them especially to whomever may be modifying the code.

It is very seldom that a student loses marks for overcommenting or that an employee is criticized by colleagues for being too verbose.

At their best comments form a bridge between code on the page and the abstractions — procedural and data— that exist in the mind of the programmer. To understand the code in the same way as the original programmer did, the reader needs access to these abstractions.

Commenting is not only useful for future readers, but it also helps you, the original programmer, to understand better what you are doing. If you can not explain why it is right, it is probably wrong. Thus good commenting leads to programs that are more likely correct.

2.1 Top of file comments

It is a good idea to have a block of comments at the top of each file containing its name, explaining who wrote the file, when, and what system it is a part of. Furthermore as files are modified, a note should be made of what was changed, when, and by whom.

```
// File func.tpl
// Member functions for the 'func' class.
// Author: T.S. Norvell
// Date: 1996 May 15
// Modified: 1996 June 2 — TSN — Added member function 'ismapped'.
```

2.2 Class comments

Associated with each class (or other data type) should be an explanation of what is being represented by the class.

```
// func — A class for representing finite functions.
template <class Domain, class Range>
class func {
        ...
}
```

2.3 Subroutine header comments

In C/C++, subroutines are called "functions". In this document, I'll use the term "subroutine" as it is less language specific.

2.3.1 Explain what, not how.

Associated with each subroutine declaration should be a short description of *what* the subroutine does.

```
/* natRoot( i ) — Returns the natural square root of a number. The natural square root is the
largest natural number whose square is less than or equal to i. */
int natRoot( int i ) ;
```

Each subroutine should represent a *procedural abstraction*; the comments should explain that abstraction.

Usually you should not write comments that attempt to explain *what* the subroutine does by explaining *how* it does it. For example, this is a poor comment:

```
/* natRoot( i ) — Starts at zero and increments until the number squared is greater or equal to zero
and returns that.*/
int natRoot( int i ) ;
```

You *should* explain how it works, but not here; this will be dealt with under the heading 'Subroutine body comments'.

2.3.2 Formal Comments.

Subroutines A very effective way to explain what a subroutine does is to write a precondition and a postcondition.

- A precondition documents the assumptions under which the subroutine was written.
- A postcondition documents the behaviour of the subroutine.

These do not have to be completely formal, but the more formal you make them the less likely they will be misunderstood.

```
/* natRoot( i ) — Returns the natural square root of a number. The natural square root is the
largest natural number whose square is less than or equal to i.
      pre: i >_ 0
      post: Result == floor( sqroot(i) ) */
int natRoot( int i ) ;
```

As illustrated above, I use the word 'Result' in the postcondition to refer to the value returned by a subroutine.

In postconditions, I use undecorated variable names (a, b, c) to refer to the initial values of variables and primed variables names (a', b', c') to refer to the final values of variables.

```
/* swap( i, j ) — Swap A[i] with A[j], if necessary, to make A[i] the smaller.
        pre : 0 _< i,j < N
        post: A'[i] _< A'[j]
            and (A'[i]==A[i] and A'[j]==A[j] or A'[i]==A[j] and A'[j]==A[i])
        may change: A[i], A[j]. */
void swap( int i, j) ;
```

If the subroutine may change a variable, this too should documented. This is what the 'may change' clause is for.

Classes and Methods Formal documentation of a class begins with a description of the abstract fields of the class, that is the values being represented by the objects of the class.

```
// func — A class for representing finite functions.
// Abstract fields:
        // F : a finite function.
template <class Domain, class Range>
class func {
        ...
}
```

For public methods of a class, the comment should be in terms of the abstract fields.

```
/* map(a, b) — Maps 'a' to 'b'.
        Post: (for all i in the domain of F, i != a => F'(i) == F(i)) and F'(a)==b
        May fail. */
```

For some classes, the representation may fail at some point, for example if space runs out. I put a special comment to this effect unless the method of indicating failure can be easily described in the postcondition.

2.3.3 Where to comment if there is a choice.

In C and C++ it is often the case that a subroutine (i.e. function) is *declared* at one point and *defined* (i.e. its body is given) at another. I would put the comments that describe what the subroutine does with the declaration. For member functions of a class, this will be within the 'class' declaration.

2.4 Representation comments.

In a class there should be comments explaining how the abstract values are represented by the data members of the class. The best place for this sort of comment is at the start of the 'private' section of the class declaration.

It is hard to adequately stress the importance of this sort of comment enough. They should explain which of the potential values for the data fields are valid. And for each valid value, they should explain what abstract value is represented.

Here is part of a class representing character strings;[1] two constructors and an operator are declared:

```
class String {
    public:
        String(char *p) ;
        String(String &str) ;
        String operator+=(String &a) ; ...
    private:
        int len_ ;
        char *s_ ; }
```

```
String::String( char *p) {
    len_ = strlen(str) ;
    s_ = new char[len_ + 1] ;
    strcpy(s_, p) ;
}

String::String( String &str ) {
    len_ = str.len_ ;
    s_ = new char[ len_ ] ;
    strcpy(s_, str.s_) ;
}
```

Apparently the programmer forgot, at some point, whether the len field represents the length of the string or the number of bytes allocated to hold it. There is a difference because the C library subroutines (strcpy and strlen) use the convention that a string is represented by a sequence of bytes followed by a 0 byte. An extra byte must be allocated to hold the 0 byte. Either representation is equally reasonable, but it is not reasonable to use one representation in one constructor and the other in the other constructor. If the programmer had documented which representation is being used in this class, the chance of getting it wrong would be much less.

Here is a 'catentation' operator for the class:

[1] This example is adapted from Cargill's book *C++ Programming Style*.

```
String String::operator+=(String &a) {
        char *temp = s_ ;
        len_ += a.len_ ;
        s_ = new char[ len_ ] ;
        strcpy(s_, temp) ;
        strcat(s_, a.s_) ;
        delete temp ;
}
```

Here the programmer has written code that makes sense for neither representation. For one, it allocates one too few bytes, and, for the other, one too many! Again clear documentation of the representation can only help.

Here we document one possible representation (len represents the number of characters in the string).

```
class String {
        ...
        private:
        /* Representation:
        s_ — points to the first element of an array of chars allocated by and owned by this object.
        len_ — (the number of bytes pointed to by s_) - 1.
        The array's end is marked by a 0 byte, so s_[len_] = 0 and s_[i] != 0 for i < len_.
        The string being represented is s_[0], s_[1], ..., s_[len_ - 1]. */
            int len_ ;
            char *s_ ; } ;
```

The comment about owning the sequence of bytes makes it clear that this allocated array must be allocated and deleted by the member functions of each string object.

2.5 Variable comments.

Generally each variable except the most short-lived should have a comment explaining its role. Relationships that hold between variables throughout their lifetime should be explained.

2.6 Subroutine body comments.

In many ways these are the least useful kind of comment. They seldom tell the reader anything they could not figure out from reading a few lines of code. A reader should be able to learn *what* the subroutine does from reading the subroutine header comments. If they really want to know *how* the subroutine does it, there is seldom a substitute for reading the code itself. Nevertheless, comments in the subroutine body may save the reader some time, especially when algorithm is cluttered up by error recovery code and things like that. For tricky algorithms, it may pay to outline the entire algorithm in pseudo-code at the start of the subroutine body.

One problem with subroutine body comments is that it can be unclear which lines they refer to.

```
//Fill up the new node.
newNodeP->datum = newData ;
newNodeP->flink = *linkP ;
*linkP = newNodeP ;
```

Here the comment refers to the next two lines only, the third is uncommented. But this is only clear after reading the code. I prefer to indent the lines to which a comment refers:

```
//Fill up the new node.
    newNodeP->datum = newData ;
    newNodeP->flink = *linkP ;
*linkP = newNodeP ;
```

A reader who understands the comment can quickly skip over the code it explains. This is a case of procedural abstraction. The details of how some task is accomplished are there if the reader wishes to read them, but, if the reader understands *what* the code does from the comment, there is often no need for him to look at *how* the code does it.

An alternative convention is to leave a blank line before and after each commented block.

One kind of comment that can be very helpful in the body of a subroutine documents the invariant of any loops[2]. This is also one of kind of comment least often sighted in the wild!

2.7 Rules

Use comments at the top of each file to identify your work. Use comments to explain what each subroutine does. Use comments to explain data representations. Use comments to explain loop invariants and tricky parts of the code. Write informal comments and augment them with formal comments where possible.

3 The name game

3.1 Avoid lame names.

The names you pick for entities such as types, variables, and subroutines can make a tremendous difference to the readability of your code. There are many possible conventions. Within one program try to be consistent. E.g. do not write some multiword names as copy_data_file and others as processTransactionFile.

Very short meaningless names should be used only for variables with a very small scope. A typical bug I recently saw a student create involved using 'i' as the name of for-loop variables in both an inner and an outer loop. Had he used terminalNumber in the large outer loop, the bug would have been avoided, so would several minutes of frustration, and the program would have been more readable.

[2] An "invariant" is a statement of something that must be true at the start of each iteration of a loop and also when the loop is exited from. The invariant should be relevant to showing the loop accomplishes its role in the program.

Singular nouns should be used for variables and types. Plural nouns might be used for types or arrays, but often singular nouns are best.

Verbs should be used for subroutines that do not return values (in C/C++ these have a 'return type' of void).

For subroutines that return values, nouns are best. E.g. natRoot rather than computeNatRoot.

Some people use very distinctive codes for names of different sorts. For example, they will always write type names with a capital initial letter. Or they will always begin or end data member names with an underscore.

Consistency is important especially in larger programs. If you call the maximum number of elements in one array sizeFoo, then call the maximum number of active elements in another sizeBar; and don't use sizeBaff to refer to current number of elements in use in some other array.

3.2 Name any constants.

Every program should be designed with change in mind. One way to make your programs more easily changeable is to give a name to any numbers (or strings, or other values) that might appear repeatedly in the code. For example.

```
int someArray[1000] ;
...
for( int i = 0 ; i < 1000 ; ++ i ) { ... }
if( i == 1000 ) { ... }
```

If needs change later, and someArray must be bigger, someone must hunt down all the 1000s and see if they need to be changed. Better would be

```
const int someArraySize = 1000 ;
// In C, one would write #define someArraySize 1000
int someArray[someArraySize] ;
...
for( int i = 0 ; i < someArraySize ; ++ i ) { ... }
if( i == someArraySize ) { ... }
```

Not only is this more changeable, it is more readable,

Furthermore, constants aren't always that easy to find. A text processing program I read once was written assuming there are 26 letters. This assumption appeared in the code not only as the number 26, but also as the numbers 27 and 52. If the program were ported to work with Swedish, one would have to go through it with a fine toothed comb, examining every number to see if it were affected. It would have been better to have a constant named NumLetters.

3.3 Rules

Name constants. Pick meaningful names. Be consistent.

4 Layout — Braces and Spaces

Layout refers to the way your code is positioned on the page. The one rule everyone agrees on is that code should be consistently and properly indented. Statements within each block of statements should be indented relative to the enclosing block and lined up with each other. How many spaces code should be indented does not matter much. Anywhere from 2 to 8 spaces is reasonable.

One good rule for braces in C/C++ is that brace pairs should line up either horizontally or vertically.

```
if( some condition ) { duh-de-dah ; }
else
{
        do this
        and that
        and the other
}
```

I prefer to place all braces off to the right like this

```
if( some condition ) {
        duh-de-dah ; }
else {
          do this
          and that
          and the other }
```

because it saves vertical space.

While on the subject of braces. I generally use them, even if they enclose only a single statement. This makes editing easier and makes it impossible to make this mistake:

```
if( A )
        for(int i=0 ; i<N ; ++i)
              if( B) this
else that
```

From the indenting, it appears that the programmer intended *that* to happen when A is false. In fact *that* will happen in every iteration of the loop where B is false. Using braces

```
if( A ) {
        for(int i=0 ; i<N ; ++i) {
              if( B) this } }
else that
```

fixes the problem. Two notes on this problem with C++'s syntax: First, the first language to have this problem was Algol, invented in 1957. Many languages invented since then have deliberately — and easily— avoided the problem. But languages keep being invented that copy the same mistake. Java, invented in 1991, 34 years after Algol, is a recent example. Second, in case you think this won't happen to you, a programmer who had been programming for 30 years recently asked me to help debug a program. The problem with his code turned out to be a variation on the last example.

Another way that layout can improve the readability of your code is if you put blank lines to separate less related parts of your algorithm. Consider this bit twiddling algorithm (which leaves something to be desired from a commenting point of view):

```
//Compute the first bit of the root into s.
      int s = 1, ss = 1;
      /*Invariant: ss is s squared */
      while( 4*ss <= n) {
           s = 2*s ; ss = 4*ss ; }

//Compute the remaining bits of the root into r.
      int r = s, rr = ss, rs = ss ;
      /* Invariant: rr is r squared, rs is r*s */
      while( s != 1 ) {
           s = s/2 ; ss = ss/4 ; rs = rs/2 ;
           if( rr+2*rs+ss <= n ){
                r += s ; rr += 2*rs+ss ; rs += ss ; } }

return r ;
```

The two loops are conceptually different phases of the algorithm and should be physically separated. The return is not related more closely to the last loop, so it is also separated. The declarations, on the other hand, are closely related to their respective loops and so are not separated from them.

Conversely, it is *occasionally* reasonable to write closely related parts of an algorithm on one line. This is particularly true when a series of assignments are conceptually one assignment. E.g. in the last algorithm whenever s or r changes, the changes to their squares and the product of the r and s are conceptually a part of the same 'multiple' assignment. Also, while I usually accord each variable a line of its own when it is declared, I felt that each of the two sets of variables used in this algorithm are conceptually more strongly linked than usual.

Horizontal space can be used to reinforce the precedence of operators:

```
b = 2*x + y > z && q;
```

rather than

```
b=2 * x+y > z&&q;
```

Better yet, redundant parentheses can help:

```
b = ((2*x + y) > z) && q ;
```

4.1 Rules

Use layout to enhance the readability of your software.

5 Simplicity itself

Simplicity is by far the most important element of style. If there are two ways to accomplish the same task with acceptable efficiency, you should always pick the simplest. Simplicity is also the hardest to teach.

To further complicate simplicity, what is simple for some may be complex for others, who have not learned the simple way to do things. The use of comments to explain a simple but nonobvious algorithm or representation will delight the reader, who will not only learn how your program works, but also a new and simple technique.

Here are a few guidelines. Many more can be found in the references given at the end of this article.

5.1 Do not have special cases without good reason.

The world of humans is full enough of special cases. For example the floor below floor 1 in most buildings is not floor 0. (The only building I have noticed that has a floor -1 houses the Programming Research Group at Oxford.) This complicates the elevator's software. Do not complicate your software by making special cases where none need exist.

Here is an algorithm to change the direction of the links in a singly linked list:

```
Node *newHead ;
if( head == 0 ) {
        newHead = 0 ; }
else {
        newHead = head ;
        head = head->next ;
        newHead->next = 0 ;
        /* Invariant: The reverse of newHead's list catenated with head's list
        == the original value of head's list. */
        while( head != 0 ) {
                const Node *temp = head->next ;
                head->next = newHead ;
                newHead = head ;
                head = temp ; } }
```

The programmer has observed that, in a nonempty list, the 'next' field of the first node is assigned the null pointer, whereas the 'next' field of subsequent nodes are not. Therefore they made a special case of the first node! This requires three assignments to handle the special case and to establish the invariant of the loop. Worse yet, it also makes a special case out of the empty list, which has no first node to make a special case out of! This requires duplication of the assignment to newHead and of the test of head against 0.

A simpler solution is also much clearer. It begins with the realization that the invariant of the loop is established by making the list represented by newHead empty:

```
Node *newHead = 0;
/* Invariant: The reverse of newHead's list catenated with head's list
== the original value of head's list. */
while( head != 0 ) {
      const Node *temp = head->next ;
      head->next = newHead ;
      newHead = head ;
      head = temp ; } }
```

This solution is 6 lines rather than 12. The reader can concentrate their efforts on the heart of the algorithm represented by the loop body.

5.2 Do not omit special cases

(This is more a matter of correctness than style, but after the last section, I thought I should not let it slip.)

Einstein said, "Simplify as much as possible, but no more." For example the bit twiddling algorithm presented above is to compute the natural root of a number. But this algorithm does not work for 0. So we must make a special case of 0, or find another algorithm. Both when programming and when testing, you should try to think of the most unusual inputs to your algorithms (that still satisfy the precondition, if any).

5.3 Use variables with care.

5.3.1 Use maximally local variables

Keeping track of what the variables are representing is one of the harder parts of reading a program. Thus one of the best things you can do for your reader is to minimize the average number of variables that exist throughout the program. For example, in the list reversal program, the variable called temp does not exist before the loop or after the loop. The reader can clearly see this and does not have to worry about what its value represents outside of the four lines of the loop body. Also by declaring it within the loop, I can declare that it is a constant, not really a variable at all. In fact the four lines of the loop body conceptually represent a single multiple assignment

$$\begin{bmatrix} \text{head->next} \\ \text{head} \\ \text{newHead} \end{bmatrix} \leftarrow \begin{bmatrix} \text{newhead} \\ \text{head->next} \\ \text{head} \end{bmatrix}$$

So temp is merely an artifact of the translation of this into C.

Similarly in the bit twiddling example, the variables r, rr, and rs do not exist during the first loop and thus can not interfere with the reader's understanding of it.

On a larger scale, variables that are declared outside of all subroutines and classes represent the worst extreme. They exist throughout the whole program and thus concern the reader all the time. The best solution is not to declare such variables. If you must declare them, limit their scope to one file by declaring them as static.

5.3.2 Initialize immediately

Of course you should (almost) never use a variable before it has been assigned a value. If you follow the last point of advice —use maximally local variables— you will (almost) never declare a variable until you are ready to assign it its first value. Thus you will never have to worry about using uninitialized variables.

Consider this snippet of code, intended to find the first occurrence of a string pat in a string targ (strings are represented by character pointers, with a 0 byte marking the end of the string).

```
int locn ;
// Check each position, t, in targ.
for( int t=0 ; targ[t] != 0 ; ++t ) {
      // Check if pat occurs at position t in targ.
      int p = 0
      while( pat[p] != 0 && targ[t+p] != 0 && pat[p]==targ[t+p]) {
            p += 1 ; }
      // Was pat found at position t ?
      if( pat[p] == 0 ) {
            locn = t ;
            break ; }
      else {
            locn = -1 ; } }
```

Here the programmer ignored the rule of initializing immediately. The price is that the code contains a subtle bug that is likely to be missed in testing: locn will not be assigned if the target string is empty. (Empty strings are represented by a pointer to a 0 byte.) Initializing locn with an appropriate value solves this problem. The obvious initialization is to -1, thus eliminating the need for the else clause.

```
int locn = -1 ;
// Check each position, t, in targ.
for( int t=0 ; targ[t] != 0 ; ++t ) {
      // Check if pat occurs at position t in targ.
      int p = 0 ;
      while( pat[p] != 0 && targ[t+p] != 0 && pat[p]==targ[t+p]) {
            p += 1 ; }
      // Was pat found at position t ?
      if( pat[p] == 0 ) {
            locn = t ;
            break ; } }
```

(Note, there is still a bug! Can you spot it?)

5.3.3 Don't give variables double duty

Every variable should represent one thing. The reader learns to associate the variable with its meaning and will be very disconcerted if later the variable is used to mean something else. Sometimes programmers will break this rule by frivolously initializing a variable to a value, only to overwrite it later. Again this guideline is harder to break if you follow the guideline of declaring variables as locally as possible.

5.4 Use modules and data abstraction

The use of modules[3] can greatly improve the readability of your code. It will also improve your code in many other ways. This is a big topic too big to even touch on in a short paper. But it is one of the main themes of the course.

5.5 Use subroutines

Subroutines can make your code shorter and thus quicker to read and to write. More importantly, they allow you to isolate and name important algorithms. They provide your reader with vocabulary with which to understand what you are doing.

5.5.1 Use subroutines for common tasks

Any time you find you are writing similar code again and again, make a subroutine of it. This is not only going to reduce your work, but it:

- Makes the code shorter to read.
- Makes debugging more effective. Since the code is written only once, you only have to debug it once.
- Gives a name to an important concept. The common task probably has some conceptual meaning. Making it a subroutine allows you to give it a meaningful name and provides an opportunity to document it.

5.5.2 Use subroutines to break up complex code

Even if a subroutine is only called from one spot, it can greatly simplify a complex algorithm by separating the 'how' of it from the 'what' and the 'why'. This is an application of procedural abstraction. There is no need for your reader to understand the details inside the subroutine, only the task that it accomplishes.

5.5.3 Ensure each subroutine has a clear mandate

A complex algorithm should not be broken arbitrarily into smaller subroutines. Each subroutine should have an easy to describe purpose — this constitutes its interface.

[3] In C++ classes are the most common form of module.

5.6 Use simple flow of control

5.6.1 Avoid nested 'if's where possible.

Consider this algorithm:

```
if( x >= y ) {
        if( y >= z ) {
                small = z ; }
        else {
                small = y ; } }
else {
        if( x >= z ) {
                small = z ; }
        else {
                small = x ; } }
```

What does it do? A few minutes of reading should convince you that it finds the smallest of three numbers. But wouldn't it have been clearer as:

```
if( x < y && x < z ) {
        small = x ; }
else if (y < z ) {
        small = y ; }
else {
        small = z ; }
```

The difference is not vast, but a number of small differences add up.

Better yet, use subroutines to tame control structures:

```
small = min(x, min(y,z)) ;
```

5.6.2 Do not replace logical operations with branches.

One use of if-statements that I find particularly abhorrent is typified by this:

```
if( A ) {
        if (B) { foo = true ; }
        else { foo = false ; } }
else { foo = true ; }
```

Such code shows a lack of knowledge of simple logic notation that is unacceptable for any professional programmer (or electrical or computer engineer). One can see that the code is equivalent to

```
foo = A && B || !A ;
```

and in turn (by an absorption law and the commutativity of 'or') to:

```
foo = !A || B ;
```

C does not have an implication operator, but we should recognize this expression as 'A implies B'.

5.6.3 Avoid side effects in expressions.

Side effects are assignments, I/O operations, and other changes to the program state or outside world. These generally should not occur in expressions. The purpose of an expression should be to compute a value. An example is

```
if( p = new Foo ) { ... }
```

I prefer

```
p = new Foo ;
if( p ) { ... }
```

which disentangles the two actions: allocating the variable and testing to see if the allocation was successful.

5.6.4 Avoid extraordinary exits from loops.

Loops are often the source of complication. The simplest sort of loop can be exited from only one place. Any time you find you are doing something more complex, it is worthwhile to ask yourself if it is necessary.

The pattern match code above is an example. It contains an extraordinary exit in the form of a break statement. This reflects the fact that there are two reasons to exit the main loop: *the pattern is found* and the *end of the target is reached*. But the complexity of the algorithm, that the break warns us of, hides another bug. If pat is an empty string and targ is too, then the locn should be set to 0, but it is set to -1! We could patch up the code, but rewriting the algorithm to merge the loops and eliminate the extraordinary exit is a clean way to fix this problem:

```
int t = 0 ; // index of target string.
int p = 0 ; // index of pattern string.
/* Inv: pat does not match targ at any index less than t and the first p items of pat match the p
items of targ that start at t. */
while( targ[t+p] != 0 && pat[p] != 0 ) {
        if( targ[t+p] == pat[p]) { p += 1 ; }
        else { p = 0 ; t += 1 ; } }
int locn ;
if( pat[p] == 0 ) { locn = t ; }
else { locn = -1 ; }
```

I should note that the break was not really the problem. We could easily have eliminated it without fixing the error. The problem was the extra complexity, of which the break was the symptom.

Breaks are sometimes perfectly legitimate. For example in pseudo-code I might write

```
while the file isn't empty
        read one thing
        do something with it
...
```

But, in C++ the way to tell if a file is empty is to try to read it and then see if you have failed.

```
while( true ) {
        file.get( c ) ;
        if( file.fail() ) break ;
        do something with c }
if( file.eof() ) {
        ... }
else {
        Deal with an unexpected failure }
```

5.6.5 Use gotos, breaks, returns and throws with the utmost of discretion.

Breaks are not the only kind of extraordinary exit. Goto statements and return statements that jump out of loops are also extraordinary exits. In fact, most uses of return, other than at the very end of subroutine, should be red flags that you may be making things more complex than necessary. Likewise *any* use of a goto statement should be a very big flashing red light that things may be out of control. This goes triply for goto's that cause backwards jumps, and jumps into control statements from the outside of them.[4]

[4] However I must confess that I often do use backwards goto's when programming in C or C++. This results from thinking in terms of recursive subroutines and then translating the recursive routines into C or C++. Style is a matter of culture and such usage tends to cause a culture clash when read by someone who isn't used to this

One use of return, that *is* acceptable, is to deal with exceptional cases in a subroutine. It is reasonable to write either

```
void make_node( Node *&p, bool &failed) {
      p = new Node ;
      if( ! p ) {
            failed = true ;
            return ; }
      failed = false ;
      ... }
```

or

```
void make_node( Node *&p, bool &failed) {
      p = new Node ;
      if( ! p ) {
            failed = true ; }
      else {
            failed = false ;
            ... } }
```

though I prefer the latter.

Exceptions are like gotos that can 'go to' some locations completely outside of the current subroutine invocation. Just as goto's cause confusion, so do exceptions. Often they cause more chaos than any goto could. There are reasons to use them in C++, but they are best used only as a last resort.

5.6.6 Let the control structure fit the algorithm

The above advice to avoid extrodinary exits can be seen as a special case of a more general principle:

Let the control structure fit the algorithm

If an algorithm logically consists of three sequential steps, write it as a sequence of three statements.

If an algorithm consists of a repetition of one operation over and over again, write a loop.

If an algorithm consists of a choice of two possible operations, write an if-else statement.

Every algorithm can be expressed using sequences, choices, and loops.

Conversely, don't use the above constructs when they are not appropriate. Avoid putting code inside a loop unless it is potentially executed more than once. Avoid putting code in choices unless there really is a choice.

Consider the following subroutine in C++. It attempts reports the distance from the first 'a' in a C-style string to the first 'b' in the same string. For example, given a string, "Rocking the Casbah", it returns 2. If either letter is missing, it returns -1:

---

way of thinking.

```
int a_b_dist(char *p) {
      for( int i=0 ; p[i] != 0 ; ++i ) {
            if( p[i] == 'a' ) {
                  for( int j=0 ; p[j] != 0 ; ++j ) {
                        if( p[j] == 'b' ) {
                              return j-i ; } }
                  return -1 ; } }
      return -1 ; }
```

This subroutine should work, but it obscures the actual structure of the algorithm by including within loops a lot of things that are done but once. It even has nested loops, which suggests a much more complex algorithm.

In fact three things happen sequentially, first an 'a' is searched for, then a 'b', then the distance is returned. The following way of writing the algorithm is far more suggestive of this simple structure:

```
int a_b_dist(char *p) {
      int i, j ;
      // Let i be the index of the first 'a'
            for( i=0 ; p[i] != 0 && p[i] != 'a' ; ++ i ) { }
      // Let j be the index of the first 'b'
            for( j=0 ; p[j] != 0 && p[j] != 'b' ; ++ j ) { }
      if( p[i] == 0 || p[j] == 0 ) {
            return -1 ; }
      else {
            return j-i ; } }
```

Here is one last example. We have an algorithm that has two phases, each repeated a number of times. We could express this as:

```
enum { phase1, phase2, done } state = phase1 ;
do {
      if( state == phase1 ) {
            do_phase1() ;
            if( phase1_done() ) state = phase2 ; }
      else {
            do_phase2() ;
            if( phase2_done() ) state = done ; }
} while( state != done ) ;
```

Within the loop, there appears to be a choice made every time. But this is not true. The choice to switch to phase2 is only made once. Once the algorithm starts on phase 2, it never goes

back to phase 1. A better reflection of the algorithm might be:

```
do {
      do_phase1() ;
} while( ! phase1_done() ) ;
do {
      do_phase2() ;
} while( ! phase2_done() ) ;
```

5.7 Rules

Make everything as simple as possible, but no simpler. Declare variables no sooner than you need to. Use modules and data abstraction. Use subroutines appropriately. Avoid complex control structures. Especially avoid complex loop structures.

6 Other sources of information.

There are several books that discuss programming style. One classic is *The Elements of Programming Style*, by Kernighan and Plauger. Although the languages it uses in examples (PL/1 and Fortran) are dated, most of the advice is not. A book that discusses issues particular to C++ is *C++ Programming Style*, by Tom Cargill. Many companies and programming groups have very specific rules for their programmers; an example is *Programming in C++: Rules and Recommendations*, by Henricson and Nyquist (available on the web at http://www.chris-lott.org/resources/cstyle/Ellemtel-rules.html); this guide contains 51 detailed rules and 79 recommendations such as: "A class which uses "new" to allocate instances managed by the class, must define an *assignment operator*."