Notes on Programming in C

Rob Pike

Introduction

Kernighan and Plauger's *The Elements of Programming Style* was an important and rightly influential book. But sometimes I feel its concise rules were taken as a cookbook approach to good style instead of the succinct expression of a philosophy they were meant to be. If the book claims that variable names should be chosen meaningfully, doesn't it then follow that variables whose names are small essays on their use are even better? Isn't `MaximumValueUntilOverflow` a better name than `maxval`? I don't think so.

What follows is a set of short essays that collectively encourage a philosophy of clarity in programming rather than giving hard rules. I don't expect you to agree with all of them, because they are opinion and opinions change with the times. But they've been accumulating in my head, if not on paper until now, for a long time, and are based on a lot of experience, so I hope they help you understand how to plan the details of a program. (I've yet to see a good essay on how to plan the whole thing, but then that's partly what this course is about.) If you find them idiosyncratic, fine; if you disagree with them, fine; but if they make you think about why you disagree, that's better. Under no circumstances should you program the way I say to because I say to; program the way you think expresses best what you're trying to accomplish in the program. And do so consistently and ruthlessly.

Your comments are welcome.

Issues of typography

A program is a sort of publication. It's meant to be read by the programmer, another programmer (perhaps yourself a few days, weeks or years later), and lastly a machine. The machine doesn't care how pretty the program is — if the program compiles, the machine's happy — but people do, and they should. Sometimes they care too much: pretty printers mechanically produce pretty output that accentuates irrelevant detail in the program, which is **as** sensible **as** putting all the prepositions **in** English text **in** bold font. Although many people think programs should look like the Algol-68 report (and some systems even require you to edit programs in that style), a clear program is not made any clearer by such presentation, and a bad program is only made laughable.

Typographic conventions consistently held are important to clear presentation, of course — indentation is probably the best known and most useful example — but when the ink obscures the intent, typography has taken over. So even if you stick with plain old typewriter-like output, be conscious of typographic silliness. Avoid decoration; for instance, keep comments brief and banner-free. Say what you want to say in the program, neatly and consistently. Then move on.

Variable names

Ah, variable names. Length is not a virtue in a name; clarity of expression *is.* A global variable rarely used may deserve a long name, `maxphysaddr` say. An array index used on every line of a loop needn't be named any more elaborately than `i`. Saying `index` or `elementnumber` is more to type (or calls upon your text editor) and obscures the details of the computation. When the variable names are huge, it's harder to see what's going on. This is partly a typographic issue; consider

```
for(i=0 to 100)
        array[i]=0
```

vs.

```
for(elementnumber=0 to 100)
        array[elementnumber]=0;
```

The problem gets worse fast with real examples. Indices are just notation, so treat them as such.

Pointers also require sensible notation. `np` is just as mnemonic as `nodepointer` *if* you consistently use a naming convention from which `np` means ‘‘node pointer’’ is easily derived. More on this in the next essay.

As in all other aspects of readable programming, consistency is important in naming. If you call one variable `maxphysaddr`, don’t call its cousin `lowestaddress`.

Finally, I prefer minimum-length but maximum-information names, and then let the context fill in the rest. Globals, for instance, typically have little context when they are used, so their names need to be relatively evocative. Thus I say `maxphysaddr` (not `MaximumPhysicalAddress`) for a global variable, but `np` not `NodePointer` for a pointer locally defined and used. This is largely a matter of taste, but taste is relevant to clarity.

I eschew embedded capital letters in names; to my prose-oriented eyes, they are too awkward to read comfortably. They jangle like bad typography.

The use of pointers.

C is unusual in that it allows pointers to point to anything. Pointers are sharp tools, and like any such tool, used well they can be delightfully productive, but used badly they can do great damage (I sunk a wood chisel into my thumb a few days before writing this). Pointers have a bad reputation in academia, because they are considered too dangerous, dirty somehow. But I think they are powerful *notation,* which means they can help us express ourselves clearly.

Consider: When you have a pointer to an object, it is a name for exactly that object and no other. That sounds trivial, but look at the following two expressions:

```
np
node[i]
```

The first points to a node, the second evaluates to (say) the same node. But the second form is an expression; it is not so simple. To interpret it, we must know what `node` is, what `i` is, and that `i` and `node` are related by the (probably unspecified) rules of the surrounding program. Nothing about the expression in isolation can show that `i` is a valid index of `node`, let alone the index of the element we want. If `i` and `j` and `k` are all indices into the node array, it’s very easy to slip up, and the compiler cannot help. It’s particularly easy to make mistakes when passing things to subroutines: a pointer is a single thing; an array and an index must be believed to belong together in the receiving subroutine.

An expression that evaluates to an object is inherently more subtle and error-prone than the address of that object. Correct use of pointers can simplify code:

```
parent->link[i].type
```

vs.

```
lp->type.
```

If we want the next element’s type, it’s

```
parent->link[++i].type
```

or

```
(++lp)->type.
```

`i` advances but the rest of the expression must stay constant; with pointers, there's only one thing to advance.

Typographic considerations enter here, too. Stepping through structures using pointers can be much easier to read than with expressions: less ink is needed and less effort is expended by the compiler and computer. A related issue is that the type of the pointer affects how it can be used correctly, which allows some helpful compile-time error checking that array indices cannot share. Also, if the objects are structures, their tag fields are reminders of their type, so

```
np->left
```

is sufficiently evocative; if an array is being indexed the array will have some well-chosen name and the expression will end up longer:

```
node[i].left.
```

Again, the extra characters become more irritating as the examples become larger.

As a rule, if you find code containing many similar, complex expressions that evaluate to elements of a data structure, judicious use of pointers can clear things up. Consider what

```
if(goleft)
    p->left=p->right->left;
else
    p->right=p->left->right;
```

would look like using a compound expression for `p`. Sometimes it's worth a temporary variable (here `p`) or a macro to distill the calculation.

Procedure names

Procedure names should reflect what they do; function names should reflect what they *return.* Functions are used in expressions, often in things like `if`'s, so they need to read appropriately.

```
if(checksize(x))
```

is unhelpful because we can't deduce whether checksize returns true on error or non-error; instead

```
if(validsize(x))
```

makes the point clear and makes a future mistake in using the routine less likely.

Comments

A delicate matter, requiring taste and judgement. I tend to err on the side of eliminating comments, for several reasons. First, if the code is clear, and uses good type names and variable names, it should explain itself. Second, comments aren't checked by the compiler, so there is no guarantee they're right, especially after the code is modified. A misleading comment can be very confusing. Third, the issue of typography: comments clutter code.

But I do comment sometimes. Almost exclusively, I use them as an introduction to what follows. Examples: explaining the use of global variables and types (the one thing I always comment in large programs); as an introduction to an unusual or critical procedure; or to mark off sections of a large computation.

There is a famously bad comment style:

```
i=i+1;        /* Add one to i */
```

and there are worse ways to do it:

```
/**********************************
 *                                *
 *          Add one to i          *
 *                                *
 **********************************/

                 i=i+1;
```

Don't laugh now, wait until you see it in real life.

Avoid cute typography in comments, avoid big blocks of comments except perhaps before vital sections like the declaration of the central data structure (comments on data are usually much more helpful than on algorithms); basically, avoid comments. If your code needs a comment to be understood, it would be better to rewrite it so it's easier to understand. Which brings us to

Complexity

Most programs are too complicated — that is, more complex than they need to be to solve their problems efficiently. Why? Mostly it's because of bad design, but I will skip that issue here because it's a big one. But programs are often complicated at the microscopic level, and that is something I can address here.

Rule 1. You can't tell where a program is going to spend its time. Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.

Rule 2. Measure. Don't tune for speed until you've measured, and even then don't unless one part of the code *overwhelms* the rest.

Rule 3. Fancy algorithms are slow when n is small, and n is usually small. Fancy algorithms have big constants. Until you know that n is frequently going to be big, don't get fancy. (Even if n does get big, use Rule 2 first.) For example, binary trees are always faster than splay trees for workaday problems.

Rule 4. Fancy algorithms are buggier than simple ones, and they're much harder to implement. Use simple algorithms as well as simple data structures.

The following data structures are a complete list for almost all practical programs:

array
linked list
hash table
binary tree

Of course, you must also be prepared to collect these into compound data structures. For instance, a symbol table might be implemented as a hash table containing linked lists of arrays of characters.

Rule 5. Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming. (See Brooks p. 102.)

Rule 6. There is no Rule 6.

Programming with data.

Algorithms, or details of algorithms, can often be encoded compactly, efficiently and expressively as data rather than, say, as lots of `if` statements. The reason is that the *complexity* of the job at hand, if it is due to a combination of independent details, *can be encoded.* A classic example of this is parsing tables, which encode the grammar of a programming language in a form interpretable by a fixed, fairly simple piece of code. Finite state machines are particularly amenable to this form of attack, but almost any program that involves the 'parsing' of some abstract sort of input into a sequence of some independent 'actions' can be constructed profitably as a data-driven algorithm.

Perhaps the most intriguing aspect of this kind of design is that the tables can sometimes be generated by another program — a parser generator, in the classical case. As a more earthy example, if an operating system is driven by a set of tables that connect I/O requests to the appropriate device drivers, the system may be 'configured' by a program that reads a description of the particular devices connected to the machine in question and prints the corresponding tables.

One of the reasons data-driven programs are not common, at least among beginners, is the tyranny of Pascal. Pascal, like its creator, believes firmly in the separation of code and data. It therefore (at least in its original form) has no ability to create initialized data. This flies in the face of the theories of Turing and von Neumann, which define the basic principles of the stored-program computer. Code and data *are* the same, or at least they can be. How else can you explain how a compiler works? (Functional languages have a similar problem with I/O.)

Function pointers

Another result of the tyranny of Pascal is that beginners don't use function pointers. (You can't have function-valued variables in Pascal.) Using function pointers to encode complexity has some interesting properties.

Some of the complexity is passed to the routine pointed to. The routine must obey some standard protocol — it's one of a set of routines invoked identically — but beyond that, what it does is its business alone. The complexity is *distributed.*

There is this idea of a protocol, in that all functions used similarly must behave similarly. This makes for easy documentation, testing, growth and even making the program run distributed over a network — the protocol can be encoded as remote procedure calls.

I argue that clear use of function pointers is the heart of object-oriented programming. Given a set of operations you want to perform on data, and a set of data types you want to respond to those operations, the easiest way to put the program together is with a group of function pointers for each type. This, in a nutshell, defines class and method. The O-O languages give you more of course — prettier syntax, derived types and so on — but conceptually they provide little extra.

Combining data-driven programs with function pointers leads to an astonishingly expressive way of working, a way that, in my experience, has often led to pleasant surprises. Even without a special O-O language, you can get 90% of the benefit for no extra work and be more in control of the result. I cannot recommend an implementation style more highly. All the programs I have organized this way have survived comfortably after much development — far better than with less disciplined approaches. Maybe that's it: the discipline it forces pays off handsomely in the long run.

Include files

Simple rule: include files should never include include files. If instead they state (in comments or implicitly) what files they need to have included first, the problem of deciding which files to include is pushed to the user (programmer) but in a way that's easy to handle and that, by construction, avoids multiple inclusions. Multiple inclusions are a bane of systems programming. It's not rare to have files included five or more times to compile a single C source file. The Unix `/usr/include/sys` stuff is terrible this way.

There's a little dance involving `#ifdef`'s that can prevent a file being read twice, but it's usually done wrong in practice — the `#ifdef`'s are in the file itself, not the file that includes it. The result is often thousands of needless lines of code passing through the lexical analyzer, which is (in good compilers) the most expensive phase.

Just follow the simple rule.