1. Indentation
2. Boolean Values
3. Precedence Problems
4. Global Variables
5. Function Arguments
6. Dirhandles
7. Repeated Tests
    1. Try it Both Ways
8. Error Messages
9. Botched Recursion
10. Function Conception
11. Main Loop
12. Repeated Tests
13. Say What You Mean
14. Do What You Mean
15. Leaning Toothpick Syndrome

Directory Copying

This chapter will discuss the function `xcopy()`, which was posted to `comp.lang.perl.misc` a few years ago. The purpose of this function is to perform a recursive directory copy.

Usually, when you write a recursive function, you must take care to avoid using global variables, or else the function doesn't work properly---the inner invocations of the function modify the global variables, and then the values are incorrect when the function returns. This function is remarkable because it works in spite of using global variables. When I first read the code, I said "this can't possibly work!" But then I tested it and I found that it *does* work. I was amazed.

Indentation

The first thing we must do is to fix the indentation of the program; apparently the author's editor doesn't do this automatically. It has the look of code written with the Windows "Notepad" application. I'm always shocked at the number of people who try to write code with "Notepad" or `pico` or other unsuitable editors. There are a lot of editor zealots in the world, and I'm not one of them. I regularly use several editors, including `emacs` and `vim`, and I don't get in arguments about which are better. But some editors are clearly deficient, and "Notepad" and `pico` are among these.

These editors were not designed for programming, and they are missing essential features. They don't autoindent the code, which means that you don't get the constant automatic feedback about missing punctuation that you get from auto-indenting editors. When you use `emacs`, if you omit a curly brace or a semicolon or make some other similar gross syntactic error, you know about it right away because the auto-indenting is obviously wrong. You can see it on the screen that some block is not where it should be, and **it’s easier to see than to think**.

But even worse, these non-programming editors don’t have a "go to line 537" feature. That means that when Perl gives you an error message about line 537, you don’t know where it is; you have to guess, or else you have to go to the top of the file and count your way down to line 537. And I have seen people use both these techniques, although never with much success.

You can’t do good work with bad tools. If you’re using `pico` or "Notepad" to write code, do yourself a favor and learn to use a better editor.

Anyway, `xcopy2.pl` is the same as `xcopy1.pl`, except that I have fixed the indentation. Now we can see the structure of the code, and which conditions control which blocks, without having to run the Perl parsing algorithm in our heads. Remember, **it’s easier to see than it is to think**.

Boolean Values

```
die "You have not defined the \$source_directory variable, sorry...\n
    if -d $source_directory !=1;
```

The red flag here is **testing a boolean value against 0 or 1**. The `-d` operator returns true or false already; there is no need to convert its value to a boolean with an explicit comparison with 1. The code should be simply:

```
die "You have not defined the \$source_directory variable, sorry...\n"
    unless -d $source_directory;
```

This also reads better. The `-d` operator is pronounced "is a directory". Now we’re asking the program to die unless `$source_directory` is a directory; before, we were asking it to die if the return value from `-d $source_directory` was unequal to 1.

Testing for equality with 1 is not always merely ungainly; sometimes it’s Just Plain Wrong. For example, the `perlfunc` manual says that the `-s` operator test if a "file has nonzero size". So the author of xcopy might have chosen to write something like this:

```
die "File is empty" unless -s $file == 1;
```

But this is wrong, because `-s` actually returns the size of the file, in bytes; it returns false when the file is empty, and true when it is nonempty, but the true value that it returns is not necessarily 1. The test above actually dies only if the file is exactly 1 byte long. The test above is correctly written as:

```
die "File is empty" unless -s $file;
```

Finally, even when testing a boolean value for equality with 1 isn’t actually wrong, it’s always silly. I think the *reductio ad absurdem* is this example:

```
if (($foo == 12) == 1) { ... }
```

Here we’re testing the result of `$foo == 12` for equality with 1 to make sure it’s true. But if we’re going to do that, why not go one step further?

```
            if ((($foo == 12) == 1) == 1) { ... }
```

And why stop there?

```
            if (((($foo == 12) == 1) == 1) != 0) { ... }
```

Perl has lots of operators that return boolean values; don’t be afraid to **let booleans be booleans**.

In addition to being indented correctly, `xcopy2.pl` eliminates these superfluous boolean comparisons. Otherwise, it is identical to the original `xcopy1.pl`.

Precedence Problems

This isn’t a red flag; it’s just an error. But it’s a common one; in this program it appears twice:

```
        24     { mkdir $target_directory || die "Couldn’t create dir : $target_node\
        36     mkdir $target_node || die "Couldn’t create dir : $target_node\n" } }
```

The intention is clear: the program should die if it can’t create the directories. But the code doesn’t actually do that; it will never call `die` at all. This is because of the relative precedence of function application and `||`; the code behaves as though it were written like this:

```
               mkdir ($target_node || die "Couldn’t create dir : $target_node\n");
```

The `die` is predicated on the value of `$target_node`, rather than on the result of the `mkdir`. To fix this, can insert parentheses to make the meaning clear:

```
               mkdir($target_node) || die "Couldn’t create dir : $target_node\n";
```

Or we can use the low-precedence `or` operator that was introduced for exactly this purpose:

```
               mkdir $target_node  or die "Couldn’t create dir : $target_node\n";
```

Or we could do both; this programmer did neither.

A good rule of thumb to follow is: use `or` for control flow; use `||` to select values.

Global Variables

Line 18 contains a fascinating and subtle error:

```
        18     my ($pwd,$i)=($_[0],$i++);
```

The fascinating thing here is that there are two *different* `$i` variables. The `my` expression is declaring a new lexical `$i` variable. But the scope of this new `my` variable does not begin until the *next* statement, so the `$i` on the right-hand side of the assignment is the *global* variable `$i`.

And in fact, if this were not the case, the function wouldn’t work properly. `$i` is used later on as the name of a dirhandle. The function calls itself recursively, and each instance opens another directory. If the instances shared the same dirhandle, then once instance would close the handle and return, and the instance to which it returned would find its dirhandle closed. So the various instances must use different dirhandle names. The global `$i` is serving as a counter which yields a different dirhandle name each time. If `$i` were lexical on the right-hand side of the assignment, it would be `undef` every time the statement was executed, and each instance of the function would use the dirhandle name `1`.

I am ambivalent about the use of `strict 'vars'`, but I think this one line, with its extremely tricky semantics and hidden global variable, is an excellent argument in favor of using it.

Function Arguments

```
        18     my ($pwd,$i)=($_[0],$i++);
```

I haven’t yet exhausted the wonders of line 18. It is noteworthy for another reason. Let’s look at the way the function is called, on line 15:

```
        13     &xcopy($source_directory,$target_directory);
```

The function is called with two arguments. Line 18 is responsible for acquiring the values of the arguments inside the function. But it only acquires one of them. The second argument is never used! The function only works because it happens to be in the scope of the lexical variable `$target_directory`, which is being passed implicitly, used as though it were a global variable; if the function were in a different file, it wouldn’t work because it wouldn’t know what the target directory was. A side note for `use strict` freaks, who often like to attribute magical powers of error diagnosis to the `use strict` declaration: `use strict` will not help here; it will not detect that anything peculiar is going on.

Let’s fix this. The function gets two arguments, so line 18 should be:

```
               my ($src,$dst) = @_;
```

I’ve used `$src` and `$dst` instead of `$source_directory` and `$target_directory` because I hate to type, but you may prefer diffrerent names. Whatever you call them, the hardwired variable names in these error messages no longer make sense:

```
        19     die "You have not defined the \$target_directory variable, sorry...\n
                 unless  -e $target_directory;
        20     die "You have not defined the \$source_directory variable, sorry...\n
                 unless  -d $source_directory;
```

I would prefer something simple like this:

```
         unless (@_ == 2) { die('Usage: xcopy($src, $dst)') }
```

Instead of a lot of verbiage describing what is wrong, we just provide a simple example of how the function should be called, because **it’s easier to see than it is to think**. The message on line 20 was the wrong one anyway; it would complain that `You have not defined the $source_directory variable` even if you had defined it.

Dirhandles

Now `$i` is used only as a dirhandle name; the programmer `++`'es it to obtain a new name for each call of the function. This depends on the variable being global. There is a more reliable way to generate a new dirhandle:

```
use IO::Dir;
my $dh = IO::Dir->new;
```

This costs an extra line of code, but you can recover the cost because you no longer need to use `closedir()`; the handle is closed automatically when `$dh` goes out of scope. This is a big semantic win, because it means your program is much less likely to have a handle leak.

Repeated Tests

Let's next concern ourselves with this:

```
if (! -d $dst && -e $dst) {
  die "Can't continue because a file has target's dir name\n";
} elsif ( ! -e $dst ) {
  mkdir $dst or die "Couldn't create dir : $target_node\n";
}
```

Here we have repeated code: the `-e $dst`. We can get rid of this by permuting the program logic a little bit:

```
if (-e $dst) {
  die "Can't continue because a file has target's dir name\n"
    unless -d $dst
} else {
  mkdir $dst or die "Couldn't create dir : $target_node\n";
}
```

For the `-X` tests, there's another little shortcut: `-X _` means to do the test on the same file that the last test was done on, so:

```
if (-e $dst) {
  die "Can't continue because a file has target's dir name\n"
    unless -d _;
} else {
  mkdir $dst or die "Couldn't create dir : $target_node\n";
}
```

Try it Both Ways

Folks in my classes have suggested alternative permutations:

```
if (-e $dst) {
  -d _ or die "Can't continue because a file has target's dir name\n"
} else {
  mkdir $dst or die "Couldn't create dir : $target_node\n";
}
```

Or:

```
if (! -e $dst) {
  mkdir $dst or die "Couldn’t create dir : $target_node\n";
} elsif (! -d _) {
  die "Can’t continue because a file has target’s dir name\n"
}
```

Which of the four versions do you like the best? Which do you like the least? I think I like the fourth one the best, although I’m not sure I can say exactly why. It seems to me to correspond most naturally to the way I think about the problem, with the most important thing (creating the directory) up front, and the less important part later.

Error Messages

This is a really rotten error message:

```
if (-e $dst) {
  die "Can’t continue because a file has target’s dir name\n"
    unless -d _;
} ...
```

Here’s why: Suppose the program dies in the middle and says:

```
Can’t continue because a file has target’s dir name
```

Now what are you going to do? *Which* file? Even if you understand what the message means, it doesn’t help. Now youget to go grovelling over the target directory, looking for the bad file. **Error messages should describe the cause of the problem**:

```
if (-e $dst) {
  die "File $dst exists but is not a directory" unless -d _
} ...
```

The important thing here is the addition of `$dst` to the message. Now, if the program dies, it will say something like:

```
File c:/temp/perl-5.7.1/lib/CGI/eg/crash.cgi exists but is not
a directory at xcopy.pl line 25
```

Notice also that Perl has appended `at xcopy.pl line 25`; Perl does this whenever the `die` message does not already end with a newline. Getting the information about line 25 is useful even if you are not trying to debug the program, because if you don’t understand the meaning of the message, you can look at the test on line 25 to see what the program was expecting to happen that did not happen.

I also fixed the broken grammar, although that is outside the scope of this book. The author was not a native speaker of English, so we should cut him some slack in the English grammar department.

Line 28 has a similarly bad error message:

```
28       opendir ($dh,$pwd) || die "Can’t list $pwd\n";
```

If this error occurs, we get the not-very-helpful message:

```
Can't list c:/tmp/boogie
```

Then we get to pull out our hair and scream "Why not? Does it have the wrong permissions? Did someone remove it while the program was running? Am I having a catastrophic disk failure?" There's no way to know from the message what the appropriate response is: Fix the permissions? Ignore it and rerun the program? Run into the machine room with a fire extinguisher?

Here's the improved version:

```
opendir ($dh,$pwd) || die "Can't open dir $pwd: $!";
```

`$!` means "Why not". When a system operation like `opendir` fails, `$!` contains the reason for failure. Now the message looks something like this:

```
Can't open dir c:/tmp/boogie: Permission denied
```

and this tells us that there is no need to go looking for the fire extingusher just yet.

Here's another message with the same problem:

```
} elsif ( ! -e $dst ) {
  mkdir $dst or die "Couldn't create dir : $target_node\n";
```

(Let's leave aside the fact that $target_node is undefined at this point in the program.)

What action should you take if see this message?

```
Couldn't create dir : c:/temp/boogie
```

It doesn't matter what you *should* do, because there's only one thing you *can* do, and that's stand around yelling "Why *not*? Why *couldn't* you create the directory, you lousy machine?" Again we should use `$!`:

```
} else {
  mkdir $dst
    or die "Couldn't create dir $dst: $!";
}
```

Now the message will say something like this:

```
Couldn't create dir c:/temp/boogie: Permission denied
```

Or:

```
Couldn't create dir c:/temp/boogie: Disk is on fire
```

(You may laugh at that, but Unix does have a "disk is on fire" error; it's called `EIO`.)

Botched Recursion

After fixing the error message, this line still has a big problem:

```
                opendir ($dh,$pwd) || die "Can’t open dir $pwd: $!";
```

The big problem is that we eliminated `$pwd`. It was defined on the notorious line 18:

```
18          my ($pwd,$i)=($_[0],$i++);
```

But we replaced line 18 with this:

```
            my ($src,$dst) = @_;
```

Since `$pwd` was the old name for the first argument, and `$src` is the new name, this suggests we should just replaced `$pwd` with `$src` throughout. But this worried me, because I had already replaced `$source_directory` with `$src` throughout, and I was worried that there was something I hadn’t understood. Merging two variables into one doesn’t usually work because the two variables usually have two separate meanings. Are `$source_directory` and `$pwd` really *exactly* the same thing?

The answer turns out to be "yes". It sure it confusing when functions use hardwired global variables.

Function Conception

The real problem here is that the author did not have a clear idea of what he wanted the function to do. A function should do a single task, and that task should be clearly defined and easily explained. `xcopy()` should have been simple.

"Recursively copy a source directory to a destination directory."

The arguments sto the function should also be clearly defined and easily explained:

The arguments are `$src`, the path of the source directory, and `$dst`, the path of the destination directory.

In this program, the author wasn’t able to make up his mind what the arguments were going to be:

```
13        &xcopy($source_directory,$target_directory);

18          my ($pwd,$i)=($_[0],$i++);

51            &xcopy($source_node) if -d $source_node;
```

Main Loop

That was a lot of preliminaries, but we are finally at the part of the program that does the actual work:

```
29          while (my $source_node=readdir $dh) {
30            next if $source_node=~/^\.*$/;
31            $source_node       = $pwd.’/’.$source_node;
32            ( my $relative_node ) = $source_node =~/$source_directory(.*)/;
               ...
```

```
        52       }
```

The test on line 30 causes the program to ignore the perfectly legitimate files named `...` and `....`, which may be a bug, but is probably not important. The red flag here is on lines 31 and 32. Why are we appending `$pwd` to the front of `$source_node` on line 31 when we are about to strip it off again on line 32? This is **taking two steps forward and one step back**.

Sidebar: X

People often write this:

```
        $file = join '', <FILE>;
```

Here, the `<FILE>` call forces Perl to split the file into lines. Then they throw away all the work that they just forced it to do because they are joining all the lines back together. An alternative:

```
        { local $/;
          $file = <FILE>;
        }
```

The drawback of this is of course that it is more verbose.

Now what is the programmer tryuing to accomplish with lines 31-32?

```
        31     $source_node       = $pwd.'/'.$source_node;
        32     ( my $relative_node ) = $source_node =~/$source_directory(.*)/;
```

It seems that he is trying to construct the filename for the destination of the file copy; to do this, he wants to take the source directory name from the front of the source file name and replace it with the target directory name.

But if we *don't* prepend the source directory name in the first place, then we won't have to remove it again afterwards.

Remember this?

The arguments are `$src`, the path of the source directory, and `$dst`, the path of the destination directory.

But this means that the path of the source file is simply:

```
        "$src/$source_node"
```

and the path of the destination file is simply:

```
        "$dst/$source_node"
```

That is a lot simpler than the code we had before!

The name `$source_node` looks peculiar in a destination filename, so I'm going to change the name of the variable to `$file`. Then we have:

```
        my ($src_file, $dst_file) = ("$src/$file", "$dst/$file");
```

which really couldn’t be any clearer. Three complicated lines of code have become one simple line. Also, the new version of the code eliminates a potential bug, which is that the program wouldn’t work properly if one of the filenames happened to contain a regex metacharacter, on account of the pattern match on line 32.

Repeated Tests

The rest of the program has this overall structure:

```
34          if (-d $source_node) {
              ...
41          } else {
              ...
50          }
51          &xcopy($source_node) if -d $source_node;
```

where the `...` segments are longish. The `-d $source_node` test appears twice, and it’s easy to eliminate the duplication by moving the call to `xcopy()` into the `if` block:

```
            if (-d $src_file) {
              ...
              xcopy($src_file, $dst_file);
            } else {
              ...
            }
```

Now let’s look at lines 34--40:

```
34          if (-d $src_file) {
35            if (! -d $dst_file && -e $dst_file ) {
36              die "Can’t mkdir $dst_file because a same name file exist\n";
37            } elsif ( ! -e $dst_file ) {
38              print "mkdir  $dst_file\n";
39              mkdir $dst_file or die "Couldn’t create dir : $dst_file\n";
40            }
```

You should now have a nagging feeling of deja vu. Does this code look familiar? It should. We have been over it already, on lines 23--27:

```
23        if (! -d $target_directory && -e $target_directory) {
24          die "Can’t continue because a file has target’s dir name\n";
25        } elsif ( ! -e $target_directory ) {
26          mkdir $target_directory || die "Couldn’t create dir : $target_nod
27        }
```

The code on lines 35--40 does the checks just before each recursive call, and the code on lines 23--27 does the same checks just after each recursive call. It’s tempting to eliminate one or the other set of checks. But why did the programmer put both sets in unless they were somehow necessary? If we didn’t have a test case worked up already, now is the time to build one. Then we can get rid of lines 35--40 and see if anything breaks.

But no, the program continues to work perfectly, so the `-d` branch of the program now looks like this:

```
        if (-d $src_file) {
          xcopy($src_file, $dst_file);
        }
```

which couldn’t be much simpler: if the source file is actually a directory, the function makes a recursive call to itself to copy the directory and its contents. Five lines of code have become zero.

Sidebar: Five lines become zero

When I first started writing this class, I expected that many of the slides would say things like:

Here, seven lines become three.

But often, they turned out to say things more like:

Here, seven lines become zero.

What can we conclude from this? My conclusion was:

Beginners use way too much code.

(Probably non-beginners also use way too much code.)

All this reminds me of when I was a little boy, learning to drive nails with a hammer. I would pound and pound and pound, and sometimes the nail would fall out and I would have to start over, POUND POUND POUND and then I would bend the nail and have to pull it out and start over.

One day I saw a professional carpenter driving a nail, and it was a revelation. I learned some tricks. For example, you must hold the hammer down by the end instead of up by the head, and when you hold the nail, use your first two fingers instead of your thumb and finger; that way your hand lies flat and you can hold the nail without as much rish of smashing your knuckles. But the most important thing I learned was that a professional carpenter does not go pound pound pound pound. The carpenter goes WHAM WHAM and the nail is in.

When I realized that, I changed my technique completely. Of course, I couldn’t go WHAM WHAM right away, but at least I was going in the right direction, and I eventually learned to do it that way. But if I had stuck with my original pound pound pound pound technique, I would never have gotten anywhere. The important piece of information was "Oh, that’s what it is supposed to look like!"

Beginning writers have the same problem: they use twenty words where five will do. A good exercise for writers is to consider each word and each sentence on its own, and for each word and each sentence to ask "can I cut this out?" Often the answer is "yes", and then you cut it out.

Edward Tufte, in his brilliant book *The Visual Display of Quantitative Information* has a similar piece of advice. How do you design a good chart or graph? Start by designing it any way you want. Then, for each little spot of ink on the page, ask "would my chart lose any information if I removed this bit of ink?" If the answer is no, then remove it. When all the remaining ink is data ink, you have a good chart.

You can do the same thing with code, and that is what this book is about.

The `else` branch has a similar repeated test:

```
                  if (-d $source_node) {
```

```
                ...
41           } else {
42             if ( -d $target_node && -e $target_node) {
43               warn "Can’t copy $target_node because a same name dir exist\n
44             } else {
```

We can get rid of lines 42--43 just as we got rid of the similar lines from the `if` branch.

Say What You Mean

```
 48     ‘$ENV{ComSpec} /c copy /b $copy_source_node $copy_target_node‘;
```

We’ve seen this before: Backquotes (`‘...‘`) tell Perl to execute a system command and to gather up the output of the command. Here, we’re not interested in the output, so `system` is more appropriate:

```
system("$ENV{ComSpec} /c copy /b $copy_source_node $copy_target_node");
```

Do What You Mean

```
44   } else {
45     print "coping $source_node to $target_node\n" unless -e $target_node
       ...
48     ‘$ENV{ComSpec} /c copy /b $copy_source_node $copy_target_node‘;
49   }
```

The `-e` test is weird here. Suppose the test fails because the file exists. Then the program skips printing the message---but it performs the copy anyway. If we take the message at face value, it suggests that we want to omit the copy if the destination file exists already:

```
} elsif (! -e $dst_file) {
  print "Copying $src_file to $dst_file\n";
  ...
  ‘$ENV{ComSpec} /c copy /b $copy_source_node $copy_target_node‘;
}
```

Leaning Toothpick Syndrome

```
(my $copy_target_node) = $dst_file; $copy_target_node=~s/\//\\/g;
(my $copy_source_node) = $src_file; $copy_source_node=~s/\//\\/g;
```

The goal here is to replace Unix-style `/` path separators with DOS-style `\` separators. The unreadable substitutions on the right-hand side are examples of the dreaded **Leaning Toothpick Syndrome**. The syntax of the `s` operator allows you to use any delimiter you want, not just slashes. So if you’re `s///`-ing a pattern or replacement text that contains a lot of slashes, don’t use `/` as the delimiter also; use something else, like `s{}{}`:

```
(my $copy_target_node) = $dst_file; $copy_target_node=~s{/}{\\}g;
(my $copy_source_node) = $src_file; $copy_source_node=~s{/}{\\}g;
```

In this case, `tr` is more appropriate than `s`. `s` searches for a regex and replaces it with a string; `tr` searches for all instances of a single character and replaces each one with a different character, which is exactly what we’re doing here:

```
        (my $copy_target_node) = $dst_file; $copy_target_node=~ tr{/}{\\};
        (my $copy_source_node) = $src_file; $copy_source_node=~ tr{/}{\\};
```

Similarly, consider using `tr/x//d` in place of `s/x//g`.

Next, I'm tempted to **avoid excess punctuation**:

```
        my $copy_target_node = $dst_file; $copy_target_node=~ tr{/}{\\};
        my $copy_source_node = $src_file; $copy_source_node=~ tr{/}{\\};
```

but in this case we can do better. We're making up new temporary variables to hold copies of `$dst_file` and `$src_file` so that we can `tr` them without damaging the original values. But we never use the original values again, so there was no point in preserving them. We may as well write:

```
        $src_file =~ tr{/}{\\};
        $dst_file =~ tr{/}{\\};
```

and just use `$dst_file` and `$src_file` later on.

Now it should be obvious that we have some repeated code, and **repeated code is a mistake**. As usual, it's easy to eliminate:

```
        tr{/}{\\} for $src_file, $dst_file;
```

An alternative approach would be to use `\` everywhere in the program, instead of using `/` and then converting to `\` at the last minute. Why use `/` if it's not what you want?

Sidebar: A Note About Portability

When I started teaching these classes, I thought that some audience member would be sure to raise their hand at this point to complain that the explicit use of `\` renders the program unportable; it will not work on a Macintosh, where the path separator is `:`, for example. I am happy to say that nobody ever has brought this up, but there is enough silly portability advice around that I would like to bring up the point anyway.

Portability proponents can be very dogmatic. We could fix this portability problem:

```
        # Instead of "$src/$file"
        $src_file = File::Spec->catfile($src, $file);
```

But there is an obvious cost: the code is more verbose and obscure. An alternative approach is to leave the program the way it is for the time being, and to change it on the first day that it needs to run on a Mac. That has a cost too, but it might be less expensive; if the day never comes, it is free.

I am not saying that portability is a bad thing. The important thing to remember about portability is that it is a feature, just like other features that software might have, and it has a cost and a benefit. As with any feature, you do not try to put it in just because. You do a cost-benefit analyszis and try to decide if added portability offers a benefit that is worth the cost. As as wise man once said:

Sufficient unto the day is the evil thereof

which, in modern English, trsnalates roughly as "don't go borrowing trouble". You have enough problems to solve today without also worrying about portability problems that might or might not arise tomorrow.

At this point we are done; the final version is in Program ???. It was 30 lines of code; now it is only 17.