Lecture 8

CSE 110

13 July 1992

1 A Function to Swap the Values of Two Variables

Consider a program which takes a long list of numbers and arranges them in numerical order. This operation is called *sorting*. By some estimates, 50% of all computer use in the world is devoted to sorting things. Many typical sorting algorithms work by finding two elements in the list that are out of order, and then swapping their positions. In C that would mean we would have a bunch of variables, one for each element in the list, and we would swap the values in the two variables that contained out-of-order elements.

If we were writing such a program, we would do a lot of swapping. We can swap the values of two variables, say `x` and `y`, as follows:

```
temp = x;   x = y;   y = temp;
```

where `temp` is a variable of the same type as that of `x` and `y`. Of course, that'll get ugly and cluttered if we have to write it a lot; it would be better if we could put it into a function and simply write something like

```
swap(x,y);
```

to swap the values of variables `x` and `y`.

1.1 A First Try

Here's our first cut at a program which sets up two variables and the calls a `swap` function to swap their values:

```
#include <stdio.h>

void swap(int n1, int n2);

int main(void)
{
  int x = 5, y = 119;


  printf("The values of x and y before the swap are %d and %d.", x, y);
  swap(x, y);
  printf("The values of x and y after the swap are %d and %d.", x, y);
  return 0;
}

void swap(int n1, int n2)
{
  int temp;

  temp = n1;
  n1 = n2;
  n2 = temp;
}
```

1.2 Why the First Try Doesn't Work

If you key this in and compile it, you'll discover it doesn't work. It compiles okay, but the values of `x` and `y` never get swapped. What's wrong?

The problem is this: `swap` only gets the *values* of its arguments `x` and `y`; it never finds out where `x` and `y` actually are so that it can change them. Put another way, the values `swap` receives are *copies* of the values stored in `x` and `y`. `swap`'s variables `n1` and `n2` are private variables; they get created when `swap` is called and destroyed again when `swap` returns. All the shuffling around of values that `swap` does is in its own private variables which get destroyed when it returns.

This is actually a feature and not a bug—normally, of course, we don't want functions that we call from `main` to be able to mess around with `main`'s variables. It would be very bad, for example, if `printf` surreptitiously changed the computer's secret number in the middle of your guessing game program.

The way out is the same as with `scanf`: instead of passing the values of the variables whose values we want to swap, we'll pass their addresses. Then `swap` will know where `x` and `y` are actually written on the blackboard, and it will be able to change their values all it wants. Note that even if we tell `swap` where `x` and `y` are on the blackboard, it still doesn't know where any of `main`'s other variables are, so it can only change the ones we wanted it to.

This would be a good time to go back and review the notes for Lecture 5, section 2.3: "Pointers and the `&` operator".

1.3 This One Works

```
#include <stdio.h>


void swap(int *n1, int *n2);


int main(void)
{
  int x = 5, y = 119;



  printf("The values of x and y before the swap are %d and %d.", x, y);
  swap(&x, &y);
  printf("The values of x and y after the swap are %d and %d.", x, y);
  return 0;
}

void swap(int *n1, int *n2)
{
  int temp;


  temp = *n1;
  *n1 = *n2;
  *n2 = temp;
}
```

1.4 The & Operator Again

First, note that the call to `swap` in `main` has changed from

```
swap(x, y);
```

to

```
swap(&x, &y);
```

. Now, recall what the `&` operator does: if `x` is some variable, then `&x` describes where that variable can be found on the blackboard. The value of `&x` is said to *point to* `x`; since the type of `x` is <`int`>, the type of `&x` is <pointer to `int`>.

1.5 The * Operator

Now we know how to make a pointer: we use the `&` operator. Once we have a pointer, how to we find out or change what it is pointing to? The complement of the `&` operator is the `*` operator, sometimes called the *dereferencing operator*. If p is a pointer value, then $*p$ is the object to which p points. So, for example, the value of `*&x` is the same as the value of `x`, because `*&x` means to get whatever `&x` points to, and `&x` points to `x`.

Let's suppose that `swap` was called from `main` as in the example above, and let's look at the individual statements in `swap` and see what they do. First, when `swap` gets called, `n1` gets assigned the value of the first argument, which is a pointer to the variable `x`. (That is, `&x`.) `n2` gets assigned the value of the second argument, which is a pointer to the variable `y`. (That is, `&y`.) Then, the line

```
temp = *n1;
```

in `swap` says to find the variable that `n1` points to (`x` in this case), get its value (5), and assign that value to the variable on the left, `temp`.

The line

```
*n1 = *n2;
```

says to find the variable that `n2` points to (`y`), get its value (119), and assign that value to the variable that `n1` points to (`x`).

The line

```
*n2 = temp;
```

says to get the value of `temp` (5), and assign that value to the variable that `n2` points to. (`y` in this case.)

1.6 `swap`'s Header

As usual, in `swap`'s header we have to tell the compiler what types of arguments `swap` will take and what type of return value it will return. We say that `swap` returns type `void`. That means that `swap` really doesn't return any value.

The parameter declaration `int *n1` declares a variable, `n1`, of type <pointer to `int`>. Here is how to remember what this means: If the declaration had said `int n1`, it would mean that `n1` is an <`int`>. But instead it says `int *n1`, so instead it means that `*n1` is an <`int`>. Since `*n1`, the thing `n1` points to, is an <`int`>, `n1` itself must be a <pointer to `int`>.

1.7 A Note About Functions that Return `void`

If a function's return type is `void`, that means it doesn't return a value at all. To return from such a function, we just write

```
return ;
```

, omitting the expression that usually follows the word `return`.

The other way to return from a `void` function is to just let control flow off the bottom of the function; this is the same as doing `return ;`.

If you do `return ;` in a non-`void` function, or let control flow off the bottom of a non-`void` function, the function may return a random garbage value to its caller, or your program may fail completely.

1.8 More About Prototypes

In section 2.2 of the notes for Lecture 5, there was a brief note about *prototypes*. You should go and reread that now if you've forgotten it; it's on page 5.

Here's the problem we must solve: At the time the compiler compiles the line `swap(&x, &y);` , it hasn't seen the definition of `swap` yet.[1] But the compiler

[1] In fact, it's quite possible that `swap`'s source code resided in a different file entirely, which was compiled eight months ago and then thrown away. I am not being facetious.

needs to write the machine instructions for the call to and return from `swap`, and to write those correctly it needs to know the types and number of the arguments and the type of the return value.[2] We need a way to give the compiler this information even in the absence, temporary or otherwise, of the `swap` function itself.

The way we do that is with a *prototype*. To write a prototype for a certain function, we write an ordinary function header for that function, but we follow it with a semicolon instead of a function body. The function header contains all the information the compiler needs to translate the call and return properly.

That's what the line `void swap(int *n1, int *n2);` is doing at the top of the program of section 1.3. It provides argument and return value type information for `swap` to the compiler in advance of the actual definition of `swap`.

If the compiler hasn't seen a prototype for a certain function at the time it compiles a call to that function, it guesses and does the best that it can. It assumes that the function's return value is <int>, which can lead to disaster if it isn't,[3] and it does the best it can to handle the arguments, which sometimes works out and sometimes doesn't.

You should have a prototype in your program for every function except `main`. (`main` always has the same return type and argument types anyway.)

Library functions like `printf` and `sqrt` must have prototypes too, so that the compiler can compile the calls to them correctly,[4] but the prototypes almost always appear in some header file, and so you include the prototype into your program by including the appropriate header file. For example, if you hunt up the `math.h` header file and look in it, you'll see, along with a lot of other nonsense, something like

```
double sqrt(double arg);
```

.

The actual definition of a function, the one that supplies the instructions about how to execute the function, includes a header for the function, and so it counts as a prototype—after the compiler has seen an entire function, it certainly has enough information to compile calls to that function. If we moved

[2] I'm afraid you won't be able to appreciate why this is until you've studied the inner workings of the compiler in detail. We won't do that in this course.

[3] Try writing a program that uses `sqrt` without including <math.h> and you'll see what kind of horrors can occur—you get completely bogus answers back from `sqrt` because the compiler thinks that `sqrt` is returning an <int> when it's really returning a <double>.

[4] In this case the source code for the called function *really* isn't available—it's locked in a vault somewhere at Borland.

the `swap` function so that it appeared before `main` in the file, we could omit the prototype, because by the time the compiler had to compile the call to `swap`, it would have seen the entire definition of the `swap` function and would have known all about it.

2 `break`

The `break` statement interrupts a loop prematurely. It's easy to use: You just write `break;`, and if the computer reaches the `break` statement, control immediately passes the the statement following the end of the smallest enclosing `while`, `do`...`while`, `for`, or `switch` statement.[5]

2.1 Examples of `break`

In each of these ghostly examples, the ⊗ symbol shows the place in the program to which the computer skips if it happens to execute the indicated `break` statement.

```
 while ( . . . ) {
     . . .
     if ( . . . )
       break ;
     . . .
 }
⊗
```

```
 while ( . . . ) {
     for ( . . . ) {
         . . .
         if ( . . . )
             break ;
         . . .
     }
     ⊗
     . . .
 }
```

[5] We haven't seen `switch` yet, but it's coming up soon.

```
do {
    . . .
    if ( . . . )
        if ( . . . )
            break ;
    . . .
} while ( . . . ) ;
```
⊗

Note that `break` does *not* care about `if` (or `else`) when it breaks; if it did, it would be useless. (Why?)

If you write a `break` statement that isn't enclosed by a `while`, `do`...`while`, `for`, or `switch` statement, the compiler will grouse and refuse to compile your program.

2.2 `break` Statement Considered Harmful?

Many early programming languages had only two control structures: They had an `if–else`, and they had the infamous `goto` statement, which unconditionally transferred control to a certain other statement.[6]

Around 1968, imperative languages such as ALGOL (a distant ancestor of C), were just beginning to have what are known as *logical control structures*, which let you express your algorithms in terms of blocks of code which were executed when certain conditions held, rather than in terms of a flow of control which jumped around the program from numbered statement to numbered statement in response to certain conditions.

In 1968 a gentleman named Edsger Dijkstra[7] wrote a note in *Communications of the ACM*, a well-known computer science journal, called *Goto Statement Considered Harmful.* He had discovered that when programmers in his organization were forbidden from using `goto`, and required to use only logical control structures, the programs they wrote had fewer errors and the errors the programs did have were easier to fix.

This is a reasonable thing to notice, and, because the logical control structure proponents were mostly right, most modern imperative languages, including

[6]This is an oversimplification. Many early programming languages, notably LISP, had an utterly different way of managing control flow in the first place, and this whole debate is moot for them. On the other hand, FORTRAN (which is much more like C than LISP is) had a whole mob of conditional and computed test statements, all of which ended by transferring control to the statement with a particular line number. The point stands anyway.

[7]In class I said it was Nicklaus Wirth, but I was mistaken.

C, stress logical control structures such as `while`, and have `goto` only as an afterthought, if at all.

Logical control structures are a little closer to the way we think than `goto` is. Rather than giving someone a laundry list of instructions like "14. If you're not at the store, `goto` step 11." we're more likely to say, "Keep walking north `until` you get to the store." `goto` is considered bad form in most cases, and although it occasionally has its uses, it really is better to avoid it whenever possible.

That warning extends to `break` (and to `continue`, although we haven't seen that yet): `break` interrupts the logical flow of control and causes an unconditional jump to somewhere else, and so it can be confusing; overuse of `break` can obscure what your program is really doing. When you see a `while` loop, you normally know what's going on; you can say, "Oh, this part of the program tries to do such-and-so `while` there is data left in the file." But if there's a `break` in the loop, you have to add on a qualifier: "...unless we hit the `break`."

I'm tolerant of `break`. Sometimes it's much easier to express something with a `break` than it would be without, and the code is shorter or clearer with the `break` than without. Nevertheless, there are a lot of logical-control fanatics in the world, and they'll tell you never to use `break`, `continue`, or `goto`, and to `return` only from the bottom of a function, and never from the middle.

In the CSE110 handbook, it says never to use `break` or `continue` in this class, and that you'll lose at least one point on any assignment you turn in that has a `break` or a `continue`. That's the logical-control fanatics talking. Since I am not a logical-control fanatic, that rule goes out the window. I am, however, a simplicity and clarity fanatic, because I spend a substantial part of my life reading other people's C code. So the rule I'd rather have you follow is this: When in doubt, write the code both ways, and pick the one that seems clearest and simplest.