Lecture 3

CSE 110

2 July 1992

1 Functions

In the bad old days before high-level languages like C, people wrote in assembly language. Every part of every program could see all the bytes at once.[1] This might sound like a good idea, but what happened was that people wrote all sorts of horrible programs. You might write a program which stored some payroll information in the memory of the computer, then printed a message on the screen, and then did something else with the payroll information. You would expect that printing a message would not change the payroll information. However, it might, depending on what the person who wrote the print function thought was the best thing to do. You might find that your program had an error because the print function had twiddled the payroll information when you were not looking. Worse still, if one day someone changed the way the payroll information was stored internally, they might not rewrite the print function to inform it of that fact, and then the print function would scramble the payroll information completely. People's programs had all kinds of horrible errors because all the sub-parts of their programs were mucking around with each others' data in unpredictable ways.

In C, and most other high-level languages, you can be sure that something like this will not happen. You can know that the print function will not try to change the payroll information. The way that you know this is that you do not tell the print function where the payroll information is on the blackboard. That is, the print function can't mess with the payroll information because it simply doesn't know where that information is.

Languages these days mostly differ in the methods they provide for letting one part of a program hide its information from another. How to divide up the blackboard is a big deal. One of C's major advantages over other languages is

[1] BASIC and FORTRAN are like this also.

that it is a particularly good language for talking about where things are on the blackboard.

In C, we break up programs into smaller, self-contained parts called *functions*. Each function has some instructions associated with it, and it has some variables that it gets to keep all to itself—no other functions can look at or alter those variables or even know where they are unless the function that owns them tells the other function. One fine point of programming is deciding the best way to carve up a problem into sub-parts so that the functions are as nearly self-contained as possible.

Every instruction in a C program is in one function or another. When you run a C program, the operating system takes care of starting the first function for you. This first function can start other functions, which in turn can start other functions. When a function starts another, we say that the first function has *invoked* or *called* the second function. When a function invokes another, the operating system stops executing instructions from the first function and starts executing instructions from the second function. When the second function is finally done, control returns to the first function and it goes on with what it was doing. Thus only one function is ever executing at any time.

Of course the functions do have to communicate with each other a little, and they do that in a specific and controlled way. The way we do this in C is by *passing arguments in* to a function when it starts, and by *receiving a return value* from a function when it is finished. Starting up a function is called *calling* the function or *invoking* the function.

Only one function is ever active at any time. When a function calls another function, it pauses, and waits for the called function to finish, before it continues with what it was doing.

When a function calls another function, it decides on the *arguments*, which are some data that it will pass to the function it is calling. The called function gets to see these arguments, and the things it does might depend on them. When the called function is finished, it gets to return some value to the calling function. So the arguments are the way a function communicates to a function that it calls, and the return value is the way a called function communicates information back to its caller.

1.1 Arguments

Suppose I decide that I will have a function which computes the square of an integer. (The square of a number is what you get when you multiply the number by itself.) We'll call this function `square`. Now whatever other function calls

`square` must tell it at least one thing: The number that `square` is supposed to square. The calling function will therefore communicate its number to `square` by passing in that number as an argument.

The way we call a function in C is by writing its name, an open parenthesis (`(`), the arguments, if there are any, and then a close parenthesis (`)`). For example, here's how we might call `square`:

```
square(57);
```

When the function that is executing reaches this instruction, it pauses. It doesn't go on to the next instruction until it has computed the square of 57. To do that, it saves somewhere some information about what it was doing and transfers control to the `square` function. When `square` is finished, control returns to the calling function, which picks up where it left off. The details of how the function remembers where it left off, and how it gets back to what it was doing after the called function is finished, are the compiler's problem.

1.2 What a Function Looks Like

A function has two parts. The first part, called the *header*, says the name of the function, how many arguments it has and what they are called and what their types are, and what type of result it returns to its callers. The second part is the *body* for the function. It is enclosed between curly braces (`{` and `}`) and it contains the instructions to the compiler about what to do when the function is called.

Here's how we might write `square`:

```
int square(int n)
{
  return n*n;
}
```

This is our first real C code, and so we'll discuss it at length.

The first line is the header. A header has three parts: First, the name of the type of the return value that the function returns to its callers, in this case `int`. Then comes the name of the function, in this case `square`. Then, in parentheses (`(` and `)`) is a list of *parameter declarations*, which describe what types of values a calling function is allowed to pass in, and what to names to give to these values. In this case there's only one argument, and `int n` means that

that argument will have type <int> and that within the body of the square function, the name n will refer to that argument.

The open curly brace ({) and close curly brace (}) *delimit* the function's body—they say where the body begins and ends. When there's more than one function in the program, the braces help the compiler understand where one function leaves off and another begins.

In between the braces is the code for the function itself. The star * means multiplication. return means to compute the value of the mathematical expression that follows, and return control to the calling function, passing it back the value of the mathematical expression. So this line computes $(n \times n)$ and immediately returns the result to the calling function, which was waiting around for that. When a function executes a return instruction, that means that it is done.

Normally, the instructions in a function are executed in order, from top to bottom. After an instruction is executed, control passes to the instruction on the next line. The return instruction is an exception to this. When a function executes a return instruction, it does *not* go on to the next instruction. Instead, control returns to the function that called the one that executed the return instruction. The next instruction that is executed is in the *calling* function, which picks up right where it left off.

1.3 How the Program Starts

The operating system arranges that if your program has a function named main, that function will be called first. If your program has no main, the linker will complain and you won't be able to run your program.[2]

When main executes a return instruction, control is returned to the operating system, and your program is over. The value you return with the return statement from main gets passed back to the operating system as a status code about whether the program succeeded or failed. Conventionally, main returns zero for success and nonzero for failure.

2 Statements

This is just terminology: Each instruction in a C program is called a *statement*. Statements are easy to recognize: Simple statements always end with

[2] We'll find out why it is the linker sometime later.

semicolons.

You can group statements together into *blocks*, which are also called *compound statements*. A compound statement is just a bunch of other statements (which might be simple or compound), with an open brace at the beginning and a close brace at the end. To execute a compound statement, the computer just executes the statements that make it up, in order.

The body of a function is always a single compound statement.

3 The Assignment Statement

The *assignment statement* performs a fundamental operation: It copies information from one part of the blackboard to another.

Its syntax is:

lvalue `=` *expression* `;`

Like all simple statements, it ends with a semicolon to tell the compiler where its end is.

It has three parts: There's an *object* on the left, an equals sign in the middle, and an *expression* on the right. *expression* means the same as it does in mathematics. An *lvalue* is an expression that refers to part of the blackboard—for example, the name of a variable is an lvalue; it refers to the part of the blackboard where the variable is stored.[3]

When the computer executes an assignment statement, it first *evaluates* the expression on the right; that means that it reads it and performs whatever calculations are necessary to determine its value. For example:

- If the expression is just a variable name, then the value of the expression is just the value stored in the variable.
- If the expression is a function call like `square(57)`, the computer pauses, calls the function, and the value of the expression is whatever the called function returned with its `return` statement.
- If the expression is just a numeral, like 57, then the value of the expression is just the value of that numeral.

[3] 'lvalue' is pronounced 'ell-value'.

After the computer has calculated the value of the expression, it stores that value into the part of its memory referred to by the lvalue.

3.1 Examples of Assignment Statements

We'll suppose that someone has already declared <`int`> variables called `x` and `y`.

```
x = 10 ;
```

The computer evaluates the expression on the right of the equals sign; the expression is just `10`, and so the value of the expression is the integer 10. Then the computer stores the integer 10 into the variable `x`.

```
y = x ;
```

The computer evaluates the expression on the right of the equals sign; the expression is just `x`, and so the value of the expression is the the value of the variable `x`, which is now 10. Then the computer stores the integer 10 into the variable `y`.

```
x = x + 1 ;
```

The computer evaluates the expression on the right of the equals sign. It evaluates the `x`, whose value is 10, and it evaluates the `1`, whose value is 1, and then it adds those two numbers together, because `+` means addition. The result, 11, is the value of the expression on the right of the equals sign. Then the computer stores the integer 11 into the variable `x`.

The following is *not* a legal assignment statement; the compiler will refuse to compile it:

```
4 = x ;
```

This fails because `4` is not an lvalue. That is, it does not refer to a part of the computer's memory, that way a variable name does. It is just a number. In short, you can't do this because `4` is not the name of a place where you can store a value.

3.2 Value Contexts and Object Contexts

Notice that in the statement

```
y = x ;
```

there is an asymmetry: `x` is evaluated, but `y` is not. In fact, `y`'s value is completely irrelevant—the compiler evaluates `x` because it is going to store that value in the part of memory referred to by `y`, but it never evaluates `y` at all.

`x` and `y` in this statement are canonical representatives of the two contexts that an expression can be in in C. One context is called the *value context*, and it means that the expression will be evaluated to produce a value. `x` here is in a value context. The other context is called the *object context*. It means that the expression will not be evaluated; instead, it is being used to refer to part of the computer's memory. `y` here is in an object context.

lvalue and *object context* are really two sides of the same coin.[4] Only an lvalue may appear in an object context; conversely, to decide if something is an lvalue, just see if it makes sense in an object context, such as on the left-hand side of an assignment statement.

For example: Is `x + y` an lvalue? No, because

```
x + y = 4 ;
```

doesn't make any sense—it says to store the value 4 into the part of the computer's memory represented by `x + y`, and that doesn't mean anything.

3.3 A Real Program

I fibbed a little about the right way to call the function `square`. Here I'll present a real program for computing the square of the number 57 and printing it out.

```
int main(void)
{
  int the_square;

  the_square = square(57);
  printf("The square of 57 is %d.\n", the_square);
  return 0;
}

int square(int n)
{
  return n*n;
}
```

[4] In fact, many people say *lvalue context* instead of *object context*.

There are two functions here: `main` and `square`. Notice that `main` is a function which returns a value of type <`int`> and which takes a value of type <`void`>. <`void`> is a special type which means "no value at all". So when the operating system first invokes `main`, it shouldn't give it any arguments. When `main` finishes, it'll return an <`int`> to the operating system to report whether it succeeded of failed.

The first line in `main` declares a variable, `the_square`, in which we will store the result of squaring the number 57. The compiler finds space for an int and arranges that if we name `the_square` again that the value in this space is used.

The second line is something we haven't seen before: An *assignment* instruction. We'll discuss it at length tomorrow, but what it says to do is to compute the value of the mathematical expression on the right of the `=` sign, and that value into the part of the blackboard represented by the object on the left of the `=` sign. The value of a function is whatever is returned by that function with a `return` instruction.

When control reaches this point in `main`, `main` stops and waits, and control passes to `square`. The compiler has arranged that the number 57 gets copied into the part of the blackboard referred to by `n`, so that when you ask for the value of `n`, you get the number 57. `square` fetches the value of `n`, multiplies it by the value of `n`, and returns that result. The compiler arranges that the result (it happens to be 3249) gets stored into the variable `the_square`. Then control returns to `main`.

The next thing that happens is that control passes to the next line in `main`. We haven't seen the `printf` function yet, but the call we've made to it here makes it print out

```
The square of 57 is 3249.
```

We don't have to supply code for `printf`; it's already written and stored in a file called a *library*. It's the linker's job to see if we used any functions, such as `printf`, for which we didn't supply any code, and, if so, to search for them in the libraries and to include the code for them if necessary. We'll discuss this more later.

Notice that we passed two arguments to `printf`, separated by a comma (`,`): A message to print, and the value of `the_square` to fill into the message. We'll talk about the details of what is going on here soon.

Finally, `main` itself returns a value, 0, to the operating system, to indicate that it completed successfully.

3.4 A Note About Local Variables

The variables `n` and `the_square` are called *local* variables. That means they are usable only in the functions in which they're declared. If you tried to use the variable `n` in the function `main`, or if you tried to use the variable `the_square` in the function `square`, the compiler would complain and would refuse to compile your program. That way you can be sure that the functions are communicating only in the way you expect them to, by passing arguments and returning return values, and that they are not mucking around with each others' private data.