Functions Revisited

This lecture describes in more detail the purpose of functions, their declarations in C, how arguments are passed to functions and how values are returned.

Why Use Functions?

A C program consists mainly of function declarations. A simple program may only declare a function called `main()`[1], but non-trivial programs will contain many function declarations.

The main purpose of functions is to break up a long program into small parts which are easier to understand. Each function should perform a relatively simple task which can be easily understood, programmed and tested. A rule of thumb is that a function should fit on one screen (about 25 lines). If the function is longer than this it probably can (and should be) broken down into two or more simpler functions.

A typical program consists of a hierarchy with a main function which uses (calls) other functions; these functions in turn call other functions and so on.

A function communicates with other functions through the values of its arguments and through the values is returns. This isolation between functions makes it easier to debug a function and to make sure that it behaves as desired. It also makes it more likely that the same function can be re-used in a future project.

Another reason to create a function is when the same (or similar) operations needs to performed on different variables. For example, if we had to compute the lengths of various strings in different places in a program, we could write a function that took a string (`char`acter array) argument and returned an integer. Instead of writing similar code in each place where we need to find the length of the string we could just write the function once and call it in each place where the length of a string has to be computed.

Other computer languages have special names for `void` functions: in `FORTRAN` they are called `subroutine`s and in Pascal they are called `procedure`s.

[1] Parentheses used after a name are used to indicate that the identifier is a function rather than a variable.

Local Variables

Variables declared inside a function are only "visible" within that function. A variable of the same name in another function is considered to be a different variable. Thus you cannot refer to a variable declared in one function from another function. For this reason they are known as "local" variables.

The value of a local variable is undefined when a function begins unless that variable is an argument or it is explicitly initialized. The value of a local variable is lost each time the function returns.

To pass values between functions you need to use function arguments and return values.

Function Declarations

A function declaration has four parts:

- The return type. This specifies the type (e.g. `int` or `char`) of the value returned by the function when it is used in an expression. If the function does not return any value it should be declared to be of type `void`.
- The name of the function. This is the name used to refer to a function when it is used in an expression.
- The parameter list. This list specifies a number of local variables that are used to pass values to the function. The list is surrounded by parentheses and the individual variable declarations are separated by commas (unlike a normal variable declaration where declarations are terminated with semicolons).
- The body of the function. This is a sequence of statements surrounded by braces. Each statements is terminated by a semicolon.

Exercise: Write the first 3 parts for a declaration of a function named `cdist` that returns an `int` and takes two `char` arguments called `x` and `y`.

Flow of Control[2]

To use a function you use its name in an expression. When the computer executes that expression (that is, evaluates its value) the computer transfers control to the first statement in the function. To "call" a function is to transfer control to it – i.e. use it in an expression.

When a `return` statement is executed control returns to back to the place where that function was called.

Each function also contains an implicit return statement after the last statement in the function and so functions also return after the last statement in the function is executed.

Function Parameters and Arguments

When a function is declared, a number of variable are declared in parentheses immediately following the function name. Strictly speaking, these variables are called *parameters* rather than arguments (the arguments are the values used in the calling expression). Within the function these variables are treated like other variables. The only difference is that when each time the function is called the values of the parameters are initialized to the values of the corresponding arguments in the function call.

There must be a one-to-one correspondence between the arguments used in a function call and the parameters in the function declaration. For example, if we declare a function with one character parameter, then when that function is used it must always have one character-type argument. Similarly if we declare a function with two `int` and one `char` parameters, then it must always be called with two `int` and one `char` arguments.

The way a computer passes arguments to a function is to copy the values of the arguments in a special set of memory locations called the *stack*. When the function executes it removes the values from the stack and assigns them to the parameters. Because the values or the arguments were copied to the parameters, this means that changes to the function parameters have no effect on the values in the calling expression.

This copying operation is called "passing by value."

An exception to the "passing by value" rule is the case of arrays. To improve efficiency, array arguments are not copied to the corresponding array parameter. Instead the location in memory of the array is passed to the function. This is called "passing by reference." Changes to an array parameter in a function will change the corresponding array argument in the calling function.

Return Values

When a function is declared its return type (possibly `void`) is specified. If the function return type is non-`void` a return statement must be used to return control to the calling function. The `return` keyword must be followed by an expression whose value is the value to be returned.

Exercise: What's wrong with the following function?

```
int strstr ( char s1[], char s2[] )
{
   ...
   return s1 ;
}
```

Order of Function Declarations

The compiler must see a declaration for a function before that function can used in an expression. This means you must declare your functions before using them in other functions, including the `main()` function.

For built-in functions the function declarations are contained in "include" files that are included in your program when a `#include` line is used in you program. By convention, the `#include` lines are placed near the start of the program before any other function declarations.

It not possible to declare one function within the declaration of another function.

[2] "Flow of control" is the sequence in which statements in a program are executed.