Functions: function definition, return statement, function prototype

We typically break a problem into smaller parts, with each part performing a specific task that contributes to the overall solution. A large program consists of one or more files, where each file contains zero or more functions, one of them being the "`main()`" function. Program execution begins with the "`main()`", which can call other functions, including library functions such as "`scanf()`" and "`fabs()`". Similar to "`main()`", a function can be called from inside other functions too. To illustrate the use of functions, suppose that we are writing a program that calculates the transpose of a matrix. We break the program into three parts. We write a function `matread()` that reads a matrix. Similarly, the function `transmat()` finds the transpose, and `matprint()` prints a matrix. First, we declare the matrices (2D array to be discussed later) along with their row and column dimensions in the "`main()`". Then we call `matread()` to read the matrix. Next, we call `matprint()` to print the matrix to verify that it is read correctly. This is followed by `transmat()`, which computes its transpose. Finally, we call `matprint()` to print the transpose of the matrix. Clearly, the number of statements in the "`main()`" is minimal. Also, the functions `matread()` and `matprint()` can be used in other matrix problems as well.

There are many reasons to write a program as a collection of many functions. It is easier to write a small, correct function that does one job. Both writing and debugging become easier. It is also easier to maintain or modify such a program. One can readily change just the set of functions that need to be rewritten, expecting the rest of the code to work correctly. Also, small functions tend to be self-documenting and highly readable.

Function definition: A function definition is the C code that describes what a function does. It has the following general form:

```
type function_name(parameter list)
{
declarations
statements
}
```

Everything in the first line constitutes the header of the function definition. And everything between the curly braces constitutes the body of the function definition. The parameter list is a comma-separated list of declarations. An example of a function definition is the following:

```
int sum3(int a, int b, int c)
{
int s;
s=a+b+c;
```

```
return s;
}
```

The `function_name` is `sum3`. The first `int` tells the compiler that the value returned by the function will be converted, if necessary, to an `int`. The parameter list consists of the declaration `int a, int b, int c`. This tells the compiler that the function takes three arguments of type `int`. An expression such as `sum3(2,4,7)` causes the function `sum3()` to be invoked, or called. The result is to execute the code that constitutes the function definition, with `a, b, c` having the values `2, 4, 7` respectively. The statement "`int s;`" is the declaration within the function body. The next two statements are also part of function body in which the last statement "`return s;`" returns an `int` value. Here is another example of a function definition.

```
void greeting(void)
{
 printf("Hello Mars!\n");
}
```

The first `void` tells the compiler that this function `greeting()` returns no value; the second `void` tells the compiler that this function takes no arguments. We can omit the second `void` to get the following form:

```
void greeting()
{
 printf("Hello Mars!\n");
}
```

However, the first `void` is mandatory. Note that we write `int main()`, which implies that `main()` takes no argument and it returns an `int`. However, we can also write `main()` that also accepts arguments. Obviously, we can also have `void` functions that take one or more arguments and non-`void` functions that take no arguments.

The function definition is thus summarized as follows. A function definition starts with the type of the function. If no value is returned, then the type is `void`. If the type is something other than `void`, then the value returned by the function will be converted, if necessary, to this type. The name of the function is followed by a parenthesized list of parameter declarations. The parameters act as placeholders for values that are passed when the function is invoked or called. The function body is a block, or compound statement, and it may also contain declarations. Note that the `sum3()` described above can be shortened as follows:

```
int sum3(int a, int b, int c)
{
return (a+b+c);
}
```

Note that the parentheses in the `return` statement are not necessary. Let us write another function that returns the square root of a real number. If the real number is negative, then we stop the program inside the function. Note that execution of a program stops whenever it hits "`return 0;`" statement within `main()` function. However, `return 0` expression can not be used in a function since it implies that the function is returning value `0`. To stop the execution inside a function [including `main()`], we may use `exit()` function. To use `exit()` function, we need to include the header file "stdlib.h".

```
double  sqroot(double x)
{
 if(x>=0)
    {
    return sqrt(x);
    }
 printf("Wrong argument, x must be positive\n");
 exit(1);
}
```

Also observe that the `return` statement may not appear at the end of the function body.

The return statement:

A `return` statement is of the form "`return;`" in the case of `void` function, and "`return expression;`" in the case of non-`void` function. The `expression` being returned may be enclosed in parentheses, but this is not required. When a return statement is encountered, execution of the function is stopped and the control is returned back to the calling environment. If the return statement contains an `expression`, then the value of the `expression` is passed back to the calling environment as well. Moreover, this value will be converted, if necessary, to the type of the function as specified in the function definition. For example if the type of the function is `double` and we have "`return  2;`" as the statement, then `2` will be converted to `2.0` before return.

There may be zero or more `return` statements in a function. A return statement can appear anywhere inside the function body. If there is no `return` statement, then control is passed back to the calling environment when the closing curly brace `}` of the function body is encountered. This is termed as "falling off the end". There can be more than one return statement inside the function body. For example, consider the following function that returns maximum of two integers.

```
int maxfunc(int a, int b)
{
 if(a>b)
 {
```

```
 return a;
}
else
{
 return b;
}

}
```

Function prototypes:

Functions must be declared before they are used. This is accomplished using a declaration syntax called the function prototype. A function prototype tells the compiler the number and type of arguments that are to be passed to the function and the type of the value that is to be returned by the function. For example, consider the following:

```
int maxfunc(int, int);
```

This tells the compiler that `maxfunc()` is a function that takes two arguments of type `int` and returns an `int`. The general form of a function prototype is

```
type function_name(parameter type list);
```

The parameter type list is typically a comma-separated list of types. Identifiers are optional. For example, the function prototype
`int maxfunc(int a, int b);` is equivalent to `int maxfunc(int, int);`
Identifiers such as `a` and `b` that appear in parameter type lists in function prototypes are not used by the compiler. Their purpose is to provide documentation to the programmer and other users of the code.

There is a way to write a C program with functions that does not use function prototypes. For example, consider the following C program:

```
#include<stdio.h>
int maxfunc(int a, int b)
{
 if(a>b)
 {
  return a;
 }
 else
 {
  return b;
```

```
 }

}

int main()
{
 int a,b;
 printf("Enter two integers:");
 scanf("%d%d",&a,&b);
 printf("Max of %d and %d is %d\n",a,b,maxfunc(a,b));
 return 0;
}
```

Listing 37: C program "nproto.c"

Here is an expected input-output.

```
Enter two integers:4 7
Max of 4 and 7 is 7
```

Since the compilation is carried out starting from the top, the function `maxfunc()` is compiled before the `main()` function. Hence, when the function `maxfunc()` is called inside the `main()` [argument of `printf()`], the type of the function and parameter type list are already known. *However, this style is rarely used in practice, and we would not use it from here onward.* The reasons are that there may be many functions in a given program, and writing all of them before `main()` makes the `main()` at the end of a large file. This is very inconvenient, since we often need to change the `main()` function during the writing and debugging stages. Also, a large program is usually split across two or more files, with each file containing a few functions. Writing `main()` at the top then does not make any sense.

Thus, we write a C program with function prototypes. The same program shown in Listing 37 is written using a function prototype.

```
#include<stdio.h>
int maxfunc(int, int);      // function prototype
int main()
{
 int a,b;
 printf("Enter two integers:");
 scanf("%d%d",&a,&b);
 printf("Max of %d and %d is %d\n",a,b,maxfunc(a,b));
 return 0;
```

```
}

int maxfunc(int a, int b)
{
 if(a>b)
 {
  return a;
 }
 else
 {
  return b;
 }

}
```

Listing 38: C program "proto.c"

Obviously, we get the same expected input-output. Since the function prototype is declared above `main()`, the compiler knows the type of the function and the parameter type list before it is called from inside `main()`. Note that we have written the function prototype above `main()`. This could have been written inside `main()` too. However, since a function may be called from other functions apart from `main()`, it is better to put it above `main()` so that other functions also know about this function. If we write many functions in more than one file, then we need to include the prototype of a specific function in each file from which this specific function is called.

Return to Lecture Topics