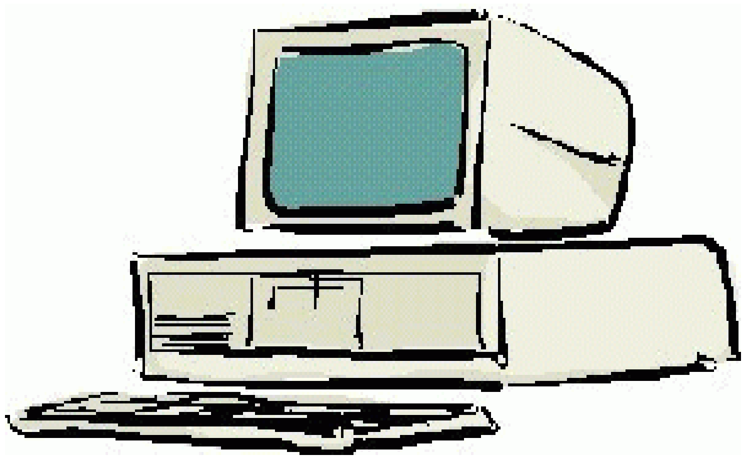
Programming in C



Session 9

Seema Sirpal
Delhi University Computer Centre

Functions

Functions are used to encapsulate a set of operations and return information to the main program or calling routine.

Encapsulation is detail, information or data hiding. Once a function is written, we need only be concerned with what the function does.

That is, what data it requires and what outputs it produces. The details, "how" the function works, need not be known.

Functions

The use of functions provides several benefits.

First, it makes programs significantly easier to understand and maintain. The main program can consist of a series of function calls rather than countless lines of code.

A second benefit is that well written functions may be reused in multiple programs. The C standard library is an example of the reuse of functions.

A third benefit of using functions is that different programmers working on one large project can divide the workload by writing different functions.

Defining & Declaring Functions

A function is declared with a prototype.

The function prototype, which has been seen in previous lessons, consists of the return type, a function name and a parameter list.

The function prototype is also called the function declaration.

Here are some examples of prototypes.

```
return_type function_name(list of parameters);

int max(int n1, int n2);             /* A programmer-defined function */
int printf(const char *format,...);  /* From the standard library */
int fputs(const char *buff, File *fp);  /* From the standard library */
```

Defining & Declaring Functions

The function definition consist of the prototype and a function body, which is a block of code enclosed in parenthesis.

A declaration or prototype is a way of telling the compiler the data types of the any return value and of any parameters, so it can perform error checking.

The definition creates the actual function in memory. Here are some examples of functions.

```
int FindMax(int n1, int n2)
{
   if (n1 > n2)
   {
      return n1;
   }
   else
   {
      return n2;
   }
}
void PrintMax(int someNumber)
{
   printf("The max is
%d\n",someNumber);
}
void PrintHW()
{
   printf("Hello World\n");
}
float FtoC(float faren)
{
   float factor = 5./9.;
   float freezing = 32.0;
   float celsius;

   celsius = factor * (faren - freezing);

   return celsius;
}
```

Defining & Declaring Functions

There are a few significant things to notice in these examples.

The parameter list of a function may have parameters of any data type and may have from no to many parameters.

The return statement can be used to return a single value from a function.

The return statement is optional. For instance, the function PrintHW does not return a value.

Finally, observe that variables can be declared within a function. These variables are local variables. They have local scope. Scope refers to the section of code where a variable name is valid and may be used. We'll see more on scope in the next Session.

Using Functions

A function should always be declared prior to its use to allow the compiler to perform type checking on the arguments used in its call.

```
/* Include Files */
#include <stdio.h>
/* Function Declarations */
int FindMax(int n1, int n2);
void PrintMax(int someNumber);
void PrintHW();
float FtoC(float faren);
int main()
{
   int i = 5;
   int j = 7;
   int k;
   float tempInF = 85.0;    /* A nice sunny day */
   float tempInC;
   PrintHW();    /* Prints Hello World */
   k = FindMax(i,j);
   PrintMax(k);    /* Prints Max Value */
   tempInC = FtoC(tempInF);
   printf("%f Fahrenheit equals %f Celsius \n",tempInF,tempInC);
   return 0;
}
/* Function Definitions */
int FindMax(int n1, int n2)
{
   if (n1 > n2)
   {
      return n1;
   }
   else
   {
      return n2;
   }
}
void PrintMax(int someNumber)
{
   printf("The max is %d\n",someNumber);
}

void PrintHW()
{
   printf("Hello World\n");
}
float FtoC(float faren)
{
   float factor = 5./9.;
   float freezing = 32.0;
   float celsius;

   celsius = factor * (faren - freezing);

   return celsius;
}
```

Using Functions

Notice that the functions are declared prior to their use.

The function definitions are actually below main in this file.

Also, notice that the function definitions and declarations match. They have the same return type, names, and parameters.

The function definitions need not even be in this file.

For instance, when you use library functions, the definitions are not in your file.

A program may be separated into multiple files. Main, along with the needed function declarations may be in one file.

Practice

1) Write functions to convert feet to inches, convert inches to centimeters, and convert centimeters to meters. Write a program that prompts a user for a measurement in feet and converts and outputs this value in meters.

   Facts to use: 1 ft = 12 inches, 1 inch = 2.54 cm, 100 cm = 1 meter.

Solution

1)
```
#include <stdio.h>

double feetToInches(double feet);
double inchesToCentimeters(double inches);
double centimetersToMeters(double cm);
int main()
{
   double feet,inches,cms,meters;

   printf("Enter your measurement in feet\n");
   scanf("%lf",&feet);

   inches = feetToInches(feet);
   cms = inchesToCentimeters(inches);
   meters = centimetersToMeters(cms);

   printf("%f feet equals %f inches\n",feet,inches);
   printf("%f feet equals %f centimeters\n",feet,cms);
   printf("%f feet equals %f meters\n",feet,meters);

   return 0;
}

double feetToInches(double feet)
{
   return 12.0 * feet;
}

double inchesToCentimeters(double inches)
{
   return 2.54 * inches;
}

double centimetersToMeters(double cm)
{
   return cm/100.0;
}
```

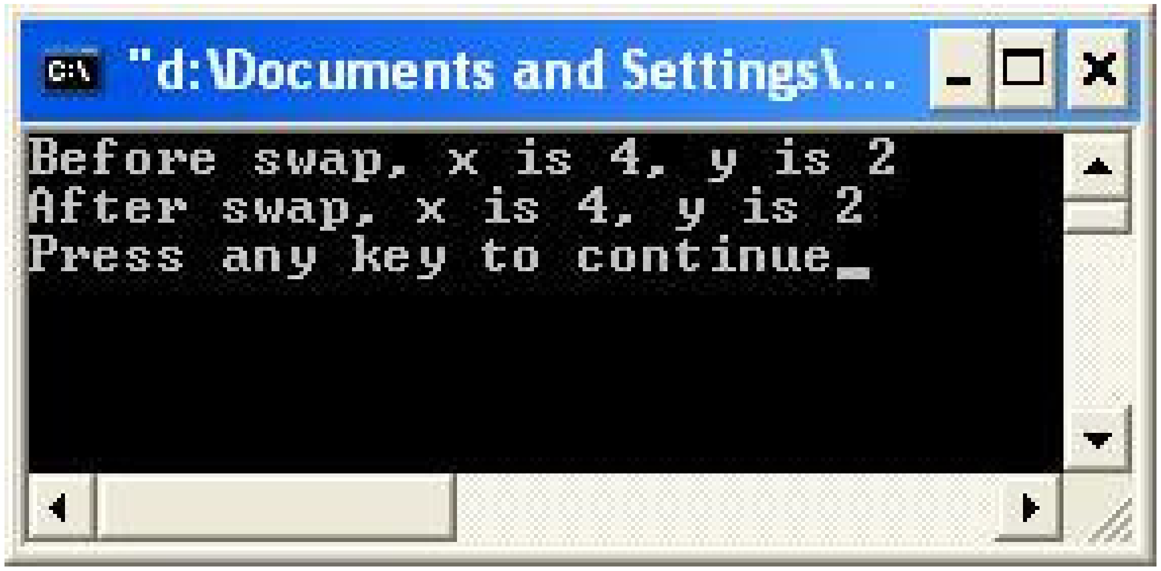
Returning Multiple Values from Functions

Function to swap two integer values.

```
#include <stdio.h>

void swap(int x, int y);
        /* Note that the variable names in the
           prototype and function
           definition need not match.
           Only the types and number of
           variables must match */

int main()
{
    int x = 4;
    int y = 2;

    printf("Before swap, x is %d, y is
%d\n",x,y);
    swap(x,y);
    printf("After swap, x is %d, y is %d\n",x,y);

}

void swap(int first, int second)
{
    int temp;

    temp = second;
    second = first;
    first = temp;
}
```

Returning Multiple Values from Functions

Results



```
Before swap, x is 4, y is 2
After swap, x is 4, y is 2
Press any key to continue_
```

The values weren't swapped.

Returning Multiple Values from Functions

In C, all arguments are passed into functions by value.

This means that the function receives a local copy of the argument. Any modifications to the local copy do not change the original variable in the calling program.

If a variable is to be modified within a function, and the modified value is desired in the calling routine, a pointer to the variable should be passed to the function.

The pointer can then be manipulated to change the value of the variable in the calling routine.

It is interesting to note that the pointer itself is passed by value. The function cannot change the pointer itself since it gets a local copy of the pointer. However, the function can change the contents of memory, the variable, to which the pointer refers.

The advantages of passing by pointer are that any changes to variables will be passed back to the calling routine and that multiple variables can be changed.

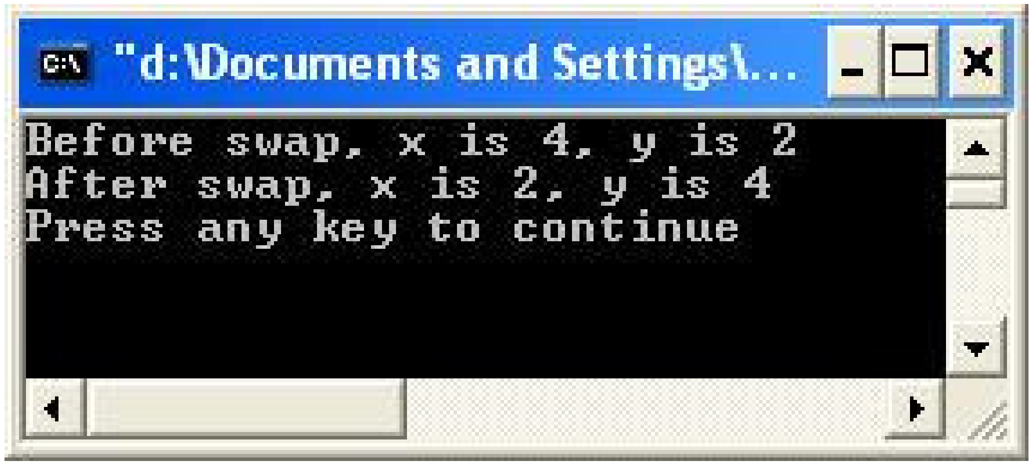
Returning Multiple Values from Functions

Example with pointers being passed into the function.

```
#include <stdio.h>

void swap(int *x, int *y);

int main()
{
   int x = 4;
   int y = 2;

   printf("Before swap, x is %d, y is
%d\n",x,y);
   swap(&x,&y);
   printf("After swap, x is %d, y is
%d\n",x,y);

   return 0;

}

void swap(int *first, int *second)
{
   int temp;

   temp = *second;
   *second = *first;
   *first = temp;
}
```

Returning Multiple Values from Functions

Results



```
Before swap, x is 4, y is 2
After swap, x is 2, y is 4
Press any key to continue
```

Practice Problem

1) Write a function that will calculate the area and circumference of a circle. Write a program to prompt a user for a radius and write out the values calculated by the function.

Hint: Pass values that will be modified by pointer.

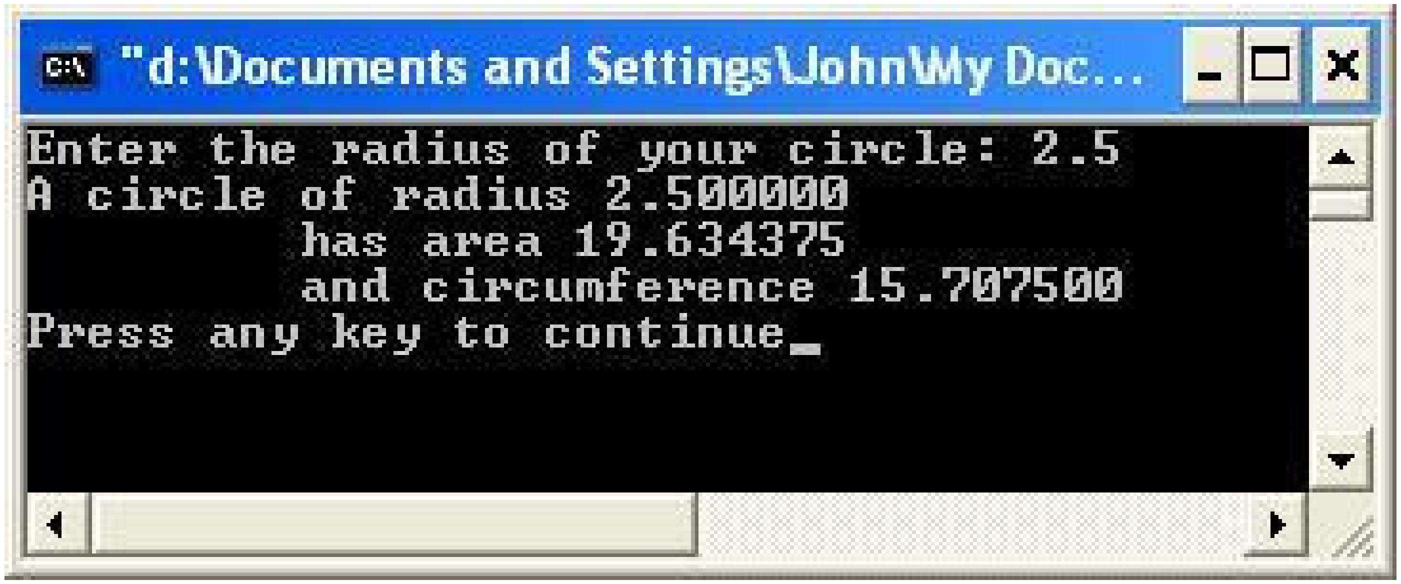
Useful facts:

pi = 3.14
area = pi * radius2
circumference = 2 * pi * radius

Solution

1)
```
#include <stdio.h>
#define PI 3.1415

void calcCircle(float radius, float *area, float *circum);
int main()
{

   float radius;
   float area;
   float circum;
   float *apt;

   printf("Enter the radius of your circle: ");
   scanf("%f",&radius);

   apt = &area;
   calcCircle(radius,apt,&circum);
   /* Notice that for the area an explicit pointer is passed.
      For the circumference, the address of operator is used.
      Passing in the address of a variable is the same as passing
      a pointer */

   printf("A circle of radius %f\n \t has area %f\n \t and
circumference %f\n",
         radius, area, circum);

   return 0;

}

void calcCircle(float radius, float *area, float *circum)
{
   float pi = PI;

   *area = pi * radius * radius;
   *circum = 2.0 * pi * radius;
}
```

Results



```
Enter the radius of your circle: 2.5
A circle of radius 2.500000
         has area 19.634375
         and circumference 15.707500
Press any key to continue_
```

Scope & Program Structure

Local Variable

The scope of a variable is simply the part of the program where it may be accessed or written.

It is the part of the program where the variable's name may be used. If a variable is declared within a function, it is local to that function.

Variables of the same name may be declared and used within other functions without any conflicts.

```
int fun1()
{
   int a;
   int b;
   ....
}

int fun2()
{
   int a;
   int c;
   ....
}
```

Scope & Program Structure

Local Variable

The scope of a variable is simply the part of the program where it may be accessed or written.

It is the part of the program where the variable's name may be used. If a variable is declared within a function, it is local to that function.

Variables of the same name may be declared and used within other functions without any conflicts.

```
int fun1()
{
   int a;
   int b;
   ....
}

int fun2()
{
   int a;
   int c;
   ....
}
```

Here, the local variable "a" in fun1 is distinct from the local variable "a" in fun2.

Changes made to "a" in one function have no effect on the "a" in the other function.

"b" exists and can be used only in fun1.

"C" exists and can be used only in fun2.

The scope of b is fun1.

The scope of c is fun2.

Scope & Program Structure

here, a, b, x, y and z are local to main, a and b are local to fun1 and a and c are local to fun2.

There are three distinct local variables all named "a".

Local variables are also referred to as automatic variables.

They come to life at the beginning of a function and die at the end automatically.

```
int fun1();
int fun2();

int main()
{
   int a;
   int b;
   int x,y,z;

   x = fun1();
   y = fun2();

   return 0;
}

int fun1()
{
   int a;
   int b;
   ....
}

int fun2()
{
   int a;
   int c;
}
```

Scope & Program Structure

External Variable

Variables may also be defined outside of any function.

These are referred to as global or external variables.

The scope of an external variable is from its declaration to the end of the file.

```
int j;
...
int main()
{
   ....
}

int k;
float funA()
{
}

int l;
float funB()
{
}
```

The variable "j" will be visible in main, funA and funB.

The variable "k" will be visible in funA and funB only.

The variable "l" will be visible only in function funB.

Scope of Function

The scope for a function is similar to that of an external variable.

Its scope is from the functions declaration to the end of the file.

here fun1 and fun2 may be called from main but fun3 may not.

The function fun3 can be called from fun1 and fun2 or from anywhere after its declaration.

```
int fun1();
int fun2();
int main()
{
    ....
}
int fun3();

int fun1()
{
    int i;
    ....
    i = fun3();
    ....
}

int fun2()
{
    .....
}
int fun3()
{
    .....
}
```

Multiple Files

Till now all the examples have shown an entire program within one file.

This is fine for short simple programs but any large real world program is likely to have its code distributed among multiple files.

There are several practical reasons for organizing a program into multiple files.

First, it allows parts of the program to be developed independently, possible by different programmers. This separation also allows independent compiling and testing of the modules in each file.

Second, it allows greater reuse. Files containing related functions can become libraries of routines that can be used in multiple programs.

Multiple Files

Having a program distributed in multiple files raises some important issues.

Within one file the scope of a global (external) variable is from its definition to the end of the file.

The keyword "extern" makes it possible to use a global variable in multiple files.

Suppose a program is in three files and it is desired to use a global variable, myState, to communicate some information between routines in each of the files.

The variable, myState, is defined in one file and declared in the other two files using the extern keyword.

Multiple Files

Notice that a variable may only be defined once.

It may be declared multiple times. Remember that a definition creates space, or reserves memory, for the variable.

The declaration just informs the compiler that a variable or function exists.

The keyword extern tells the compiler that the variable is defined in another file.

```
File One:
int myState;
int main()
{
    ....
}

File Two:
extern int myState;
float foo1()
{
    ....
}
float foo2()
{
    ....
}
.....

File Three:
extern int myState;
float fooA()
{
    .....
}
....
```

Multiple Files

The second issue with multiple files is that of function declarations, or prototypes.

Function prototypes should always be used even if the compiler used does not enforce their use.

They allow the compiler to perform type checking on the arguments in function calls and can prevent many errors.

Suppose a function that is defined in one file is to be called in several others.

Each of these other files should contain a prototype for this function. One approach would be as follows.

Multiple Files

```
fileone.c:
int fun1(float x, float y);
int main()
{
   float x,y,z;
   x = 5.0;
   y = 6.0;
   z = fun1(x,y);
   ....
}

filetwo.c:
int fun1(float x, float y);
float fun2()
{
   float a = 4.0;
   float b = 6.0;
   float c;
   c = fun1(a,b);
   ....
}
.....

filethree.c:
int fun1(float x, float y);
....
....
float fun1(float x, float y)
{
   return (x * y);
}
....
```

Include Files and Program Structure

Suppose we had many functions and hated typing.

Fortunately, there is a simple solution, "include" files.

All function prototypes and global variables can be put into an include file, and code in this file will be sourced into each file using the #include preprocessor directive.

```
Include File: filethree.h
int fun1(float x, float y);
......
fileone.c:
#include "filethree.h"
int main()
{
   float x,y,z;
   x = 5.0;
   y = 6.0;
   z = fun1(x,y);
   ....
}
filetwo.c:
#include "filethree.h";
float fun2()
{
   float a = 4.0;
   float b = 6.0;
   float c;
   c = fun1(a,b);
   ....
} ....

filethree.c:
#include "filethree.h";
....
....
float fun1(float x, float y)
{
   return (x * y);
} ....
```

Include Files and Program Structure

Putting the main program in a single file and related functions in their own files is recommended.

So, you might have a large program divided into several files such as main.c, input.c, output.c and calc.c.

Here, main.c would contain the main program. Input functions would be in input.c. Output functions would be in output.c and calculation functions in calc.c.

Each of the "function" files should have its own include file containing the prototypes of all the functions in that file. So in this example, you would also have the include files input.h, output.h and calc.h.

Assume that main uses functions from all three files and that calc.c requires routines from output.c to log error messages.

Include Files and Program Structure

```
main.c:
#include "input.h"
#include "output.h"
#include "calc.h"
main()
{
   ....
}

input.c:
function1() /* Functions can, of course, have any name */
{
}
function2()
{
}

output.c:
functionA()
{
}
functionB()
{
}

calc.c:
#include "output.h"
functiony()
{
}
functionz()
{
}
```