LESSON 3

FUNCTION

FUNCTION CALL BY VALUE

Structured programming is a problem solving strategy and a programming methodology that includes the following two guidelines:

1. The flow of control in a program should be as simple as possible.
2. The construction of a program should embody top-down design.

Top-down design, also referred to as stepwise refinement, consists of repeatedly decomposing a problem into smaller problems. Eventually, one has a collection of small problems or tasks, each of which can be easily coded.

The function construct in C is used to write code for the bottom most directly solvable problems. These functions are combined into other functions and ultimately used in main( ) to solve the original problem. The function mechanism is provided in C to perform distinct programming tasks. Some functions, such as printf( ) and scanf( ) , are provided by the system. Others can be written by the programmer.

A program is made up of one or more functions, with one of these being main( ). Program execution always begins with main( ). When program control encounters a function name, the function is called, or invoked. This means that program control passes to the function. After the function does its work, program control is passed back to the calling environment, which then continues with its work. As a simple example, consider the following program, which prints a message

```
main ( )
{
  void prn_message( ); /* function declaration */
  prn_message( );  /*function invocation */
}
void prn_message( )  /* function definition */
{
  printf ("Message for you: ");
  printf("\nHave a nice dayl\n");
}
```

Execution begins in main( ). When program control encounters prn_message( ), the function is invoked and program control is passed to it. After the two printf( ) statements in prn_message( ) have been executed, program control passes back to the calling environment, which in this example is main( ). Since there is no more

University of Houston Clear Lake

work to be done in main( ), the program ends.

The C code that describes what a function does is called the function definition. It has the following general form

```
type function_name(parameter list)
declaration of parameters
{
local variable declarations statement
}
```

Everything before the first brace comprises the header of the function definition, and everything between the braces comprises the body of the function definition. In the function definition for prn_message( ) above, the parameter list is empty, so there are no declarations of parameters. The body of the function consists of two printf( ) statements. Since the function does not return a value, the type of the function was specified as void. The type of a function depends on the type of the value that the function returns. If an int value is returned, the type of the function can be omitted. It is assumed to be an int. This is poor form however, and the type should be declared even if it is an int. If the function does not return a value, it is said to be a void function and this type should always be declared. While some older compilers may not complain when a void function is not declared, the newer ANSI compilers will often require the declaration.

Parameters are syntactically identifiers, and they can be used within the body of the function. Sometimes the parameters in a function definition are called formal arguments to emphasize their role as place holders for actual values that are passed to the function when it is called. Upon function invocation, the value of the argument corresponding to a formal parameter is used within the body of the executing function.

To illustrate these ideas, let us rewrite the above program so that prn_message( ) has a formal parameter. The parameter will be used to specify how many times the message is printed.

```
main( )
{
void prn_message( );
int ; ("Input a small positive integer:
");
scanf("%d", &n);
prn__me ssage(n);
}
void prn_message(k)
```

```
int k;
{
int i;
printf ("Message for you:\n");
for (i = 0;i <k; ++i)          ll
printf(" Have a nice day!\n");
}
```

PROGRAM ANALYSIS

```
main(   )

void prn_message0:
int n;
printf ("Input a small positive integer:   ");
scant ("%d", &n);
```

The function prn_message( ) is declared prior to its invocation as to its return type. This type is void because it does not return any data. Even though the formal definition of the function will include parameters it is not necessary to include them inside the parenthesis of the declaration. The variable n is declared to be an int. The function printfO is used to prompt the user for a small integer. The function scanf( ) is used to store the value typed in by the user in the variable n.

```
prn_message(n);
```

This statement causes the function prn_message( ) to be called. The value of n is passed as an argument to the function.

```
void prn_message(k)
int k;
```

This is the header of the function definition for prn_message( ). The identifier k is a parameter that is declared to be of type int. One can think of the parameter k as representing the value of the actual argument that is passed to the function when it is called. A call to this function occurred in main( ) in the statement

```
prn_message(n);
```

In main( ) suppose that the value of n is 2. Then when program control passes to prn_message( ), the variable k will have the value 2. Since no value is returned by

the function, we specify void as the function type.

```
{
   int i;
   printf("Message for you:\n");
   for(i = 0; i <k; ++i)
       printf ("Have a nice day!\n");
}
```

This is the body of the function definition for prn_message( ). If we think of k as having the value 2, then the message is printed twice. When program control reaches the end of the function, control is passed back to the calling environment.

Be aware and note that parameters and local variables used in one function definition have no relation to those in another. For example, if the variable i had been used in main( ), it would have had no relationship to the variable i used in prn_message( ).

The return statement

The return statement is used for two purposes. When a return statement is executed, program control is immediately passed back to the calling environment. In addition, if an expression follows the keyword return, then the value of the expression is returned to the calling environment as well. This value must agree in type with the function definition header. If no type is explicitly declared, the type is implicitly int. A return statement has one of the following two forms:

```
return;
return expression;
return (expression);
```

Some examples are

```
return 4;
return (3);
return (a + b);
```

Although it is not necessary, it is considered good programming practice to enclose in parentheses the expression being returned so that it is more clearly visible. As an example let us write a program that computes the minimum of two integers.

```
main( )
{
    int min(int x, int y);
    int j, k, m;
```

```
    printf ("Input two integers: ");
    scant ("%d%d", &j,&k);
    m = min(j , k) ;
    printf ("\n%d is the minimum of %d and %d\n\n", m, j, k);
}
int min(int x, int y)
{
  if (x < y)
     return (x);
  else
     return (y);
}/*end of min */
```

PROGRAM ANALYSIS

```
main(   )
{
int min(int x, int y);
int j, k, m;
printf ("Input two integers: ");
scanf("%d%d", &j, &k);
```

The variables j, k and m are declared to be of type int. The function prototype min(int x, int y) is declared to return an int. The user is asked to input two integers. The function scanf( ) is used to store the values in j and k.

```
m = min(j, k);
```

The values of j and k are passed as arguments to min(int x, int y). The function min(int x, int y) is expected to return a value, and that value is assigned to m.

```
printf("\n%d is the minimum of %d and %d\n\n", m, j, k);
```

The values of m, j , and k are printed out.

```
int min(int x, int y)
```

This is the header of the function definition for min(int x, int y). The parameter list consists of x and y. They are declared to be of type int.

```
{
        if(x<y)
           return (x);
        else
           return (y);
```

```
}
```

This is the body of the function definition for min(int x, int y). If the value of x is less than the value of y, then the value of x is returned to the calling environment; otherwise the value of y is returned.

Even a small function such as min(int x, int y) provides useful structuring to the code. If you want to write a function max(int x, int y) that computes the maximum of two values, then you can copy min(int x, int y) and modify it slightly. Let us write that function as well.

```
int max(int x, int y)
{
  if(x>y)
     return (x);
  else
     return (y);
}
```

These examples were designed min( ) and max( ) to work with integer values. Suppose instead that we want these functions to work with values of type double. We will rewrite min( ) and leave the rewriting of max( ) as an exercise.

```
double min(double x, double y)
{
if (x  <  y)
return (x)
else
return (y)
}
```

Since min( ) now returns a value of type double, you must put this type in the header of the function definition, just before the name of the function. Moreover, other functions that make use of min( ) must declare it as a function that returns a double. For example the function prototype must be declared in main,

```
main( )
{
  double min(double x, double y);
  double min(double x, double y);
  double  x, y, z;
}
```

The declaration in main( ) tells the compiler that min( ) and max( ) are functions that return a value of type double.