Unit 3 Functions in C+ +

Introduction

We know that functions play an important role in C program development. Dividing a program into functions is one of the major principles of top-down, structured programming. Another advantage of using functions is that it is possible to reduce the size of a program by calling and using them at different places in the program.
Recall that we have used syntax similar to the following in developing C programs.

```
void show( );   /* Function declaration */
void main( )    /* Function definition */
{
        … …          /*  Function  call*/
        show( );
        … …
}

void show( )          /* Function definition */
{
        … …          /*  Function  body */
        … …
}
```

When function is called, control is transferred to the first statement in the function body. The other statements in the function body are then executed and control returns to the main program when the closing brace is encountered. C++ is no exception. ***Functions*** continue to be the building blocks of C++ programs. In fact, C++ has added many new features to functions to make them more reliable and flexible. Like C++ operators, a C++ functions can be overloaded to make it perform different tasks depending on the arguments passed to it. Most of these modifications are aimed at meeting the requirements of object-oriented facilities.

The Main Function

C does not specify any return type for the **main ( )** function which is the starting point for the execution of a program. The definition of **main ( )** would look like this:

```
main ( )
{
        … …
        … …   // main  program statements

}
```

This is perfectly valid because the **main ( )** in **C** does not return any value.

In **C++,** the **main ( )** returns a value of type **int** to the operating system. C++, there for explicitly defines **main ( )** as matching one of the following prototypes:

```
int main ( );
int main (int argc,   char *argv[ ]);
```

The functions that have a return value should use the return statement for termination. The **main ( )** function in C++ is therefore, defined as follows:

Since the typo of functions is **int** by default, the keyword int in the **main ( )** header is optional. Most C++ compilers will generate an error or warning if there is no **return** statement. **Turbo C++** issues the warning

Functi o n should return a value

And then proceeds to compile the program. It is good programming practice to actually to return a value from **main ( ).**

Many operating systems test the return value (called ***exit value)*** to determine if there is any problem. The normal conversion is that an exit value of zero means the program ran successfully, while a nonzero value means there was a problem. The explicit use of a **return (0)** statement will indicate that the program was successfully executed.

Function Prototyping

Function prototyping is one of the major improvements added to C++ functions. The prototype describes the function interface to the compiler by giving details such as the number and arguments and the type of return values. With function prototyping, a ***template*** is always used when declaring and defining a function. When a function is called, the compiler uses template to ensure that proper arguments are passed, and the return value is treated correctly. Any violation in matching the arguments or the return types will be caught by the compiler at the time of compilation itself. These checks and controls did not exist in conventional C functions

Remember, C also uses prototyping. But it was introduced first in C++ by Stroustrup and the success of this feature inspired the ANSI C committee to adopt it. However, there is a difference in prototyping between C and C++. While C++ makes the prototyping essential, ANSI C makes it optional, perhaps, to preserve the compatibility with classic C.

Function prototype is a *declaration statement* in the calling program and is of the following form:

type   function-name (argument-list);

*The **argument-list*** contains the types and names of arguments that must be passed to the function.

Example: :,

float volume (int x, float y, float z);

Note that each argument variable must be declared independently inside the parentheses. That is, combined declarations like

float volume (int x, float y, z);

is illegal.

In a function declaration, the names of the arguments are *dummy* variables and therefore, they are optional. That is, the form

float volume (int, float, float);

is acceptable at the place of declaration. At this stage, the compiler only checks for the type of arguments when the function is called.

In general, we can either include or exclude the variable names in the argument of prototypes. The variable names in the prototype just act as placeholders and, therefore if names are used, they don't have to match the names used **in *the function call*** *or* ***function definition***.

In the function definition, names are required because the arguments must be referenced inside the function.

Example:

```
float   volume (int  a, float  b, float  c)
{
        float v = a*b*c;
}
```

The function **volume( )** can be invoked in a program as follows:

float cubel = volume (bl, wl, hl); // Function call

The variable bl, w1, and h1 are known as the actual parameters which specify the dimensions of cubel. Their types (which have been declared earlier) should match with types declared in the prototype. Remember, the calling statement should not include names in the argument list.

We can also declare a function with an ***empty argument*** *list,* as in the following example

void display( );

In **C++,** this means that the function does not pass any parameters. It is identical to the statement

void display(void);

However, in **C**, an empty parenthesis implies any number of arguments. That is we have foregone prototyping. A **C++** function can also have an 'open' parameter list by the use of **ellipses** in the prototype as shown below:

void do_something (...)

Call by Reference .

In traditional C, a function call passes arguments by value. The *called function* creates a new set of variables and copies the values of arguments into them. The function does not have access to the actual variables in the calling program and can only work on the copies of values. The mechanism is fine if the function does not need to alter the values of the original variables in the calling program. But, there may arise situations where we would like to change the values of variables in the *calling program.* For example, in bubble sort, we compare two adjacent elements in the list and interchange their values if the first element is greater than the second. If a function is used for *bubble* sort, then it should be able to alter the values of variables in the calling function, which is not possible if the call-by value method is use.

Provision of 'the ***reference variables*** in C++ permit us to pass parameters to the functions by reference. When we pass arguments by reference, the 'formal' arguments in the called function become aliases to the 'actual' arguments in the calling function. This means that when the function is working with its own arguments, it is actually working on the original data. The following function explains this concept.

```
void swap (int &a, int &b)   // a and b are reference variables
{
        int t = a;
        a = b;
        b = t;
 }
```

Now if m and n are two integer variables, then the function call

swap(m ,n);

will exchange the values of m and n using their aliases (reference variables) a and b. In traditional **C**, this is accomplished *using* ***pointers*** **and** ***indirection*** as follows:

```
void swap (int *a,  int *b)    /* Function definition */

{

        int t;
        t = *a;              /'* assign the value at address a to t */
        *a = *b;             /* put the value at b into a */
        *b = t;              /* put the value at t into b */
 }
```

This function can be called as follows:

```
swap (&x,   &y);          ./* call   by passing */
                            /* addresses of variables */
```

This approach is also acceptable in C++. Note that the **call-by-reference** method is neater in its approach.

Return by Reference

A function can also return a **reference**. Consider the following function:

```
int & max(int &x, int &y)

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

Since the return type of' max ( ) is int &, the function returns reference to x or y (and not the values). Then a function call such as **max (a, b)** will yield a reference to either a or b depending on their values. This means that this function call can appear on the left-hand side of an assignment statement. That is, the statement

```
max (a, b) = -1;
```

is legal and assigns 1 to a if it is larger, otherwise -1 to b.

Inline Functions

One of the objectives of using functions in a program is to save some memory space, which becomes appreciable when a function is likely to be called many times. However, every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function. When a function is small, a substantial percentage execution time may be spent in such overheads.

One solution to this problem is to use macro definitions, popularly known as *macros*. **Preprocessor macros** are popular in C. The major drawback with macros is that they are not really functions and therefore, the usual error checking does not occur during compilation.

C++ has a different solution to this problem. To eliminate the cost of calls to small functions, C++ proposes a new feature called ***inline function***. An inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function code (something similar to **macros expansion**). The inline **functions are** defined as follows:

```
inline function-header
{
        function body
}
```

Example:

```
inline double cube(double a)

{
    return (a * a * a);

}
```

The above inline function can be invoked by statements like

c = cube (3.0);

d = cube (2.5 + 1.5);

On the execution of these statements, the values of c and d will be 27 and 64 respectively. If the arguments are expressions such as 2.5 + 1.5.the function passes the value of the expression, 4 in this case. This makes the inline feature far superior to macros.

It is easy to make a function inline. All we need to do is to prefix the keyword **inline** to the function definition. All inline functions must be defined before they are called.

We should exercise care before making a function inline. The speed benefits of **inline** functions diminish as the function grows in size. At some point the overhead of the function call becomes small compared to the execution of the function, and the benefits of **inline** functions may be lost. In such cases, the use of normal functions will be more meaningful. Usually, the functions are made inline when they are small enough to be defined in one or two lines.

Example:

inline double cube (doub1e a) (return (a*a*a);}

Remember that the **inline keyword** merely sends a request, not a command, to the compiler. The compiler may ignore this request if the function definition is too long or too complicated and compile the function as a normal function.

Some of the situations where inline expansion may not work are:

1. **For functions returning values, if a loop, a switch, or a goto exists.**
2. **For functions not returning values, if a return statement exists.**
3. **If functions contain static variables.**
4. **If inline functions are recursive.**

Note: Inline expansion makes a program run taster because the overhead of a function call: return is eliminated. However, it makes the program to take up more memory because the statements that define the inline function are reproduced at each point where the function is called. So, a trade-off becomes necessary.

Following Program illustrates the use of **inline** functions.

```
#include <iostream>
 using namespace std;

        inline float mul (float x,  float y)
        {
                return (x*y);
        }
```

```
inline double div (double p,  double q)
{
        return (p/q);
}

int main ( )
{
        float a =  12.345; float b = 9.82;
        cout « mul (a, b)  « "\n";
        cout « div (a, b) « "\n";
        return 0;
}
```

The output of above program would be

```
121.228
1.25713
```

- **Default Arguments**

C++ allows us to call a function without specifying all its arguments. In such cases the function assigns a default value to the parameter which does not have a matching argument in the function call. Default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values. Here is an example of a prototype (i.e. function declaration) with **default values**:

```
float amount (f1oat  principal, int  period, float  rate=0.15);
```

The **default value** is specified in a manner syntactically similar to a variable initialization. The above prototype declares a default value of 0.15 to the argument rate. The call

```
value = amount (500, 5, 0.12);          // No missing argument
```

passes an explicit value of 0.12 to rate.

A **default** argument is checked for type at the time of declaration and evaluated at the time of call. One important point to note is that only the trailing arguments can have **default** values and therefore we must add defaults from right to left.

We cannot provide a default value to a particular argument in the middle of an argument list. Some examples of function declaration with default values are:

```
int mul (int i,       int j=5, int k =10);          //      legal
int  mul (int i =5,   int j);                       //      illegal
int  mul (int i=0,    int j,     int  k=10);        //      illegal
int  mul (int i=2,    int j=5,   int  k=10):        //      legal
```

Default arguments are useful in situations where some arguments always have the same value. For instance, bank interest may remain the same for all customers for a particular period of deposit. It also provides a greater flexibility to the programmers. A function can be written with more parameters than are required for its most common application. Using **default arguments**, a programmer can use only those arguments that are meaningful to a particular situation.

Following program illustrates the use of **default arguments**.

```
#include  <iostream>
using namespace std;
 int main( )
 {
        float amount;
        float value (float p,   int n,  float  r=0.15);          // prototype
        void  printine(char ch ='*',   int  len=40);             // prototype
        printline();                                             // uses  default  values  for arguments
        amount  =  value(5000.00,5);                  ,// default for 3rd argument
        cout « "\n          Final   Value =  " « amount «  "\n\n";
        printline('=');                                  // use default value for 2nd argument
        return 0;
 }

 float value (float p,   int n,   float  r)
 {
        int year =1;
        float sum = p;
        while (year <=  n)
        {
              sum = sun'(1+r);
              year = year+1;
        }
        return (sum);
  }

 void printline (char ch, int len)
 {
        for (int   i = l;   i <=len   ;i++)
              printf ("%c", ch);
         printf("\n");
  }
```

The output of Program would be :

```
Final   Value =  10056.8
```

Advantages of providing the default arguments are:

1. **We can use default arguments to add new parameters to the existing functions.**
2. **Default arguments can be used to combine similar functions into one.**

- **const Arguments**

In C++, an argument to a function can be declared as **const** as shown below.

```
int  strlen(const char *p);

int   length(const string &s);
```

The qualifier **const** tells the compiler that the function should not modify the argument. The compiler will generate an error when this condition is violated. This type of declaration is significant only when we pass arguments by reference or pointers.

- **Function Overloading**

As stated earlier, ***overloading*** refers to the use of the same thing for different purposes. **C++** also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as ***function polymorphism*** in **OOP**.

Using the concept of function **overloading;** we can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type.

For example, **an overloaded add()** function handles different types of data as shown below.

```
//        Declarations
int add(int a, int b);                    //      prototype 1
int add(int a, int b, int c);             //      prototype 2
double add(double x, double y);           //      prototype 3
double add(int p, double q);              //      prototype 4
double add(double p, double q);           //      prototype 5
// Function Calls
cout << add (5, 10);                      //      Use prototype 1
cout << add (15, 10.0);                   //      Use prototype 4
cout << add (12.5, 7.5);                  //      Use prototype 3
cout << add (5, 10, 15);                  //      Use prototype 2
cout  << add (0.75, 5);                   //      Use prototype 5
```

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution. A best match must be unique, function selection involves the following steps:

1. **The compiler first tries to find an exact match in which the types of actual argument are the same, and use that function.**

2. **If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as,**

   char to int

   float to double

   to find a match.

3. **When either of them fails, the compiler tries to use the built-in conversions (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error message. Suppose we use the following two functions:**

   long square (long n)

   double square(double x)

   A function call such as

   square (10)

   will cause an error because int argument can be converted to either long or double thereby creating an ambiguous situation as to which version of square( ) should be used.

4. **If all of the steps fail, then the compiler will try the user-defined conversions combination with integral promotions and built-in conversions to find a unique match. User-defined conversions are often used in handling class objects.**

Following program illustrate function overloading.

```
// Function volume() is overloaded three times

#include<iostream>
using namespace std;
// Declarations prototypes.
int volume (int);
double volume (double, int);
long volume(long ,int , int);
```

```
int main ( )
{
        cout << volume (10) << "\n";
        cout << volume(2.5, 8) << "\n";
        cout << volume(100L,75, l5) <<"\n";
        return 0;
}
// Function definitions :
int volume (int s)              // cube
{
        return (s * s * s);
}

double volume(double r, int h)      //cylinder
{
        return (3.14519 * r * r * h);
}

long volume(long l, int b, int h)      //rectangular box
{
        return (l * b * h);
}
```

The output of above program would be :

1000

157.26

112500

Overloading of the function should be done with caution. We should not overload unrelated functions and should reserve function overloading for functions that perform closely related tasks. Sometimes, the default arguments may *be* used instead of overloading. This may reduce the number of functions to be defined.

Overloaded functions are extensively used for handling class objects.