Chapter 6

FUNCTIONS AND POINTERS

Most computer programs that solve real-world problems are much larger than the programs in the first few chapters. Experience has shown that the best way to develop and maintain a large program is to construct it from smaller pieces or program units each, of which is more manageable than the original program. Generally, user-defined program units are called *subprograms*. This technique is called *divide-and-conquer*.

This chapter deals with the method of declaration of the user-defined function and its use in C++. Besides, this chapter also covers pointer data type which is one of the strength of the C++ language.

6.1 FUNCTION AND PARAMETER DECLARATIONS

In C++ all subprograms are referred to as functions. A function allows you to treat a related group of C++ statements as a single unit. The programmer can write functions to define specific tasks that could be used at many points in a program.

Functions allow the programmer to modulize a program. All variables declared in function definitions are *local variables* – they are known only in the function in which they are defined. Most functions have a list of parameters that provide the means for communicating information between functions.

There are several motivations for dividing a program into functions. The *divide-and-conquer* approach makes program development more manageable. Another motivation is software reusability – using existing functions as building blocks to create new programs. Software reusability is a major factor in object-oriented programming. With good function naming and definition, programs can be created from standardized functions that accomplish specific tasks. A third motivation is to avoid repeating code in a program. Packing code as a function allows the code to be executed from several location in a program simply by calling the function.

Defining a Function

The lines that compose a function within a C++ program are called a *function definition*. The syntax for defining a function is

```
data_type name_of_function (parameters){
    statements;
}
```

A function definition consists of four parts:

- A reserved word indicating the return data type of the function's return value.
- The function name
- Any parameters required by the function, contained within parentheses.
- The function's statements enclosed in curly braces { }.

Example: The following function determines the largest integer among three parameters passed to it.

```
int maximum( int x, int y, int z )
{
  int max = x;
  if ( y > max )
    max = y;
  if ( z > max )
    max = z;
  return max;
}
```

You designate a data type for function since it is common to return a value from a function after it executes.

General format of a function is describe in Figure 6.1

Variable names that will be used in the function header line are called *formal parameters*.

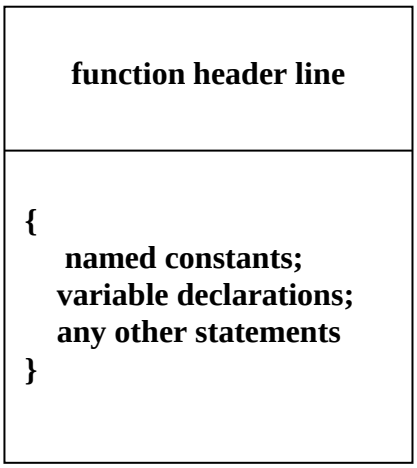

Figure 6.1 General Format of a function

How to Call Functions

To execute a function, you must invoke, or call, it from the *main()* function.

The values or variables that you place within the parentheses of a function call statement are called *arguments* or *actual parameters*.

Example: Function *maximum* is invoked or called in *main()* with the call

maximum(a, b, c)

Function Prototypes

One of the most important features of C++ is the function prototype. A *function prototype* declares to the compiler that you intend to use a function later in the program. It informs the compiler the name of the function, the type of data returned by the function, the number of parameters the function expects to receive, the types of the parameters and the order in which these parameters are expected. The compiler uses function prototypes to validate function calls.

If you try to call a function at any point in the program prior to its function prototype or function definition, you will receive an *error* when you compile the project.

Example 6.1.1

```
// Finding the maximum of three integers
#include <iostream.h>
int maximum(int, int, int);   // function prototype
int main()
{
  int a, b, c;
  cout << "Enter three integers: ";
  cin >> a >> b >> c;

  // a, b and c below are arguments to the maximum function call
  cout << "Maximum is: " << maximum (a, b, c) << endl;
  return 0;
}

// Function maximum definition
// x, y and z are parameters to the maximum function definition
int maximum( int x, int y, int z)
{
  int max = x;
  if ( y > max )
    max = y;
  if ( z > max )
    max = z;
  return max;
}
```

The output of the above program:

```
Enter three integers: 22  85  17
Maximum is:  85
```

Passing by Value

If a variable is one of the actual parameters in a function call, the called function receives a *copy* of the values stored in the variable. After the values are passed to the called function, control is transferred to the called function.

Example: The expression *maximum(a, b, c)* calls the function *maximum()* and causes the values currently residing in the variables *a*, *b* and *c* to be passed to *maximum().*

The method of passing values to a called function is called *pass by value*.

Functions with Empty Parameter Lists

Functions may have empty parameter list. The function prototype for such a function requires either the keyword void or nothing at all between the parentheses following the function name.

Example:

```
int display();
int display(void);
```

6.2 RETURNING VALUES

To actually return a value, the function must use a *return* statement, which has the form:

```
return expression;
```

or

```
return(expression);
```

Remember that values passes back and forth between functions must be of the same data type.

When the *return* statement is encountered, the expression is evaluated first. The value of the expression is then automatically converted to the data type declared in the function header before being sent back to the calling function. After the value is returned, program control reverts to the calling function.

Inline functions

For small functions, you can use the *inline* keyword to request that the compiler replace calls to a function with the function definition wherever the function is called in a program.

Example 6.2.1

```
// Using an inline function to calculate
// the volume of a cube.
#include <iostream.h>
inline double cube(double s) { return s * s * s; }
int main()
{
```

```
  cout << "Enter the side length of your cube:  ";
  double side;
  cin >> side;
  cout << "Volume of cube with side "
       << side << " is " << cube(side) << endl;
  return 0;
}
```

The output of the above program:

```
Enter the side length of your cube:   3.5
Volume of cube with side  3.5  is  42.875
```

Function Overloading

C++ enables several functions of the same name to be defined, as long as these functions have different sets of parameters (at least their types are different). This capability is called *function overloading*.

When an overloaded function is called, the C++ compiler selects the proper functions by examining the number, types and order of the arguments in the call.

Function overloading is commonly used to create several functions of the same name that perform similar tasks but on different data types.

Example:

```
void showabs(int x)
{
  if( x < 0)
    x = -x;
    cout << “The absolute value of the integer is “ << x << endl;
}

void showabs(double x)
{
  if( x < 0)
    x = -x;
    cout << “The absolute value of the double is “ << x << endl;
}
```

The function call

```
showabs(10);
```

causes the compiler to use the first version of the function *showabs*.

The function call

```
showabs(6.28);
```

causes the compiler to use the second version of the function *showabs*.

Default Arguments

C++ allows *default arguments* in a function call. Default argument values are listed in the function prototype and are automatically transmitted to the called function when the corresponding arguments are omitted from the function call.

Example: The function prototype

```
void example (int, int = 5, float = 6.78);
```

provides default values for the two last arguments.

If any of these arguments are omitted when the function is actually called, the C++ compiler supplies these default values.

Thus, all following function calls are valid:

```
example(7, 2, 9.3);              // no default used
example(7, 2);           // same as example(7, 2, 6.78)
example(7);              // same as example(7, 5, 6.78)
```

6.3 VARIABLE SCOPE

Scope refers to where in your program a declared variable or constant is allowed used. A variable can be used only when inside its scope.

Variable scope

Global scope refers to variables declared outside of any functions or classes and that are available to all parts of your program.

Local scope refers to a variable declared inside a function and that is available only within the function in which it is declared.

Example 6.3.1

```
#include <iostream.h>
int x;    // create a global variable named firstnum
void valfun();   // function prototype (declaration)
int main()
{
  int y;    // create a local variable named secnum
  x = 10; // store a value into the global variable
  y = 20;   // store a value into the local variable
  cout << "From main(): x = " << x << endl;
  cout << "From main(): y =  " << y << endl;
  valfun();      // call the function valfun
  cout << "\nFrom main() again: x = " << x << endl;
  cout << "From main() again: y = " << y << endl;
  return 0;
```

```
}

void valfun()   // no values are passed to this function
{
  int y;  // create a second local variable named y
  y = 30; // this only affects this local variable's value
  cout << "\nFrom valfun(): x = " << x << endl;
  cout << "\nFrom valfun(): y = " << y << endl;
  x = 40;    // this changes x for both functions
  return;
}
```

The output of the above program:

```
From main(): x = 10
From main(): y = 20

From valfun(): x = 10
From valfun(): y = 30

From main() again: x = 40
From main() again: y = 20
```

In the above program, the variable *x* is a global variable because its storage is created by a definition statement located outside a function. Both functions, *main()* and *valfun()* can use this global variable with no further declaration needed. The program also contains two separate local variables, both named *y*. Each of the variables named *y* is local to the function in which their storage is created, and each of these variables can only be used within the appropriate functions.

Scope Resolution Operator

When a local variable has the same name as a global variable, all uses of the variable's name within the scope of the local variable refer to the local variable.

In such cases, we can still access to the global variable by using *scope resolution operator* (::) immediately before the variable name.

The :: operator tells the compiler to use the global variable.

Example 6.3.2

```
#include <iostream.h>
float number = 42.8;       // a global variable named number
int main()
{
  float number = 26.4;     // a local variable named number
  cout << "The value of number is " << number << endl;
  return 0;
}
```

The output of the above program:

The value of number is 26.4

Example 6.3.3

```
#include <iostream.h>
float number = 42.8;       // a global variable named number
int main()
{
  float number = 26.4;     // a local variable named number
  cout << "The value of number is " << ::number << endl;
  return 0;
}
```

The output of the above program:

The value of number is 42.8

6.4 VARIABLE STORAGE CLASS.

The lifetime of a variable is referred to as the *storage duration*, or *storage class*.

The four available storage classes are *auto, static, extern* and *register*.

If one of these class names is used, it must be placed before the variable's data type in a declaration statement.

Examples:

```
auto int num;
static int miles;
register int dist;
extern float price;
extern float yld;
```

6.4.1 Local Variable Storage Classes

Local variables can only be members of the *auto, static,* or *register* storage classes.

If no class description is included in the declaration statement, the variable is automatically assigned to the *auto* class.

Automatic Variables

The term *auto* is short for automatic.

Automatic storage duration refers to variables that exist only during the lifetime of the command block (such as a function) that contains them.

Example 6.4.1

```
#include <iostream.h>
int funct(int);    // function prototype
int main()
{
  int count, value;          // count is a local auto variable
  for(count = 1; count <= 10; count++)
     value = funct(count);
     cout << count << '\t' << value << endl;
  return 0;
}

int funct( int x)
{
  int sum = 100;     // sum is a local auto variable
  sum += x;
  return sum;
}
```

The output of the above program:

```
1       101
2       102
3       103
4       104
5       105
6       106
7       107
8       108
9       109
10      110
```

Note: The effect of increasing *sum* in *funct()*, before the function's return statement, is lost when control is returned to *main()*.

In some applications, we want a function to remember values between function calls. This is the purpose of the *static* storage class.

A local *static* variable is not created and destroyed each time the function declaring the static variable is called. Once created, local static variables *remain in existence* for the life of the program.

Example 6.4.2

```
#include <iostream.h>
int funct( int);    // function prototype
int main()
{
  int count, value;          // count is a local auto variable
```

```
  for(count = 1; count <= 10; count++)
     value = funct( count);
     cout << count << '\t' << value << endl;
  return 0;
}

int funct( int x)
{
  static int sum = 100;     // sum is a local auto variable
  sum += x;
  return sum;
}
```

The output of the above program:

```
1       101
2       103
3       106
4       110
5       115
6       121
7       128
8       136
9       145
10      155
```

Since *sum* has a permanent memory space it retains the same value in the period of time between leaving function and again entering it later.
Note:

1. The initialization of *static* variables is done only once when the program is first compiled. At compile time, the variable is created and any initialization value is placed in it. Thereafter, the value in the variable is kept without further initialization each time is called. (compile-time initialization).
2. All static variables are set to zero when no explicit initialization is given.

Register Variables

Register variables have the same time duration as *automatic* variables. The only difference between register and automatic variables is where the storage for the variable is located.

Register variables are stored in CPU's internal registers rather than in memory.

Examples:

```
register int time;
register double difference;
```

6.4.2 Global Variable Storage Classes

Global variables are created by definition statements external to a function. Once a global variable is created, it exists until the program in which it is declared is finished executing.

Global variables may be declared as *static* or *extern* (but not both).

The purpose of the *extern* storage class is to extend the scope of a global variable beyond its normal boundaries. To understand this, we must notice that the programs we have written so far have always been contained together in one file. Thus, when you have saved or retrieved programs, you have only needed to give the computer a single name for your program. Larger programs typically consist of many functions stored in multiple files and all of these files are compiled separately. Consider the following exaple.

Example:

```
//file1
int a;
float c;
static double d;
.
.
int main()
{
    func1();
    func2();
    func3();
    func4();
   .
}
int func1();
{
  .
  .
}
int func2();
{
  .
  .
}
//end of file1

//file2
double b;
int func3();
{
  .
  .
}
```

```
int func4();
{
  .
  .
}
//end of file2
```

Although the variable *a* has been declared in *file1*, we want to use it in *file2*. Placing the statement *extern int a* in *file2*, we can extend the scope of the variable *a* into *file2*.

Now the scope of the variable *a* is not only in *file1*, but also in *func3* and *func4*.

```
//file1
int a;
float c;
static double d;
.
.
int main()
{
    func1();
    func2();
    func3();
    func4();
    .
}
extern double b;
int func1();
{
  .
  .
}
int func2();
{
  .
  .
}
//end of file1

//file2
double b;
extern int a;
int func3();
{
  .
  .
}
int func4();
{
   extern float c;
  .
```

```
 .
}
//end of file2
```

Besides, placing the statement *extern float c;* in *func4()* extends the scope of this global variable, created in *file1*, into *func4(),* and the scope of the global variable *b*, created in *file2*, is extended into *func1()* and *func2()* by the declaration statement *extern double b;* placed before *func1()*.

Note:
1. We cannot make *static* variables *external*.
2. The scope of a *global static* variable cannot extend beyond the file in which it is declared.

6.5 PASS BY REFERENCE USING REFERENCE PARAMETERS

Reference Parameters

Two ways to invoke functions in many programming languages are: call by value and call by reference

When an argument is passed *call by value*, a copy of the argument's value is made and passed to the called function. Changes to the copy do not affect the original variable's value in the caller.

With *call-by-reference*, the caller gives the called function the ability to access the caller's data directly, and to modify that data if the called function chooses so.

To indicate that the function parameter is passed-by-reference, simply follow the parameter's type in the function prototype of function header by *an ampersand* (&).

For example, the declaration

```
int& count;
```

in the function header means "*count* is a reference parameter to an *int*".

Example 6.5.1

```
// Comparing call-by-value and call-by-reference with references.
#include <iostream.h>
int squareByValue( int );
void squareByReference( int & );

int main()
{
  int x = 2, z = 4;
  cout << "x = " << x << " before squareByValue\n"
       << "Value returned by squareByValue: "
       << squareByValue( x ) << endl
```

```
      << "x = " << x << " after squareByValue\n" << endl;
  cout << "z = " << z << " before squareByReference" << endl;
  squareByReference( z );
  cout << "z = " << z << " after squareByReference" << endl;
  return 0;
}
int squareByValue( int a )
{
  return a *= a;   // caller's argument not modified
}
void squareByReference( int &cRef )
{
  cRef *= cRef;    // caller's argument modified
}
```

The output of the above program:

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByReference

z = 4 before squareByReference
z = 16 after squareByReference
```

Since *cRef* is a reference parameter, so, *squareByReference()* now has direct access to the argument *z*. Thus, any change to *cRef* within *squareByReference()* directly alters the value of *z* in *main()*. The assignment of value to *cRef* within *squareByReference()* is relected in *main()* as the altering of *z*'s value.

Recall from Chapter 2 that the ampersand, &, in C++ means "the address of". Additionally, an & symbol used within a declaration refers to "the address of" the preceding data type. Using this information, declaration such as *double& num1* and *int& secnum* are sometimes more clearly understood if they are read backward. Reading the declaration *int& cRef* in this manner yields the information that "*cRef* is the address of an *int* value."

Example 6.5.2

```
// This program can solve quadratic equation
#include <iostream.h>
#include <math.h>
#include <iomanip.h>
in quad( double, double, double, double &, double &);
int main()
{
  double a, b, c, x1, x2;
   int code;
   cout << "Enter the coefficients of the equation: "<< endl;
   cin  >> a >> b >> c;
   code = quad(a, b, c, x1, x2);
   if (code == 1 || code == 2)
     cout << "x1 = " << x1 << setw(20) << "x2 = " << x2 << endl;
```

```
   else if (code == 3)
      cout << " There is no solution " << endl;
   return 0;
}

int quad(double a, double b, double c, double &px1, double &px2)
{
   double del;
   del = b*b – 4.0*a*c;
   if (del == 0.0)
   {
      px1 = -b/(2*a);
      px2 = px1;
      return 1;
    }
   else if (del > 0.0)
   {
      px1 = (-b + sqrt(del))/(2*a);
      px2 = (-b – sqrt(del))/(2*a);
      return 2;
    }
   else
      return 3;
}
```

Note: The called-by-value parameters *a, b,* and *c* are used to passed the data from the calling function to the called function, and the two reference parameters *px1* and *px2* are used to pass the results from the called function to the calling function.

6.6 RECURSION

In C++, it's possible for a function to call itself. Functions that do so are called *seft-referential* or *recursive functions*.

Example: To compute factorial of an integer

1! = 1
n! = n*(n-1)!

Example 6.6.1

```
// Recursive factorial function
#include <iostream.h>
#include <iomanip.h>
unsigned long factorial( unsigned long );
int main()
{
  for ( int i = 0; i <= 10; i++ )
     cout << setw( 2 ) << i << "! = " << factorial( i ) << endl;
  return 0;
```

```
}
// Recursive definition of function factorial
unsigned long factorial( unsigned long number )
{
  if (number < 1)  // base case
    return 1;
  else              // recursive case
    return number * factorial( number - 1 );
}
```

The output of the above program:

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

How the Computation is Performed

The mechanism that makes it possible for a C++ function to call itself is that C++ allocates new memory locations for all function parameters and local variables as each function is called. There is a *dynamic data area* for each execution of a function. This allocation is made dynamically, as a program is executed, in a memory area referred as the stack.

A *memory stack* is an area of memory used for rapidly storing and retrieving data areas for active functions. Each function call reserves memory locations on the stack for its parameters, its local variables, a return value, and the address where execution is to resume in the calling program when the function has completed execution (*return address*). Inserting and removing items from a stack are based on last-in/first-out mechanism.

Thus, when the function call *factorial(n)* is made, a data area for the execution of this function call is pushed on top of the stack. This data area is shown as figure 6.2.

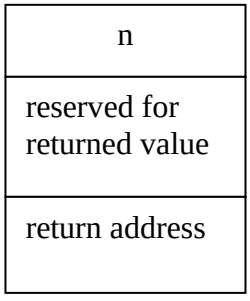

Figure 6.2 The data area for the first call to factorial

The progress of execution for the recursive function *factorial* applied with n = 3 is as follows:

factorial(3) = 3*factorial(2)
= 3*(2*factorial(1))
= 3*(2*(1*factorial(0)))
= 3*(2*(1*1))
= 3*(2*1)
= 3*2
= 6

During the execution of the function call *factorial(3)*, the memory stack evolves as shown in the figure 6.3. Whenever the recursive function calls itself, a new data area is pushed on top of the stack for that function call. When the execution of the recursive function at a given level is finished, the corresponding data area is popped from the stack and the return value is passed back to its calling function.

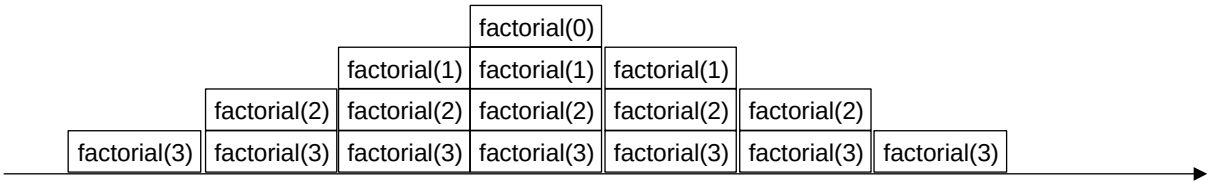

Figure 6.3 The memory stack for execution of function call *factorial(3)*

Example 6.6.2

Fibonacci numbers:

$F_N = F_{N-1} + F_{N-2}$ for $N \geq 2$

$F_0 = F_1 = 1$

```
#include<iostream.h>
int fibonacci(int);
int main()
{
   int m;
   cout <<"Enter a number: ";
   cin>> m;
   cout <<"The fibonacci of "<<m<<" is: "
                << fibonacci(m)<< endl;
   return 0;
}
int fibonacci(int n)
{
        if (n<= 1) return 1;
        return (fibonacci(n-1)+ fibonacci(n-2));
```

}

The output of the above program:

```
Enter a number:  4
The fibonacci of 4 is:  5
```

Recursion vs. Iteration

In this section, we compare recursion and iteration and discuss why the programmer might choose one approach over the other in a particular situation.

Both iteration and recursion are based on a control structure: Iteration uses a repetition structure; recursion uses a selection structure. Both iteration and recursion involve repetition: Iteration explicitly used a repetition structure while recursion achieves repetition through repeated function calls. Iteration and recursion both involve a termitation test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized. Iteration with counter-controlled repetition and recursion both gradually approach termination: Iteration keeps modifying a counter variable until the counter assumes a value that makes the loop-continuation condtion falil; recursion keeps producing simpler versions of the original problem until the base case is reached. Both iteration and recursion can occur indefinitely: An infinite loop occurs with iteration if the loop-continuation test never become false; infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case.

Recursion has many inconveniences. It repeatedly invokes the mechanism, and consequently the overhead, of function calls. This can be expensive in both CPU time and memory space. Each recursive call causes another copy of the function (actually only the function's variables) to be created; this can consume considerable memory. Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment is omitted.

Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more naturally reflects the problem and results in a program that is easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution is not apparent.

6.7 PASSING ARRAYS TO FUNCTIONS

To pass an array to a function, specify the name of the array without any brackets. For example, if array *hourlyTemperature* has been declared as

```
int hourlyTemperature[24];
```

The function call statement

```
modifyArray(hourlyTemperature, size);
```

passes the array *hourlyTemperature* and its size to function *modifyArray*.

For the function to receive an array through a function call, the function's parameter list must specify that an array will be received.

For example, the function header for function *modifyArray* might be written as

```
void modifyArray(int b[], int arraySize)
```

Notice that the size of the array is not required between the array brackets.

Example 6.7.1

To illustrate the use of array and function, we set for ourselves the following tasks:

1. Read in the amount to be deposited in the bank, the interest rate and the number of years to deposit.
2. Invoke the function to compute a table which keeps the amount we get after *i* years of deposit at the *i*-th component of the array.
3. Display out the above array

```
/* Compute compound interest   */
#include<iostream.h>
#include<iomanip.h>
#define YMAX 50
void interest(double, double, int, double []);
int main()
{
   double deposit, rate;
    int i, years;
   double compounded[YMAX];
   cout<< "\n ENTER DEPOSIT, INTEREST RATE, NUMBER OF YEARS \n";
   cin>>deposit>>rate>>years;
   cout<<endl;
   if(years>YMAX)
     cout<<"\n Number of years must be less than
                  or equal"<<YMAX;
   else
   {
      interest(deposit, rate, years, compounded);
      for( i = 0; i < years; ++i)
         cout<< i+1 << setw(25)<< compounded[i]
           << endl;
   }
  cout<< endl;
  return 0;
}
void interest(double deposit, double rate,
                              int years, double cp[])
{
   int i;
```

```
  for( i = 0; i < years; ++i){
    deposit = deposit*(1.0 + rate);
    cp[i] = deposit;
  }
}
```

Example 6.7.2

In the following program, we have to search an integer array for a given element. We use linear search in which each item in the array is examined sequentially until the desired item is found or the end of the array is reached.

```
#include<iostream.h>
int linearSearch( int [], int, int);
int main()
{
   const int arraySize = 100;
   int a[arraySize], searchkey, element;
  for (int x = 0; x < arraySize, x++)
                                    // create some data
    a[x] = 2*x;
  cout<< "Enter integer search key: "<< endl;
  cin >> searchKey;
  element = linearSearch(a, searchKey, arraySize);
  if(element !=-1)
    cout<<"Found value in element "<< element
          << endl;
  else
    cout<< "Value not found " << endl;
  return 0;
}
int linearSearch(int array[], int key,  int sizeofArray)
{
   for(int n = 0; n< sizeofArray; n++)
   if (array[n] = = key)
      return n;
   return –1;
}
```

6.8 POINTERS

In this section, we discuss one of the most powerful features of the C++ programming language, the *pointer*. Pointers are among C++'s most different capabilities to master. In section 6.5, we saw that references can be used to perform call-by-reference. Pointers enable programs to simulate call-by-reference and to create and manipulate dynamic data structures (i.e., data structures that can grow and shrink).

A *pointer* is a special type of variable that stores the memory address of other variables.

You declare a variable as a pointer by placing the *indirection operator* (*) after the data type or before the variable name.

Examples:

```
int *pFirstPtr;
int *pSecondPtr;
```

You use the *address-of operator* (&) to assign to the pointer variable the memory address of another variable.

Example:

```
double dPrimeInterest;
double *pPrimeInterest;
pPrimeInterest = &dPrimeInterest;
```

Once you assign the memory address of a variable to a pointer, to access or modify the contents of the variable pointed to by the pointer, you precede a pointer name in an expression with the *de-reference* (*) *operator*.

Example 6.8.1

The program in this example demonstrates the pointer operators. Memory locations are output in this example as hexadecimal integers.

```
#include<iostream.h>
int main()
{
   int a;
   int *aPtr;    // aPtr is a pointer to an integer

   a = 7;
   aPtr = &a;    //aPtr set to address of a
   cout << "The address of a is " << &a
        << "\nThe value of aPtr is " << aPtr;
   cout << "\n\nThe value of a is "<< a
        << "\nThe value of *aPtr is " << *aPtr
        << endl;
   return 0;
}
```

The output of the above program:

```
The address of a is  0x0065FDF4
The value of aPtr is 0x0065FDF4

The value of a is 7
The value of *aPtr is 7
```

Notice that the address of *a* and the value of *aPtr* are identical in the output, confirming that the address of *a* is assigned to the pointer variable *aPtr*.

6.8.1 Calling Functions by Reference with Pointer Arguments

In C++, programmers can use pointers and the dereference operator to simulate call-by-reference. When calling a function with arguments should be modified, the addresses of the arguments are passed. This is normally achieved by applying the *address-of* operator (&) to the name of the variable whose value will be used. A function receiving an address as an argument must define a pointer parameter to receive the address.

Example 6.8.2

```
// Cube a variable using call-by-reference
// with a pointer argument
#include <iostream.h>
void cubeByReference( int * );   // prototype
int main()
{
  int number = 5;
  cout << "The original value of number is " << number;
  cubeByReference( &number );
  cout << "\nThe new value of number is " << number << endl;
  return 0;
}

void cubeByReference( int *nPtr )
{
  *nPtr = (*nPtr) * (*nPtr) * (*nPtr);  // cube number in main
}
```

The output of the above propgram:

```
The original value of number is 5
The new value of number is 125
```

6.8.2 Pointers and Arrays

Notice that the name of an array by itself is equivalent to the *base address* of that array. That is, *the name z in isolation is equivalent to the expression &z[0]*.

Example 6.8.3

```
#include<iostream.h>
int main()
{
    int z[] = { 1, 2, 3, 4, 5};
    cout << "The value return by 'z' itself is
                    the addr " << z << endl;
    cout << "The address of the 0th element of
                        z is " << &z[0] << endl;
```

```
    return 0;
}
```

The output of the above program:

The value return by 'z' itself is the addr 0x0065FDF4
The address of the 0th element of z is 0x0065FDF4

Accessing Array Element Using Pointer and Offset

Now, let us store the address of array element 0 in a pointer. Then using the *indirection operator*, *, we can use the address in the pointer to access each array element.

For example, if we store the address of *grade[0]* into a pointer named *gPtr*, then the expression **gPtr* refers to *grade[0]*.

One unique feature of pointers is that *offset* may be included in pointer expression.

For example, the expression **(gPtr + 3)* refers to the variable that is *three* (elements) beyond the variable pointed to by *gPtr*.

The number 3 in the pointer expression is an *offset*. So *gPtr + 3* points to the element *grade[3]* of the *grade* array.

Example 6.8.4

```
#include <iostream.h>

int main()
{
  int b[] = { 10, 20, 30, 40 }, i, offset;
  int *bPtr = b;   // set bPtr to point to array b

  cout << "Array b printed with:\n"
       << "Array subscript notation\n";

  for ( i = 0; i < 4; i++ )
    cout << "b[" << i << "] = " << b[ i ] << '\n';

  cout << "\nPointer/offset notation\n";
  for ( offset = 0; offset < 4; offset++ )
    cout << "*(bPtr + " << offset << ") = "
         << *( bPtr + offset ) << '\n';
  return 0;
}
```

The output of the above program is:

Array b printed with:

Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40

6.8.3 Pointers and Strings

In C++ we often use character arrays to represent strings. A string is an array of characters ending in a null character (‘\0’). Therefore, according to section 6.8.2 we can scan through a string by using a pointer. Thus, in C++, it is appropriate to say that a string is a *constant pointer* – a pointer to the string’s first character.

A string may be assigned in a declaration to either a character array or a variable of type *char* *. The declarations

```
char color[] = “blue”;
char* colorPtr = “blue”;
```

each initialize a variable to the string “*blue*”. The first declaration creates a 5-element array *color* containing the characters ‘*b*’, ‘*l*’, ‘*u*’, ‘*e*’ and *‘\0’*. The second declaration creates pointer variable *colorPtr* that points to the string “*blue*” somewhere in the memory.

The first declaration determines the size of the array automatically based on the number of initializers provided in the initializer list.

Example 6.8.5

```
/* Printing a string one character at a time using a non-constant pointer to constant data
*/

#include<iostream.h>
int main( )
{
   char strng[] = “Adams”;
   char *sPtr;

   sPtr = &strng[0];
   cout << “\nThe string is: \n”;
   for( ; *sPtr != ‘\0’; sPtr++)
        cout << *sPtr << ‘ ‘;
   return 0;
}
```

The output of the above program:

```
The string is:
A d a m s
```

Note: The name of a string by itself is equivalent to the *base address* of that string.

6.8.4 Passing Structures as Parameters

Complete copies of all members of a structure can be passed to a function by including the name of the structure as an argument to the called function.

Example 6.8.6

```
#include <iostream.h>
struct Employee       // declare a global type
{
  int idNum;
  double payRate;
  double hours;
};
double calcNet(Employee);   // function prototype
int main()
{
  Employee emp = {6782, 8.93, 40.5};
  double netPay;
  netPay = calcNet(emp);       // pass by value
  cout << "The net pay for employee "
      << emp.idNum << " is $" << netPay << endl;
  return 0;
}

double calcNet(Employee temp) // temp is of data
                                    // type Employee
{
  return (temp.payRate * temp.hours);
}
```

The output is:

```
The net pay for employee 6782 is $361.665
```

In the above program, the function call

```
calcNet(emp);
```

passes a copy of the complete *emp* structure to the function *calcNet( ).* The parameter passing mechanism here is *call-by-value*.

An alternative to the pass-by-value function call, we can pass a structure by *passing a pointer*. The following example shows how to pass a structure by *passing a pointer*.

Example 6.8.7

```
#include <iostream.h>
struct Employee   // declare a global type
{
  int idNum;
  double payRate;
  double hours;
};
double calcNet(Employee *); //function prototype
int main()
{
  Employee emp = {6782, 8.93, 40.5};
  double netPay;
  netPay = calcNet(&emp);      // pass an address
  cout << "The net pay for employee "
     << emp.idNum << " is $" << netPay << endl;
  return 0;
}

double calcNet(Employee* pt)   //pt is a pointer
{                          //to a structure of Employee type
  return (pt->payRate * pt->hours);
}
```

The output is:

The net pay for employee 6782 is $361.665

6.9 THE typedef DECLARATION STATEMENT

The *typedef* declaration statement permits us to construct alternate names for an existing C++ data type name. The syntax of a *typedef* statement is:

typedef data-type new-type-name

For example, the statement:

typedef float REAL;

make the name REAL a synonym for float. The name REAL can now be used in place of the term *float* anywhere in the program after the synonym has been declared.

The definition
REAL val;
is equivalent to

```
    float val;
```

Example: Consider the following statement:

```
    typedef struct
    {
      char name[20];
      int idNum;
    }  EMPREC;
```

The declaration

```
    EMPREC employee[75];
```

is equivalent to

```
    struct
    {
      char name[20];
      int idNum;
    } employee[75];
```

Example: Consider the following statement:

```
    typedef  double*  DPTR;
```

The declaration:

```
    DPTR pointer1;
```

is equivalent to

```
    double* pointer1;
```