Chapter Three

3. Arrays and Strings

3.1. Introduction

In a program have values associated with them. During program execution these values are accessed by using the identifier associated with the variable in expressions etc. In none of the programs written so far have very many variables been used to represent the values that were required. Thus even though programs have been written that could handle large lists of numbers it has not been necessary to use a separate identifier for each number in the list. This is because in all these programs it has never been necessary to keep a note of each number individually for later processing. For example in summing the numbers in a list only one variable was used to hold the current entered number which was added to the accumulated sum and was then overwritten by the next number entered. If that value were required again later in the program there would be no way of accessing it because the value has now been overwritten by the later input. If only a few values were involved a different identifier could be declared for each variable, but now a loop could not be used to enter the values. Using a loop and assuming that after a value has been entered and used no further use will be made of it allows the following code to be written. This code enters five numbers and outputs their sum:

```
#include<iostream.h>

    int main()
    {
    int x;
    int sum=0;
    for (int i = 1; i<=5; i++)
      {
      cout<<"enter the number:"<<endl;
        cin >> x;
        sum += x;
      }
      cout<<"The sum is="<<sum;
       return 0; }
```

3.2. What is an arrays

An **array** is a data structure which allows a collective name to be given to a group of elements which ***all have the same type***. An individual element of an array is identified by its own unique ***index*** (or **subscript**).

An array can be thought of as a collection of numbered boxes each containing one data item. The number associated with the box is the index of the item. To access a particular item the index of the box associated with the item is used to access the appropriate box. The index **must** be an integer and indicates the position of the element in the array. Thus the elements of an array are **ordered** by the index.

3.3. One Dimensional Array

3.3.1. Declaration of Arrays

An array declaration is very similar to a variable declaration. First a type is given for the elements of the array, then an identifier for the array and, within square brackets, the number of elements in the array. The number of elements **must be an integer**.

For example data on the average temperature over the year in Ethiopia for each of the last 100 years could be stored in an array declared as follows:

```
float annual_temp[100];
```

This declaration will cause the compiler to allocate space for 100 consecutive float variables in memory. The number of elements in an array must be fixed at compile time. It is best to make the array size a constant and then, if required, the program can be changed to handle a different size of array by changing the value of the constant,

```
const int NE = 100;
float annual_temp[NE];
```

then if more records come to light it is easy to amend the program to cope with more values by changing the value of NE. This works because the compiler knows the value of the constant NE at compile time and can allocate an appropriate amount of space for the array. It would not work if an ordinary variable was used for the size in the array declaration since at compile time the compiler would not know a value for it.

3.3.2. Accessing Array Elements

Given the declaration above of a 100-element array the compiler reserves space for 100 consecutive floating point values and accesses these values using an index/subscript that takes values from 0 to 99. The ***first element*** in an array in C++ always has the ***index 0***, and if the array has ***`n`*** elements the ***last*** element will have the ***index `n-1`***.

An **array element** is accessed by writing the identifier of the array followed by the subscript in square brackets. Thus to set the 15th element of the array above to 1.5 the following assignment is used:

```
annual_temp[14] = 1.5;
```

Note that since the first element is at index 0, then the ***`i`th*** element is at ***index `i-1`***. Hence in the above the 15th element has index 14.

An array element can be used anywhere an identifier may be used. Here are some examples assuming the following declarations:

```
const int NE = 100,
          N = 50;
int i, j, count[N];
float annual_temp[NE];
float sum, av1, av2;
```

A value can be read into an array element directly, using cin

```
cin >> count[i];
```

The element can be increased by 5,

```
count[i] = count[i] + 5;
```

or, using the shorthand form of the assignment

```
count[i] += 5;
```

Array elements can form part of the condition for an if statement, or indeed, for any other logical expression:

```
if (annual_temp[j] < 10.0)
   cout << "It was cold this year "
        << endl;
```

for statements are the usual means of accessing every element in an array. Here, the first NE elements of the array annual_temp are given values from the input stream cin.

```
for (i = 0; i < NE; i++)
  cin >> annual_temp[i];
```

The following code finds the average temperature recorded in the first ten elements of the array.

```
sum = 0.0;
for (i = 0; i <10; i++)
    sum += annual_temp[i];
av1 = sum / 10;
```

Notice that it is good practice to use named constants, rather than literal numbers such as 10. If the program is changed to take the average of the first 20 entries, then it all too easy to forget to change a 10 to 20. If a const is used consistently, then changing its value will be all that is necessary.

For example, the following example finds the average of the last k entries in the array. k could either be a variable, or a declared constant. Observe that a change in the value of k will still calculate the correct average (provided **`k<=NE`**).

```
sum = 0.0;
for (i = NE - k; i < NE; i++)
  sum += annual_temp[i];
av2 = sum / k;
```

Important - C++ does not check that the subscript that is used to reference an array element actually lies in the subscript range of the array. Thus C++ will allow the assignment of a value to `annual_temp[200]`, however the effect of this assignment is unpredictable. For example it could lead to the program attempting to assign a value to a memory element that is outside the program's allocated memory space. This would lead to the program being terminated by the operating system. Alternatively it might actually access a memory location that is within the allocated memory space of the program and assign a value to that location, changing the value of the variable in your program which is actually associated with that memory location, or overwriting the machine code of your program. Similarly reading a value from `annual_temp[200]` might access a value that has not been set by the program or might be the value of another variable. It is the programmer's responsibility to ensure that if an array is declared with `n` elements then no attempt is made to reference any element with a subscript outside the range 0 to `n-1`. Using an index, or subscript, that is out of range is called Subscript Overflow. Subscript overflow is one of the commonest causes of erroneous results and can frequently cause very strange and hard to spot errors in programs.

3.3.3. Initialization of arrays

The initialization of simple variables in their declaration has already been covered. An array can be initialized in a similar manner. In this case the initial values are given as a list enclosed in curly brackets. For example initializing an array to hold the first few prime numbers could be written as follows:

```
int primes[] = {1, 2, 3, 5, 7, 11, 13};
```

Note that the array has not been given a size, the compiler will make it large enough to hold the number of elements in the list. In this case primes would be allocated space for seven elements. If the array is given a size then this size must be greater than or equal to the number of elements in the initialization list. For example:

```
int primes[10] = {1, 2, 3, 5, 7};
```

would reserve space for a ten element array but would only initialize the first five elements.

Example Program: Printing Outliers in Data

The requirement specification for a program is:

A set of positive data values (200) are available. It is required to find the average value of these values and to count the number of values that are more than 10% above the average value.

Since the data values are all positive a negative value can be used as a sentinel to signal the end of data entry. Obviously this is a problem in which an array must be used since the values must first be entered to find the average and then each value must be compared with this average. Hence the use of an array to store the entered values for later re-use.

An initial algorithmic description is:

```
initialize.
enter elements into array and sum elements.
evaluate average.
scan array and count number greater than
                       10% above average.
output results.
```

This can be expanded to the complete algorithmic description:

```
set sum to zero.
set count to zero.
set nogt10 to zero.
enter first value.
while value is positive
```

```
        {
         put value in array element with index count.
         add value to sum.
         increment count.
         enter a value.
        }
    average = sum/count.
    for index taking values 0 to count-1
        if array[index] greater than 1.1*average
           then increment nogt10.
    output average, count and nogt10.
```

In the above the variable nogt10 is the number greater than 10% above the average value. It is easy to argue that after exiting the while loop, count is set to the number of positive numbers entered. Before entering the loop count is set to zero and the first number is entered, that is count is one less than the number of numbers entered. Each time round the loop another number is entered and count is incremented hence count remains one less than the number of numbers entered. But the number of numbers entered is one greater than the number of positive numbers so count is therefore equal to the number of positive numbers.

A **main()** program written from the above algorithmic description is given below:

```
void main()
{
  const int NE = 200;  // maximum no of elements in array
  float sum = 0.0;     // accumulates sum
  int count = 0;       // number of elements entered
  int nogt10 = 0;      // counts no greater than 10%
                       // above average
  float x;             // holds each no as input
  float indata[NE];    // array to hold input
  float average;       // average value of input values
  int i;               // control variable

      // Data entry, accumulate sum and count
      // number of +ve numbers entered
  cout << "Enter numbers, -ve no to terminate: " << endl;
  cin >> x;
  while (x >= 0.0)
    {
      sum = sum + x;
      indata[count] = x;
      count = count + 1;
      cin >> x;
    }

      // calculate average
  average = sum/count;

      // Now compare input elements with average
  for (i = 0; i < count; i++)
```

```
        {
          if (indata[i] > 1.1 * average)
            nogt10++;
        }

          // Output results
      cout << "Number of values input is " << n;
      cout << endl
           << "Number more than 10% above average is "
           << nogt10 << endl;
    }
```

Since it was assumed in the specification that there would be less than 200 values the array size is set at 200. In running the program less than 200 elements may be entered, if `n` elements where `n < 200` elements are entered then they will occupy the first `n` places in the array `indata`. It is common to set an array size to a value that is the maximum we think will occur in practice, though often not all this space will be used.

Example Program: Test of Random Numbers

The following program simulates the throwing of a dice by using a random number generator to generate integers in the range 0 to 5. The user is asked to enter the number of trials and the program outputs how many times each possible number occurred.

An array has been used to hold the six counts. This allows the program to increment the correct count using one statement inside the loop rather than using a switch statement with six cases to choose between variables if separate variables had been used for each count. Also it is easy to change the number of sides on the dice by changing a constant. Because C++ arrays start at subscript 0 the count for an i occurring on a throw is held in the i-1th element of this count array. By changing the value of the constant die_sides the program could be used to simulate a die_sides-sided die without any further change.

```
    #include <iostream.h>
    #include <stdlib.h>    // time.h and stdlib.h required for
    #include <time.h>      // random number generation

    void main()
    {
      const int die_sides = 6;    // maxr-sided die
      int count[die_sides];       // holds count of each
                             // possible value
      int no_trials,         // number of trials
          roll,              // random integer
          i;                 // control variable
      float sample;          // random fraction 0 .. 1
```

```
        // initialize random number generation and count
        // array and input no of trials
    srand(time(0));
    for (i=0; i < die_sides; i++)
      count[i] = 0;
    cout << "How many trials? ";
    cin >> no_trials;

        // carry out trials
    for (i = 0; i < no_trials; i++)
      {
        sample = rand()/float(RAND_MAX);
        roll = int ( die_sides * sample);
             // returns a random integer in 0 to die_sides-1
        count[roll]++;          // increment count
      }

        // Now output results
    for (i = 0; i < die_sides; i++)
      {
        cout << endl << "Number of occurrences of "
             << (i+1) << " was " << count[i];
      }
    cout << endl;
  return 0;
  }
```

3.3.4. Copying Arrays

The assignment operator *cannot* be applied to array variables:

```
const int SIZE=10
int x [SIZE] ;
int y [SIZE] ;
x = y ;             // Error - Illegal
```

Only individual elements can be assigned to using the index operator, e.g., **`x[1] = y[2];`**.

To make all elements in 'x' the same as those in 'y' (equivalent to assignment), a loop has to be used.

```
// Loop to do copying, one element at a time
for (int i = 0 ; i < SIZE; i++)
      x[i] = y[i];
```

This code will copy the elements of array y into x, overwriting the original contents of x. A loop like this has to be written whenever an array assignment is needed.

Notice the use of a constant to store the array size. This avoids the literal constant '10' appearing a number times in the code. If the code needs to be edited to use different sized arrays, only the constant needs to be changed. If the constant is not used, all the '10's would have to be changed individually - it is easy to miss one out.

3.4.Multidimensional arrays

An array may have more than one dimension. Each dimension is represented as a subscript in the array. Therefore a two dimensional array has two subscripts, a three dimensional array has three subscripts, and so on.

Arrays can have any number of dimensions, although most of the arrays that you create will likely be of one or two dimensions.

A chess board is a good example of a two-dimensional array. One dimension represents the eight rows, the other dimension represents the eight columns.

Suppose the program contains a class named square. The declaration of array named board that represents would be

```
Square board[8][8];
```

The program could also represent the same data with a one dimensional, 64-square array. For example, it could include the statement

```
Square board[64];
```

Such a representation does not correspond as closely to the real-world object as the two dimensional array, however.

Suppose that when the game begins. The king id located in the fourth position in the first row. Counting from zero that position corresponds to board[0][3] in the two dimensional array, assuming that the first subscript corresponds to the row, and the second to the column.

3.4.1. Initializing Multidimensional Arrays

To initialize a multidimensional arrays , you must assign the list of values to array elements in order, with last array subscript changing while the first subscript while the first subscript holds steady. Therefore, if the program has an array **`int theArray[5][3],`** the first three elements go **`int theArray[0];`** the next three into **`theArray[1];`** and so forth.

The program initializes this array by writing

```
int theArray[5][3] ={ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
                      12, 13, 14, 15};
```

for the sake of clarity, the program could group the initializations with braces, as shown below.

```
int theArray[5][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9},
                       {10, 11, 12}, {13, 14,15} };
```

The compiler ignores the inner braces, which clarify how the numbers are distributed.

Each value should be separated by comma, regardless of whither inner braces are include.

The entire initialization must set must appear within braces, and it must end with a semicolon.

3.4.2. Omitting the Array Size

If a one-dimensional array is initialized, the size can be omitted as it can be found from the number of initializing elements:

```
int x[] = { 1, 2, 3, 4} ;
```

This initialization creates an array of four elements.

Note however:

```
int x[][] = { {1,2}, {3,4} } ; // error is not allowed.
and must be written
int x[2][2] = { {1,2}, {3,4} } ;
```

Example of multidimensional array

```
    #include<iostream.h>
    int main(){
     int SomeArray[5][2] =  {{0,0},{1,2}, {2,4},{3,6},
                           {4,8}}
     for ( int i=0; i<5; i++)
      for (int j = 0; j<2;j++)
      {
          cout<<endl<<SomeArray[i][ j];
      }
Return 0;
  }
```

3.5.Pointers

A pointer is simply the *address* of a memory location and provides an indirect way of accessing data in memory. A pointer variable is defined to ‘point to’ data of a specific type. For example:

```
int  *ptr1;      // pointer to an int
char *ptr2;      // pointer to a char
```

The *value* of a pointer variable is the address to which it points. For example, given the definitions

```
int     num;
```

we can write:

```
ptr1 = &num;
```

The symbol `&` is the **address** operator; it takes a variable as argument and returns the memory address of that variable. The effect of the above assignment is that the address of `num` is assigned to `ptr1`. Therefore, we say that `ptr1` points to `num`. Figure 5.**Error! Bookmark not defined.** illustrates this diagrammatically.

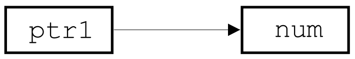


Figure: A simple integer pointer.

Given that ptr1 points to num, the expression

```
*ptr1
```

dereferences `ptr1` to get to what it points to, and is therefore equivalent to `num`. The symbol `*` is the **dereference** operator; it takes a pointer as argument and returns the contents of the location to which it points.

In general, the type of a pointer must match the type of the data it is set to point to. A pointer of type void*, however, will match any type. This is useful for defining pointers which may point to data of different types, or whose type is originally unknown. A pointer may be **cast** (type converted) to another type. For example,

```
ptr2 = (char*) ptr1;
```

converts `ptr1` to `char` pointer before assigning it to `ptr2`.

Regardless of its type, a pointer may be assigned the value 0 (called the **null** pointer). The null pointer is used for initializing pointers, and for marking the end of pointer-based data structures (e.g., linked lists). ☐

3.5.1. Dynamic Memory

In addition to the program stack (which is used for storing global variables and stack frames for function calls), another memory area, called the **heap**, is provided. The heap is used for dynamically allocating memory blocks during program execution. As a result, it is also called **dynamic memory**. Similarly, the program stack is also called **static memory**.

Two operators are used for allocating and deallocating memory blocks on the heap. The new operator takes a type as argument and allocated a memory block for an object of that type. It returns a pointer to the allocated block. For example,

```
int  *ptr = new int;
char *str = new char[10];
```

allocate, respectively, a block for storing a single integer and a block large enough for storing an array of 10 characters.

Memory allocated from the heap does not obey the same scope rules as normal variables. For example, in

```
void Foo (void)
{
    char *str = new char[10];
    //...
}
```

when `Foo` returns, the local variable `str` is destroyed, but the memory block pointed to by `str` is *not*. The latter remains allocated until explicitly released by the programmer.

The `delete` operator is used for releasing memory blocks allocated by `new`. It takes a pointer as argument and releases the memory block to which it points. For example:

```
delete ptr;         // delete an object
delete [] str;      // delete an array of objects
```

Note that when the block to be deleted is an array, an additional `[]` should be included to indicate this.

Should `delete` be applied to a pointer which points to anything but a dynamically-allocated object (e.g., a variable on the stack), a serious runtime error may occur. It is harmless to apply `delete` to the 0 pointer.

Dynamic objects are useful for creating data which last beyond the function call which creates them. Listing 5.1 illustrates this using a function which takes a string parameter and returns a *copy* of the string.

Listing 5.1

```
1  #include <string.h>

2  char* CopyOf (const char *str)
3  {
4      char *copy = new char[strlen(str) + 1];

5      strcpy(copy, str);
6      return copy;
7  }
```

Annotation (analysis)

1 This is the standard string header file which declares a variety of functions for manipulating strings.

4 The `strlen` function (declared in `string.h`) counts the characters in its string argument up to (but excluding) the final null character. Because the null character is not included in the count, we add 1 to the total and allocate an array of characters of that size.

5 The `strcpy` function (declared in `string.h`) copies its second argument to its first, character by character, including the final null character.

Because of the limited memory resources, there is always the possibility that dynamic memory may be exhausted during program execution, especially when many large blocks are allocated and none released. Should `new` be unable to allocate a block of the requested size, it will return 0 instead. It is the responsibility of the programmer to deal with such possibilities.

□

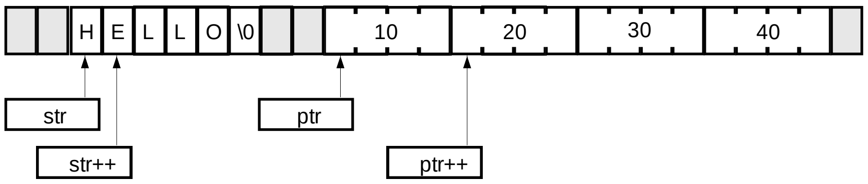
3.5.2. Pointer Arithmetic

In C++ one can add an integer quantity to or subtract an integer quantity from a pointer. This is frequently used by programmers and is called pointer arithmetic. Pointer arithmetic is *not* the same as integer arithmetic, because the outcome depends on the size of the object pointed to. For example, suppose that an `int` is represented by 4 bytes. Now, given

```
char *str = "HELLO";
int  nums[] = {10, 20, 30, 40};
int  *ptr = &nums[0];           // pointer to first element
```

str++ advances str by one char (i.e., one byte) so that it points to the second character of "HELLO", whereas ptr++ advances ptr by one int (i.e., four bytes) so that it points to the second element of nums. Figure 5.1 illustrates this diagrammatically.

Figure 5.1 Pointer arithmetic.



It follows, therefore, that the elements of "HELLO" can be referred to as *str, *(str + 1), *(str + 2), etc. Similarly, the elements of nums can be referred to as *ptr, *(ptr + 1), *(ptr + 2), and *(ptr + 3).

Another form of pointer arithmetic allowed in C++ involves subtracting two pointers of the same type. For example:

```
int *ptr1 = &nums[1];
int *ptr2 = &nums[3];
int n = ptr2 - ptr1;     // n becomes 2
```

Pointer arithmetic is very handy when processing the elements of an array. Listing 5.2 shows as an example a string copying function similar to strcpy.

Listing 5.2

```
1  void CopyString (char *dest, char *src)
2  {
3      while (*dest++ = *src++)
4          ;
5  }
```

Annotation

3 The condition of this loop assigns the contents of src to the contents of dest and then increments both pointers. This condition becomes 0 when the final null character of src is copied to dest.

In turns out that an array variable (such as nums) is itself the address of the first element of the array it represents. Hence the elements of nums can also be referred to using pointer arithmetic on nums, that is, nums[i] is equivalent to *(nums + i). The difference between nums and ptr is that nums is a constant, so it cannot be made to point to anything else, whereas ptr is a variable and can be made to point to any other integer.

Listing 5.3 shows how the HighestTemp function (shown earlier in Listing 5.**Error! Bookmark not defined.**) can be improved using pointer arithmetic.

Listing 5.3

```
1  int HighestTemp (const int *temp, const int rows, const int columns)
2  {
3      int highest = 0;

4      for (register i = 0; i < rows; ++i)
5      for (register j = 0; j < columns; ++j)
6              if (*(temp + i * columns + j) > highest)
7                  highest = *(temp + i * columns + j);
8      return highest;
9  }
```

Annotation

1 Instead of passing an array to the function, we pass an `int` pointer and two additional parameters which specify the dimensions of the array. In this way, the function is not restricted to a specific array size.

6 The expression `*(temp + i * columns + j)` is equivalent to `temp[i][j]` in the previous version of this function.

`HighestTemp` can be simplified even further by treating `temp` as a one-dimensional array of `row * column` integers. This is shown in Listing 5.4.

Listing 5.4

```
int HighestTemp (const int *temp, const int rows, const int columns)
{
    int highest = 0;

    for (register i = 0; i < rows * columns; ++i)
        if (*(temp + i) > highest)
            highest = *(temp + i);
    return highest;
}
```

□

3.5.3. Function Pointers

It is possible to take the address of a function and store it in a function pointer. The pointer can then be used to indirectly call the function. For example,

```
int (*Compare)(const char*, const char*);
```

defines a function pointer named `Compare` which can hold the address of any function that takes two constant character pointers as arguments and returns an integer. The string comparison library function `strcmp`, for example, is such. Therefore:

```
Compare = &strcmp;          // Compare points to strcmp function
```

The `&` operator is not necessary and can be omitted:

```
Compare = strcmp;           // Compare points to strcmp function
```

Alternatively, the pointer can be defined and initialized at once:

```
int (*Compare)(const char*, const char*) = strcmp;
```

When a function address is assigned to a function pointer, the two types must match. The above definition is valid because `strcmp` has a matching function prototype:

```
int strcmp(const char*, const char*);
```

Given the above definition of `Compare`, `strcmp` can be either called directly, or indirectly via `Compare`. The following three calls are equivalent:

```
strcmp("Tom", "Tim");            // direct call
(*Compare)("Tom", "Tim");        // indirect call
Compare("Tom", "Tim");           // indirect call (abbreviated)
```

A common use of a function pointer is to pass it as an argument to another function; typically because the latter requires different versions of the former in different circumstances. A good example is a binary search function for searching through a sorted array of strings. This function may use a comparison function (such as `strcmp`) for comparing the search string against the array strings. This might not be appropriate for all cases. For example, `strcmp` is case-sensitive. If we wanted to do the search in a non-case-sensitive manner then a different comparison function would be needed.

As shown in Listing 5.5, by making the comparison function a parameter of the search function, we can make the latter independent of the former.

Listing 5.5

```
 1  int BinSearch (char *item, char *table[], int n,
 2                 int (*Compare)(const char*, const char*))
 3  {
 4      int bot = 0;
 5      int top = n - 1;
 6      int mid, cmp;

 7      while (bot <= top) {
 8          mid = (bot + top) / 2;
 9          if ((cmp = Compare(item,table[mid])) == 0)
10              return mid;             // return item index
11          else if (cmp < 0)
12              top = mid - 1;          // restrict search to lower half
13          else
14              bot = mid + 1;          // restrict search to upper half
15      }
16      return -1;                      // not found
17  }
```

Annotation

1 Binary search is a well-known algorithm for searching through a *sorted* list of items. The search list is denoted by `table` which is an array of strings of dimension `n`. The search item is denoted by `item`.

2 `Compare` is the function pointer to be used for comparing `item` against the array elements.

7 Each time round this loop, the search span is reduced by half. This is repeated until the two ends of the search span (denoted by `bot` and `top`) collide, or until a match is found.

9 The item is compared against the middle item of the array.

10 If `item` matches the middle item, the latter's index is returned.

11 If `item` is less than the middle item, then the search is restricted to the lower half of the array.

14 If `item` is greater than the middle item, then the search is restricted to the upper half of the array.

16 Returns -1 to indicate that there was no matching item.

The following example shows how `BinSearch` may be called with `strcmp` passed as the comparison function:

```
char *cities[] = {"Boston", "London", "Sydney", "Tokyo"};
cout << BinSearch("Sydney", cities, 4, strcmp) << '\n';
```

This will output 2 as expected. □

3.5.4. References

A reference introduces an **alias** for an object. The notation for defining references is similar to that of pointers, except that `&` is used instead of `*`. For example,

```
double num1 = 3.14;
double &num2 = num1;     // num is a reference to num1
```

defines `num2` as a reference to `num1`. After this definition `num1` and `num2` both refer to the same object, as if they were the same variable. It should be emphasized that a reference does not create a copy of an object, but merely a symbolic alias for it. Hence, after

```
num1 = 0.16;
```

both `num1` and `num2` will denote the value 0.16.

A reference must always be initialized when it is defined: it should be an alias for something. It would be illegal to define a reference and initialize it later.

```
double &num3;        // illegal: reference without an initializer
num3 = num1;
```

You can also initialize a reference to a constant. In this case a *copy* of the constant is made (after any necessary type conversion) and the reference is set to refer to the copy.

```
int &n = 1;         // n refers to a copy of 1
```

The reason that `n` becomes a reference to a copy of 1 rather than 1 itself is safety. Consider what could happen if this were not the case.

```
int &x = 1;
++x;
int y = x + 1;
```

The 1 in the first and the 1 in the third line are likely to be the same object (most compilers do constant optimization and allocate both 1's in the same memory location). So although we expect `y` to be 3, it could turn out to be 4. However, by forcing `x` to be a copy of 1, the compiler guarantees that the object denoted by `x` will be different from both 1's.

The most common use of references is for function parameters. Reference parameters facilitates the **pass-by-reference** style of arguments, as opposed to the **pass-by-value** style which we have used so far. To observe the differences, consider the three swap functions in Listing 5.6.

Listing 5.6

```
 1  void Swap1 (int x, int y)        // pass-by-value (objects)
 2  {
 3      int temp = x;
 4      x = y;
 5      y = temp;
 6  }

 7  void Swap2 (int *x, int *y)      // pass-by-value (pointers)
 8  {
 9      int temp = *x;
10      *x = *y;
11      *y = temp;
12  }

13  void Swap3 (int &x, int &y)      // pass-by-reference
14  {
15      int temp = x;
16      x = y;
17      y = temp;
18  }
```

Annotation

1 Although `Swap1` swaps `x` and `y`, this has no effect on the arguments passed to the function, because `Swap1` receives a *copy* of the arguments. What happens to the copy does not affect the original.

7 `Swap2` overcomes the problem of `Swap1` by using pointer parameters instead. By dereferencing the pointers, `Swap2` gets to the original values and swaps them.

13 `Swap3` overcomes the problem of `Swap1` by using reference parameters instead. The parameters become aliases for the arguments passed to the function and therefore swap them as intended.

Swap3 has the added advantage that its call syntax is the same as Swap1 and involves no addressing or dereferencing. The following main function illustrates the differences:

```
int main (void)
{
    int i = 10, j = 20;
    Swap1(i, j);    cout << i << ", " << j << '\n';
    Swap2(&i, &j);  cout << i << ", " << j << '\n';
    Swap3(i, j);    cout << i << ", " << j << '\n';
}
```

When run, it will produce the following output:

```
10, 20
20, 10
10, 20
```
□

3.5.5. Typedefs

Typedef is a syntactic facility for introducing symbolic names for data types. Just as a reference defines an alias for an object, a typedef defines an alias for a type. Its main use is to simplify otherwise complicated type declarations as an aid to improved readability. Here are a few examples:

```
typedef char *String;
Typedef char Name[12];
typedef unsigned int uint;
```

The effect of these definitions is that String becomes an alias for char*, Name becomes an alias for an array of 12 chars, and uint becomes an alias for unsigned int. Therefore:

```
String  str;    // is the same as: char *str;
Namename;       // is the same as: char name[12];
uintn;          // is the same as: unsigned int n;
```

The complicated declaration of Compare in Listing 5.5 is a good candidate for typedef:

```
typedef int (*Compare)(const char*, const char*);

int BinSearch (char *item, char *table[], int n, Compare comp)
{
    //...
        if ((cmp = comp(item, table[mid])) == 0)
            return mid;
    //...
}
```

The typedef introduces Compare as a new type name for any function with the given prototype. This makes BinSearch's signature arguably simpler.