```
a[0] = 1
a[1] = 5
a[2] = 6
a[3] = 8
a[4] = 9
```

Initialization at Definition

Arrays can be initialized at the point of their definition as follows:

data-type array-name[size] = { list of values separated by comma };

For instance, the statement

```
int age[ 5 ] = { 19, 21, 16, 1, 50 };
```

defines an array of integers of size 5. In this case, the first element of the array age is initialized with 19, second with 21, and so on as shown in Figure 6.5. A semicolon always follows the closing brace. The array size may be omitted when the array is initialized during array definition as follows:

```
int age[] = { 19, 21, 16, 1, 50 };
```

In such cases, the compiler assumes the array size to be equal to the number of elements enclosed within the curly braces. Hence, in the above statement, the size of the array is considered as five.

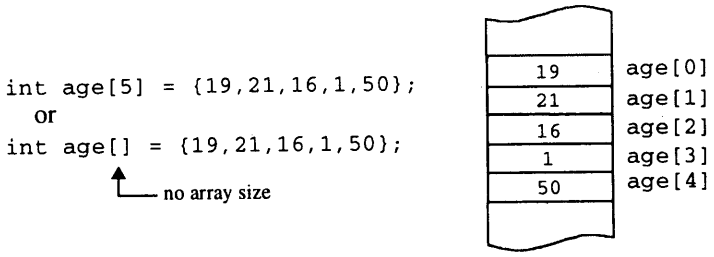

Figure 6.5: Array initialization at its definition

Caution! No Array Bound Validation

C++ does not support bound checking i.e., it does not check for the validity of the array index value while accessing the array elements. If the program tries to store something beyond the size of an array, neither the compiler nor the run-time will indicate the error. Such a situation may cause overwriting of data or code leading to fatal errors. Therefore, the programmer has to take extra care to use indexes within the array limits. For example, consider the following program:

```
void main()
{
   int age[ 40 ];
   age[ 50 ] = 11;
   age[ 50 ]++;
}
```

It defines age to be an array of 40 integers, and then modifies the 51st element! The compiler does not consider such an access as illegal and produces the executable code. Execution of such programs can behave in an unpredictable manner. Detecting such errors in a program is a difficult and time consuming task. Thus, it is the responsibility of the programmer to see that the value of an array index is within the array bounds while accessing an array element.

6.3 Array Illustrations

The program `elder.cpp` finds the age of the eldest and youngest person in a family. It reads the ages of all the members of a family stores them in an array and then scans the array to find out the required information.

```
// elder.cpp: finding youngest and eldest person age
#include <iostream.h>
void main()
{
   int i, n;
   float age[25], younger, elder;
   cout << "How many persons are there in list <max-25> ? ";
   cin >> n;
   for( i = 0; i < n; i++ )
   {
      cout << "Enter person" << i+1 << " age: ";
      cin >> age[i];
   }
   // finding youngest and eldest person age begins here
   younger = age[0];
   elder = age[0];
   for (i = 1; i < n; i++)
   {
      if( age[i] < younger )
         younger = age[i];
      else
         if( age[i] > elder )
            elder = age[i];
   }
   // finding younger and elder person ends here
   cout << "Age of eldest person is " << elder << endl;
   cout << "Age of youngest person is " << younger;
}
```

Run

```
How many persons are there in list <max-25> ? 7
Enter person1 age: 25
Enter person2 age: 4
Enter person3 age: 45
Enter person4 age: 18
Enter person5 age: 35
Enter person6 age: 23
Enter person7 age: 32
Age of eldest person is 45
Age of youngest person is 4
```

Bubble Sort

A classical bubble sort is the first standard sorting algorithm most programmers learn to code. It has gained popularity because it is intuitive, easy to write and debug, and consumes little memory. In each

pass, the first two items in a list are compared and placed in the correct order. Items two and three are then compared and reordered, followed by items three and four, then four and five, and so on. The sort continues until a pass with no swap occurs. High-value items near the beginning of a list (as shown in Figure 6.6) move to their correct position rapidly and are called turtles, because they move only one position with each pass. The program `bubble.cpp` illustrates the implementation of the bubble sort.

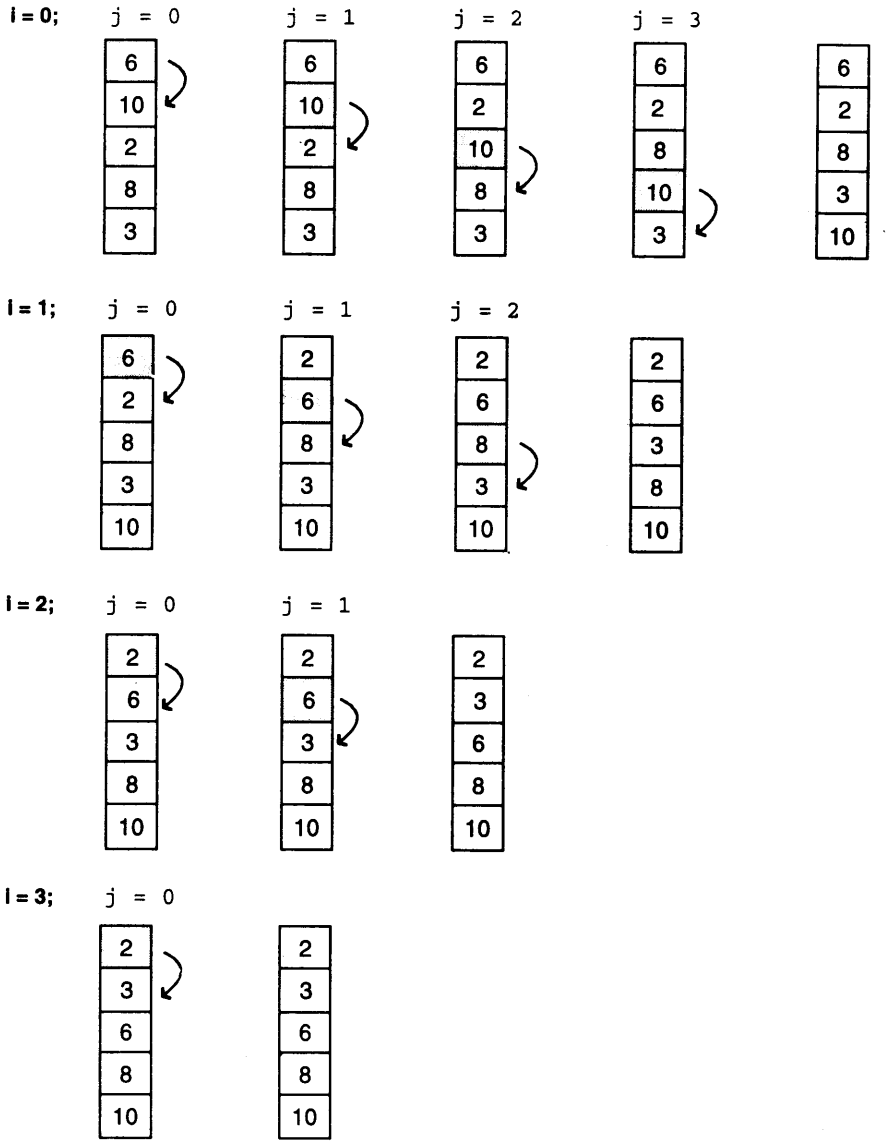

Figure 6.6: Trace of Bubble Sort

```
// bubble.cpp: sorting of numbers using bubble sorting
#include <iostream.h>
void main()
{
   int i, j, n, age[25], flag, temp;
   cout << "How many elements to sort <max-25> ? ";
   cin >> n;
   for( i = 0; i < n; i++ )
   {
      cout << "Enter age[ " << i << " ]: ";
      cin >> age[i];
   }
   // sorting starts here using bubble sort technique
   for( i = 0; i < n-1; i++ )    // for i = 0 to n-2
   {
      flag = 1;
      for( j = 0; j < (n-1-i); j++ )   // for j = 0 to (n-i-2)
      {
         if( age[j] > age[j+1] )
         {
            flag = 0;   // still not sorted and requires next iteration
            // exchange contents of age[j] and age[j+1]
            temp = age[j];
            age[j] = age[j+1];
            age[j+1] = temp;
         }
      }
      if( flag )
         break;  // data are now in order; no need of next iteration
   }
   // sorting ends here
   cout << "Sorted list..." << endl;
   for( i = 0; i < n; i ++ )
      cout << age[i] << " ";
}
```

Run

```
How many elements to sort <max-25> ? 7
Enter age[ 0 ]: 3
Enter age[ 1 ]: 5
Enter age[ 2 ]: 9
Enter age[ 3 ]: 4
Enter age[ 4 ]: 2
Enter age[ 5 ]: 1
Enter age[ 6 ]: 6
Sorted list...
1 2 3 4 5 6 9
```

Comb Sort

Comb sort is a generalization of the bubble sort that permits comparison of non-adjacent items. It retains the simplicity of a bubble sort, but with a dramatic increase in speed. Consider a sample list of 100

elements to be arranged in the ascending order. In this method elements are compared to sort them and the space between the elements to be compared is known as the *gap*. (For instance, the gap in bubble sort is one.) A gap of 80 would compare elements 1 and 81, 2 and 82, ..., and 20 and 100, and switch pairs when appropriate. Such a pass would take 20 comparisons rather than the 99 of an equivalent bubble sort. The benefit is that the swap could move the elements as much as 80 notches closer to their final destination. It is found that the ideal way to select the next gap is to divide the previous gap by 1.3 (which is known as the *shrinking factor*). The shrinking factor 1.3 has been experimentally found out to be the optimal value. The gap value remains constant once it reaches 1. A bubble sort is converted into comb sort by the following process:

- Initialize the gap with 1 in the inner loop.
- Initialize the gap size and the dimension of the list.
- Recalculate the gap with the do-loop by dividing the previous gap by 1.3, taking the integer part and using the result or 1, whichever is greater.
- Repeat the loop until the gap is 1 and the switch counter is 0, indicating that the sort operation is completed.

The program `comb.cpp` illustrates the implementation of the comb sort. The only difference between bubble sort and comb sort is that, in bubble sort, the turtles (data) crawl whereas in comb sort they jump. Successively shrinking the gap is analogous to combing long, tangled hair—stroking first with fingers alone, then with a pick comb that has widely spaced teeth, followed by finer combs with progressively closer teeth. Comb sort has a similar shrinking effect on the gap (hence, the name comb sort). Each stroke presorts the list (i.e., it kills or winds up some turtles). Therefore, by the time the gap declines to unity (a Bubble sort), all the elements are so close to their final position that applying a bubble sort at this stage is efficient.

```
// comb.cpp: sorting of numbers using comb sorting
#define SHRINKINGFACTOR 1.3
#include <iostream.h>
void main()
{
   int i, j, n, age[25], flag, temp;
   cout << "How many elements to sort <max-25> ? ";
   cin >> n;
   for( i = 0; i < n; i++ )
   {
      cout << "Enter age[ " << i << " ]: ";
      cin >> age[i];
   }
   // sorting starts here using comb sort technique
   int size = n;
   int gap = size;    // gap is initialized to size i.e, length of a list
   do
   {
     gap = (int) (float(gap)/SHRINKINGFACTOR);
      switch( gap )
      {
         case 0:
           gap = 1; // the smallest gap is 1 as in bubble sort
            break;
```

```
            case 9:
            case 10:
               gap = 11;
               break;
         }
         flag = 1;
         int top = size - gap;
         for( i = 0; i < top; i++)
         {
            j = i+gap;
            if( age[i] > age[j] )
            {
               flag = 0;   // still not sorted and requires next iteration
               // exchange contents of age[i] and age[j]
               temp = age[i];
               age[i] = age[j];
               age[j] = temp;
            }
         }
      } while( !flag || gap > 1 );
      // sorting ends here
      cout << "Sorted list..." << endl;
      for( i = 0; i < n; i ++ )
         cout << age[i] << " ";
}
```

Run

```
How many elements to sort <max-25> ? 7
Enter age[ 0 ]: 3
Enter age[ 1 ]: 5
Enter age[ 2 ]: 9
Enter age[ 3 ]: 4
Enter age[ 4 ]: 2
Enter age[ 5 ]: 1
Enter age[ 6 ]: 6
Sorted list...
1 2 3 4 5 6 9
```

Although the algorithm for comb sort and shell sort appear to be very similar (both use a gap and a shrink factor), they do in fact perform differently. The shell sort does a complete sort (until there are no more swaps to be made) for each gap size. comb sort makes only a single pass for each gap size--it can be thought of as a more optimistic version of the shell sort. There are other differences that result from this optimism: The ideal shrink factor for shell sort is 1.7, compared with 1.3 of comb sort. The complexity obtained by plotting sorting time against the list of size n, for shell sort, appears as a step function of ($n*\log_2 n*\log_2 n$), whereas for comb sort it approximates to a flatter curve of ($n * \log_2 n$).

6.4 Multi-dimensional Arrays

Most of the scientific data can be easily modeled using multi-dimensional arrays. Such representations allow manipulation of data easily and even allow the programmer to write simple and efficient programs. Matrix is a two dimensional array and two subscripts are required to access each element.

Definition

A multidimensional array is defined as follows:

data-type array-name[*s1*][*s2*]...[*sn*];

For instance, the statement

```
int axis[3][3][2];
```

defines a three-dimensional array with the array-name `axis`.

The general format for defining a two-dimensional array is

data-type array-name[*row-size*][*column-size*];

For instance, the statements

```
int marks[4][3];
float b[3][3];
```

define arrays named `marks` and `b` respectively. The expression `marks[0][0]`, accesses the first element of the matrix `marks` and `marks[3][2]` accesses the last row and last column. The expression `b[2][1]`, accesses the 3rd row and 2nd column element of the `b` matrix. The representation of a two-dimensional array in memory is shown in Figure 6.7.

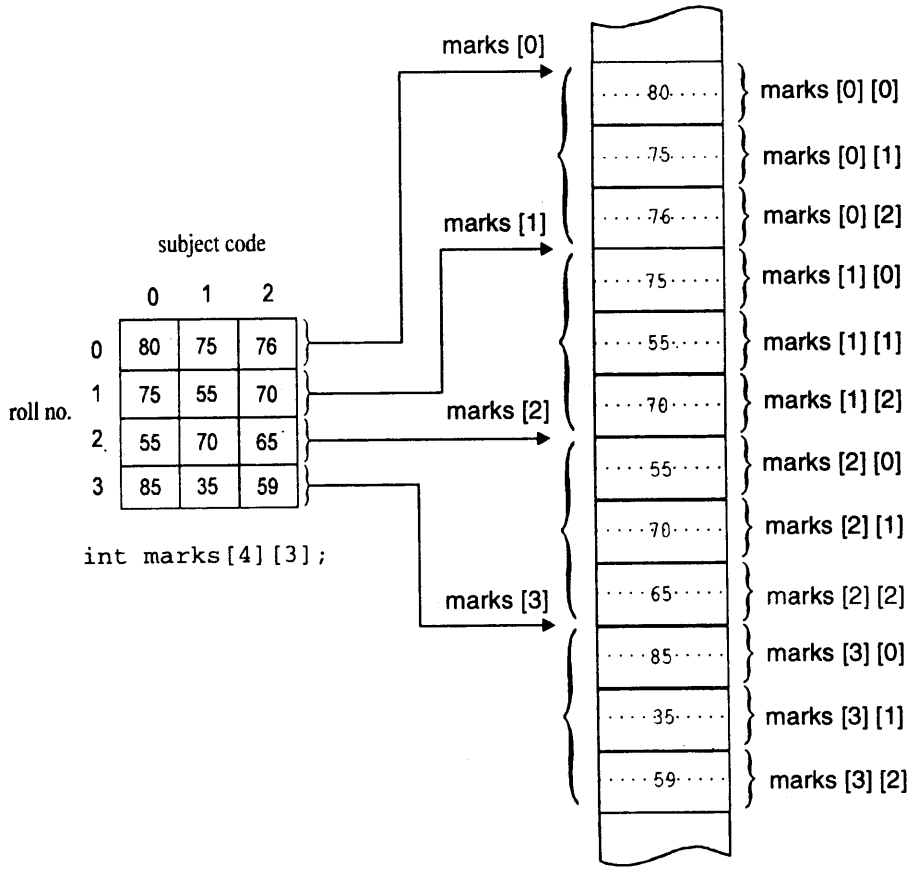

Figure 6.7: Two dimensional array to store marks

Accessing two Dimensional Array Elements

The elements of a two dimensional array can be accessed by the following statement

```
marks[i][j]
```

where `i` refers to the row number and `j` refers to the column number. The subscripts must be integer constants or variables or they can be expressions generating integer results. The program `matrix.cpp` illustrates the use of two dimensional arrays in matrix addition and subtraction.

```
// matrix.cpp: addition and subtraction of matrices
#include <iostream.h>
void main()
{
   int a[5][5], b[5][5], c[5][5];
   int i, j, m, n, p, q;
   cout << "Enter row and column size of A matrix: ";
   cin >> m >> n;
   cout << "Enter row and column size of B matrix: ";
   cin >> p >> q;
   if((m == p) && (n == q)) // check if matrices can be added
   {
      cout << "Matrices can be added or subtracted...\n";
      // Read matrix A
      cout << "Enter matrix A elements...\n";
      for( i = 0; i < m; ++i )
         for( j = 0; j < n; ++j )
            cin >> a[i][j];
      // Read matrix B
      cout << "Enter matrix B elements...\n";
      for( i = 0; i < p; i++ )
         for( j = 0; j < q; j++ )
            cin >> b[i][j];
      // Addition of two matrices: C <- A + B
      for( i = 0; i < m; i++ )
         for( j = 0; j < n; j++ )
            c[i][j] = a[i][j] + b[i][j];
      // printing summation
      cout << "Sum of A and B matrices...\n";
      for(i = 0; i < m; ++i)
      {
         for(j = 0; j < n; ++j)
            cout << c[i][j] << "  ";
         cout << endl;
      }
      // Subtraction of two matrices: C <- A - B
      for( i = 0; i < m; i++ )
         for( j = 0; j < n; j++ )
            c[i][j] = a[i][j] - b[i][j];
      // printing matrix subtraction result
      cout << "Difference of A and B matrices...\n";
      for(i = 0; i < m; ++i)
```

```
      {
         for(j = 0; j < n; ++j)
         {
            cout.width( 2 );
            cout << c[i][j] << "  ";
         }
         cout << endl;
      }
   }
}
```

Run

```
Enter row and column size of A matrix: 3 3
Enter row and column size of B matrix: 3 3
Matrices can be added or subtracted...
Enter matrix A elements...
1 2 3
4 3 1
3 1 2
Enter matrix B elements...
3 2 1
3 3 2
1 2 1
Sum of A and B matrices...
4  4  4
7  6  3
4  3  3
Difference of A and B matrices..
-2   0   2
 1   0  -1
 2  -1   1
```

Initialization at Definition

A two-dimensional array can be initialized during its definition as follows:

data-type matrix-name[*row-size*][*col-size*] = {
{ *elements of first row* },
{ *elements of second row* },
....
{ *elements of n-1 row* }
};

For instance, the statement

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

defines two dimensional array of order 3x3 and initializes all its elements. The first subscript (size of the

row) can be omitted. Hence, the above definition can be replaced by

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

The inner braces can be omitted, permitting the numbers to be written in one continuous sequence as follows:

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

It has the same effect as the earlier definitions, but it suffers from readability.

6.5 Strings

Strings are used in programming languages for storing and manipulating text, such as words, names, and sentences. It is represented as an array of characters and the end of the string is marked by the NULL (`'\0'`) character. String constants are enclosed in double quotes. For instance,

```
"Hello World"
```

is a string. A string is stored in memory by using the ASCII codes of the characters that form the string. The representation of the string `Hello World` in memory is shown in Figure 6.8.

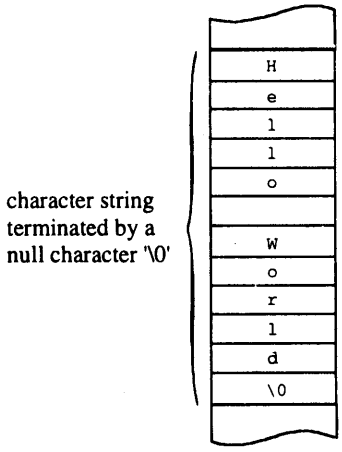

Figure 6.8: String representation in memory

Definition

An array of characters representing a string is defined as follows:

```
char array-name[size];
```

As usual, the size of the array must be an integer value. For instance, the statement

```
char name[50];
```

defines an array and reserves 50 bytes of memory for storing a set of characters. The length of this string cannot exceed 49 since, one storage location must be reserved for storing the end of the string

marker. The program name.cpp defines an array and uses it to store characters.

```
// name.cpp: read and display string
#include <iostream.h>
void main()
{
   char name[50]; // string definition
   cout << "Enter your name <49-max>: ";
   cin >> name;
   cout << "Your name is " << name;
}
```

Run

```
Enter your name <49-max>: Archana
Your name is Archana
```

In main(), the statement

```
cin >> name;
```

reads characters and stores them into the variable name. The statement

```
cout << "Your name is " << name;
```

outputs the contents of the string variable name.

Initialization at the Point of Definition

The string variable can be initialized at the point of its definition as follows:

char array-name[size] = { list of values separated by comma };

For instance, the statement

```
char month[] = { 'A','p','r','i','l', 0 };
```

defines the string variable and assigns the character 'A' to month[0], 'p' to month[1],.., 0 to month[5]. The end of the string in the above statement can also be represented as follows:

```
char month[] = { 'A','p','r','i','l', '\0' };
```

C++ offers another style for initializing an array of characters. For instance, the statement

```
char month[] = "April";
```

has the same effect as the above statements. In this case, the characters of the string are enclosed in a pair of double quotes. The compiler takes care of storing the ASCII codes of the characters of the string in memory, and also stores the NULL terminator at the end.

Special characters can also be embedded within a string as illustrated in the program succ.cpp. When manipulated using C++ I/O operators, they are interpreted as special characters and action is taken according to their predefined meaning.

```
// succ.cpp: string with special characters
#include <iostream.h>
void main()
{
   char msg[] = "C to C++\nC++ to Java\nJava to ...";
   cout << "Please note the following message: " << endl;
   cout << msg;
```

Run

```
Please note the following message:
C to C++
C++ to Java
Java to ...
```

Note that the characters \ and n used in the string definition

```
char msg[] = "C to C++\nC++ to Java\nJava to ...";
```

are treated as a new line character.

6.6 Strings Manipulations

C++ has several built-in functions such as `strlen()`, `strcat()`, `strlwr()`, etc., for string manipulation. To use these functions, the header file `string.h` must be included in the program using the statement

```
#include <string.h>
```

String Length

The string function `strlen()` returns the length of a given string. A string constant or an array of characters can be passed as an argument. The length of the string excludes the end-of-string character (NULL). The `strlen.cpp` illustrates the use of `strlen()` and user defined function to find the length of the string.

```
// strlen.cpp: computing length of string
#include <iostream.h>
#include <string.h>
void main()
{
   char s1[25];
   cout << "Enter your name: ";
   cin >> s1;
   cout << "strlen( s1 ): " << strlen(s1) << endl;
}
```

Run

```
Enter your name: Smrithi
strlen( s1 ): 7
```

String Copy

The string function `strcpy()` copies the contents of one string to another. It takes two arguments, the first argument is the destination string array and the second argument is the source string array. The source string is copied into the destination string. The program `strcpy.cpp` illustrates the use of `strcpy()` to copy a string.

```
// strcpy.cpp: copying string
#include <iostream.h>
#include <string.h>
void main()
{
   char s1[25], s2[25];
```

```
   cout << "Enter a string: ";
   cin >> s1;
   strcpy( s2, s1 );
   cout << "strcpy( s2, s1 ): " << s2;
}
```

Run

```
Enter a string: Garbage
strcpy( s2, s1 ): Garbage
```

String Concatenation

The string function `strcat()` concatenates two strings resulting in a single string. It takes two arguments, which are the destination and source strings. The destination and source strings are concatenated and the resultant string is stored in the destination (first) string. The program `strcat.cpp` illustrates the use of `strcat()` to concatenate two strings.

```
// strcat.cpp: string concatenation
#include <iostream.h>
#include <string.h>
void main()
{
   char s1[40], s2[25];
   cout << "Enter string s1: ";
   cin >> s1;
   cout << "Enter string s2: ";
   cin >> s2;
   strcat( s1, s2 );
   cout << "strcat( s1, s2 ): " << s1;
}
```

Run

```
Enter string s1: C
Enter string s2: ++
strcat( s1, s2 ): C++
```

String Comparison

The string function `strcmp()` compares two strings, character by character. It accepts two strings as parameters and returns an integer, whose value is

- < 0 if the first string is less than the second
- == 0 if both are identical
- > 0 if the first string is greater than the second

Whenever two corresponding characters in the string differ, the string which has the character with the higher ASCII value is greater. For example, consider the strings `hello` and `Hello!!`. The first character itself differs. The ASCII code for `h` is 104, while the ASCII code for `H` is 72. Since the ASCII code of `h` is greater, the string `hello` is greater than the string `Hello!`. Once a differing character is found, there is no need to compare remaining characters in the string. The program `strcmp.cpp` illustrates the use of `strcmp()` to compare two strings.

```
// strcmp.cpp: string concatenation
#include <iostream.h>
#include <string.h>
void main()
{
   char s1[25], s2[25];
   cout << "Enter string s1: ";
   cin >> s1;
   cout << "Enter string s2: ";
   cin >> s2;
   int status = strcmp( s1, s2 );
   cout << "strcmp( s1, s2 ): ";
   if( status == 0 )
      cout << s1 << " is equal to " << s2;
   else
      if( status > 0 )
         cout << s1 << " is greater than " << s2;
      else
         cout << s1 << " is less than " << s2;
}
```

Run

```
Enter string s1: Computer
Enter string s2: Computing
strcmp( s1, s2 ): Computer is less than Computing
```

String to Upper/Lower Case

The functions `strlwr()` and `strupr()` convert a string to lower-case and upper-case respectively and return the address of the converted string. The program `uprlwr.cpp` illustrates the conversion of string to lower and upper cases.

```
// uprlwr.cpp: converting string to upper or lower case
#include <iostream.h>
#include <string.h>
void main()
{
   char s1[25], temp[25];
   cout << "Enter a string: ";
   cin >> s1;
   strcpy( temp, s1 );
   cout << "strupr( temp ): " << strupr( temp ) << endl;
   cout << "strlwr( temp ): " << strlwr( temp ) << endl;
}
```

Run

```
Enter a string: Smrithi
strupr( temp ): SMRITHI
strlwr( temp ): smrithi
```

6.7 Arrays of Strings

An array of strings is a two dimensional array of characters and is defined as follows:

char array-name[row_size][column_size];

For instance, the statement

```
char person[10][15];
```

defines an array of string which can store names of 10 persons and each name cannot exceed 14 characters; 1 character is used to represent the end of a string. The name of the first person is accessed by the expression `person[0]`, and the second person by `person[1]`, and so on. The individual characters of a string can also be accessed. For instance, the first character of the first person is accessed by the expression `person[0][0]` and the fifth character in the 3rd `person`'s name is accessed by `person[2][4]`. The program `names.cpp` illustrates the manipulation of an array of strings.

```
// names.cpp: array of strings storing names of the persons
#include <iostream.h>
#include <string.h>
const int LEN = 15;
void main()
{
   int i, n;
   char person[10][LEN];
   cout << "How many persons ? ";
   cin >> n;
   for( i = 0; i < n; i++ )
   {
      cout << "Enter person" << i+1 << " name: ";
      cin >> person[i];
   }
   cout<< "-------------------------------------------------------------\n";
   cout << "P#    Person Name   Length   In lower case  In UPPER case\n";
   cout<< "-------------------------------------------------------------\n";
   for( i = 0; i < n; i++ )
   {
      cout.width( 2 );
      cout << i+1;
      cout.width( LEN );
      cout << person[i] << "    ";
      cout.width( 2 );
      cout << strlen( person[i] ) << "    ";
      cout.width( LEN );
      cout << strlwr(person[i]);
      cout.width( LEN );
      cout << strupr(person[i]) << endl;
   }
   cout<< "-------------------------------------------------------------\n";
}
```

Run

```
How many persons ? 5
```

```
Enter person1 name: Anand
Enter person2 name: Viswanath
Enter person3 name: Archana
Enter person4 name: Yadunandan
Enter person5 name: Mallikarjun
-------------------------------------------------------
P#    Person Name   Length   In lower case  In UPPER case
-------------------------------------------------------
 1          Anand      5             anand         ANAND
 2      Viswanath      9         viswanath      VISWANATH
 3        Archana      7           archana        ARCHANA
 4     Yadunandan     10        yadunandan     YADUNANDAN
 5    Mallikarjun     11       mallikarjun    MALLIKARJUN
-------------------------------------------------------
```

An array of string can be initialized at the point of its definition as follows:

char array-name[row_size][column_size] = { "row1 string", "row2-string", ... };

It can also be defined as

char array-name[row_size][column_size] =
{ { row1 string characters}, { row2 string characters}, .. };

For instance, the statement

```
char person[][12]={"Anand","Viswanath","Archana","Yadunandan","Mallikarjun"};
```

defines an array of strings and initializes them at the point of definition (see Figure 6.9 for the memory representation). The above statement is equivalent to

```
char person[5][12]={"Anand","Viswanath","Archana","Yadunandan","Mallikarjun"};
```

The second dimension must be specified explicitly in the array definition, otherwise, the compiler generates an error message. However, the first dimension can be skipped; the compiler computes this value based on the number of values specified in the initialization list. This rule applies only when the initialization appears at the point of definition.

	0	1	2	3	4	5	6	7	8	9	10	11	
0	A	n	a	n	d	\0							person[0]
1	V	i	s	w	a	n	a	t	h	\0			person[1]
2	A	r	c	h	a	n	a	\0					person[2]
3	Y	a	d	u	n	a	n	d	a	n	\0		person[3]
4	M	a	l	l	i	k	a	r	j	u	n	\0	person[4]

Figure 6.9: Array of strings representation in memory

6.8 Evaluation Order / Undefined Behaviors

The order of evaluation of sub-expressions within an expression is undefined. Consider the following segment of code:

```
int i = 0;
v[i] = i++;
```

The second statement can be evaluated either as:

```
v[0] = 0;
```

or

```
v[1] = 0;
```

The compiler can generate better code in the absence of restrictions on the expression evaluation order. It can take advantage of underlying hardware architecture and generate the most optimal code. The compiler can warn about such ambiguities. Unfortunately, most compilers do not report a warning about such ambiguities.

The operators

```
&&  ||
```

guarantees that their left-hand side operand is evaluated first before their right-hand side operand. For instance, in the statement,

```
x = (y = 5, y+1);
```

the expression (`y = 5, y+1`), the comma operator first assigns 5 to `y` and then evaluates the right-hand side operand and the resulting value 6 is assigned to the `x` variable. Note that the sequencing operator comma (,) is logically different from the comma used to separate arguments in a function call. Consider the following statements:

```
f1( a[i], i++ );      // two arguments
f2( (a[i], i++) );    // one argument
```

The call of `f1()` has two arguments, `a[i]` and `i++`, and the order of evaluation of the argument is undefined. However, most compilers follow evaluation of arguments at a function call from right to left. The function

```
f1( int a, int b )
{
   cout << a << " " << b;
}
```

when invoked as

```
f1( a[i], i++ );
```

where `a[] = { 1, 2, 3, 4, 5 }` and `i = 0`. The output will be 2 and 0. The parameters evaluated are passed in the following order:

1. The contents of the variable `i` whose value is 0 is assigned to `b`, and then the expression `i++` will be evaluated, thereby `i` becomes 1.
2. The value of `a[i]` (now `i` holds the value 1) is 2 and is assigned to the variable `a`.

Review Questions

6.1 What are arrays ? Explain how they simplify programming with suitable examples.

6.2 Explain how comb sort algorithm is superior over bubble sort. What is their time complexity. Hint: time complexity is measured in terms of number of elements compared, since comparison operation is the active operation in any sorting algorithm.

6.3 What are the side-effects of the following statements:

```
int a[100];
```

```
a[0] = 20;
a[100] = 200;
cout << a[101];
a[-1] = 5;
cout << a[-1];
```

Does the compiler reports an error when illegal accesses are made to an array ?

6.4 What are multi-dimensional arrays ? Explain their syntax and mechanism for accessing their elements.

6.5 Write an interactive program for calculating grades of N students from 3 tests and present the result in the following format:

```
-------------------------------------------------------------
   Sl.NO.          SCORES          AVERAGE      GRADE
-------------------------------------------------------------
     xx          xx  xx  xx           xx          x
-------------------------------------------------------------
```

6.6 Write a program for computing the norm of the matrix.

6.7 Can arrays be initialized at the point of their definition ? If yes, explain its syntax with suitable examples ?

6.8 Write a program to find the symmetry of the matrix.

6.9 What are strings ? Are they standard or derived data type ? Write an interactive program to check whether a given string is palindrome or not. What happens if the end-of-string character is missing ?

6.10 Write a program to sort integer numbers using shell sort and compare its time complexity with that of the comb sort.

6.11 Write a program for computing mean(m), variance, and standard deviation(s) of a set of numbers using the following formulae:

$$\text{mean} = m = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\text{variance} = \frac{1}{n}\sum_{i=1}^{n} (x_i - m)^2$$

$$s = \sqrt{\text{variance}}$$

6.12 Write a program to find the transpose of a matrix. (The transpose can be obtained by interchanging the elements of rows and columns).

6.13 Write a program to find the saddle points in a matrix. It is computed as follows: Find out the smallest element in a row. The saddle point exists in a row if an element is the largest element in that corresponding column. For instance, consider the following matrix:

```
 7   5   6
10   2   3
 1   3   3
```

The saddle point results are as listed below:
In row 1, saddle point exists at column 2.
In row 2, saddle point does not exist.
In row 3, saddle point does not exist.

6 14 Write an interactive program to multiply two matrices and print the result in a matrix form.

7

Modular Programming with Functions

7.1 Introduction

It is difficult to implement a large program even if its algorithm is available. To implement such a program with ease, it should be split into a number of independent tasks, which can be easily designed, implemented, and managed. This process of splitting a large program into small manageable tasks and designing them independently is popularly called *modular programming* or *divide-and-conquer technique*. Large programs are more prone to errors and it is difficult to locate and isolate errors that creep into them. A repeated group of instructions in a program can be organized as a *function*. It can be invoked instead of having the same pattern of code wherever it is required as shown in Figure 7.1.

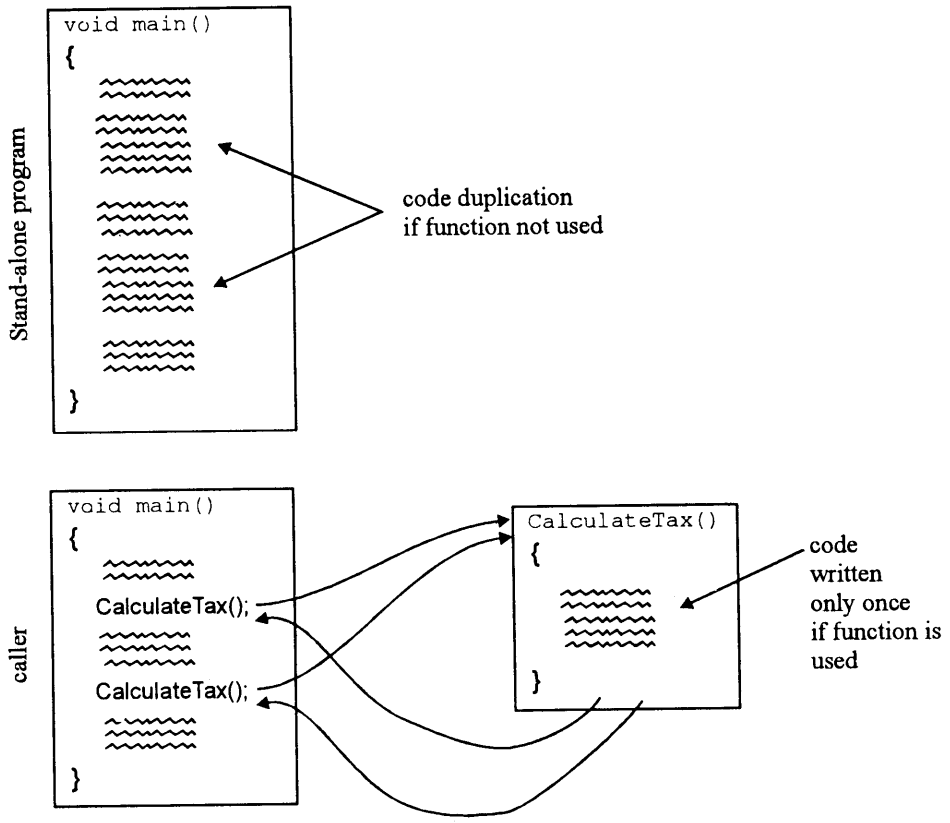

Figure 7.1: Functions for eliminating redundancy of code

A function is a set of program statements that can be processed independently. A function can be invoked which behaves as though its code is inserted at the point of the function call. The communication between a *caller* (calling function) and *callee* (called function) takes place through parameters. The functions can be designed, developed, and implemented independently by different programmers. The independent functions can be grouped to form a software library. Functions are independent because variable names and labels defined within its body are local to it. The use of functions offer flexibility in the design, development, and implementation of the program to solve complex problems. The advantages of functions include the following:

- Modular programming
- Reduction in the amount of work and development time
- Program and function debugging is easier
- Division of work is simplified due to the use of divide-and-conquer principle
- Reduction in size of the program due to code reusability
- Functions can be accessed repeatedly without redevelopment, which in turn promotes reuse of code
- *Library of functions* can be implemented by combining well designed, tested, and proven functions

The program `tax1.cpp` computes the tax amount of two persons based on their annual salary without the use of functions.

```
// tax1.cpp: tax calculation without using function
#include <iostream.h>
void main()
{
   char Name[25];
   double Salary, Tax;
   cout << "Enter name of the 1st person: ";
   cin >> Name;
   cout << "Enter Salary: ";
   cin >> Salary;
   if( Salary <= 90000 )
      Tax = Salary * 12.5 / 100;
   else
      Tax = Salary * 18.0 / 100;
   cout << "The tax amount for " << Name << " is: " << Tax << endl;
   cout << "Enter name of the 2nd person: ";   cin >> Name;
   cout << "Enter Salary: ";    cin >> Salary;
   if( Salary <= 90000 )
      Tax = Salary * 12.5 / 100;
   else
      Tax = Salary * 18.0 / 100;
   cout << "The tax amount for " << Name << " is: " << Tax << endl;
}
```

Run

```
Enter name of the 1st person: Rajkumar
Enter Salary: 130000
The tax amount for Rajkumar is: 23400
Enter name of the 2nd person: Savithri
Enter Salary: 90000
The tax amount for Savithri is: 11250
```

Multiple copies of the same pattern of code can be eliminated by grouping repeated statements together to generate a function `CalculateTax()`, as illustrated in the program `tax2.cpp`.

```
// tax2.cpp: tax calculation using function
#include <iostream.h>
void CalculateTax()
{
   char Name[25];
   double Salary, Tax;
   cout << "Enter name of the person: ";
   cin >> Name;
   cout << "Enter Salary: ";
   cin >> Salary;
   if( Salary <= 90000 )
      Tax = Salary * 12.5 / 100;
   else
      Tax = Salary * 18.0 / 100;
   cout << "The tax amount for " << Name << " is: " << Tax << endl;
}
void main()
{
    CalculateTax();
    CalculateTax();
}
```

Run

```
Enter name of the person: Rajkumar
Enter Salary: 130000
The tax amount for Rajkumar is: 23400
Enter name of the person: Savithri
Enter Salary: 90000
The tax amount for Savithri is: 11250
```

In `main()`, the statement

```
CalculateTax();
```

is invoked twice to calculate tax for two persons. It computes the tax amount and displays it. The same function can be invoked to calculate tax amounts for a large number of people using a loop construct.

7.2 Function Components

Every function has the following elements associated with it:

- Function declaration or prototype.
- Function parameters (formal parameters)
- Combination of function declaration and its definition.
- Function definition (function declarator and a function body).
- `return` statement.
- Function call.

A function can be executed using a *function call* in the program. The various components associated with functions are shown in Figure 7.2.

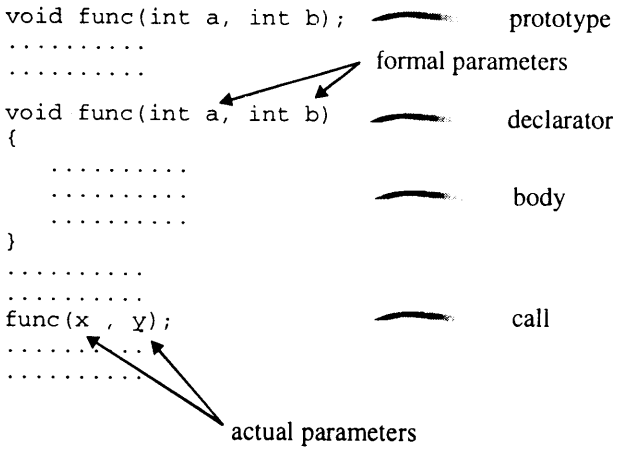

Figure 7.2: Components of a function

The program `max1.cpp` illustrates the various components of a function. It computes the maximum of two integer numbers.

```
// max1.cpp: maximum of two integer numbers
#include <iostream.h>
int max( int x, int y );          // prototype
void main()                       // function caller
{
   int a, b, c;
   cout << "Enter two integers <a, b>: ";
   cin >> a >> b;
   c = max( a, b );               // function call
   cout << "max( a, b): " << c << endl;
}
int max( int x, int y )           // function definition
{
   // all the statements enclosed in braces forms body of the function
   if( x > y )
      return x;                   // function return
   else
      return y;                   // function return
}
```

Run

```
Enter two integers <a, b>: 20 10
max( a, b ): 20
```

As discussed earlier, `main()` is a function, so it is not surprising that `max()` which is also a function, appears similar to `main()`. The only special feature about `main()` is that it is always executed first. It does not matter whether `main()` is the first function in the program listing or is placed elsewhere in the program; it will always be the first one to execute.

There are five elements involved in using a function: the function prototype, the function definition, the function call, the function parameters, and the function return.

Function Prototype

The first function related statement in `max1.cpp` is the function prototype. This is the line before the beginning of `main()`:

```
int max( int x, int y );          // prototype
```

It provides the following information to the compiler:

- The name of the function,
- The type of the value returned (optional; default is an integer),
- The number and the types of the arguments that must be supplied in a call to the function.

Function prototyping is one of the key improvements added to the C++ functions. When a function call is encountered, the compiler checks the function call with its prototype so that correct argument types are used. The compiler informs the user about any violations in the actual parameters that are to be passed to a function.

A function prototype is a declaration statement which has the following syntax:

```
ret_val function_name( argument1, argument2, ... , argumentn );
```

The `ret_val` specifies the datatype of the value in the return statement. The function can return any data-type; if there is no return value, a keyword `void` is placed before the function name. In a function without any return value, a dummy return statement can be included before the closing brace. A program can have more than one `return` statements. (Note: `return` is a keyword. The statement `return 0;` is sufficient in place of the `return(0);`). The number of arguments to a function can be fixed or variable. The function declaration terminates with a semicolon.

Consider the prototype statement

```
int max( int x, int y );          // prototype
```

It informs the compiler that the function `max` has two arguments of type integer (the list of data types separated by commas form the argument list). The function `max()` returns an integer value; the compiler knows how many bytes to retrieve and how to interpret the value returned by the function. Function declarations are also called *prototype*, since they provide a model or blue print for the function. C++ makes prototyping mandatory if functions are defined after the function `main`. C++ assumes `void` type in case no arguments are specified in the argument list; the default return type is an integer.

Function Definition

The function itself is referred to as function definition. The first line of the function definition is known as *function declarator* and is followed by the *function body*. Figure 7.3 shows that the declarator and the function body make up the function definition. The declarator and declaration must use the same function name, the number of arguments, the arguments type and the return type. No other function definitions are allowed within a function definition.

The body of the function is enclosed in braces. C++ allows the definition to be placed anywhere in the program. If the function is defined before its invocation, then its prototypes declaration is optional.

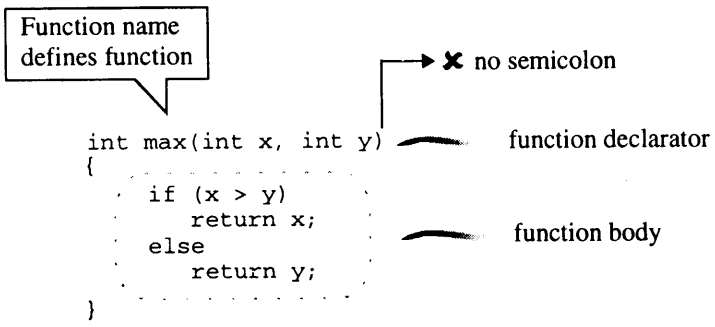

Figure 7.3: Function definition

Function Call

A function is a dormant entity, which gets life only when a call to the function is made. A function call is specified by the function name followed by the arguments enclosed in parentheses and terminated by a semicolon. The return type is not mentioned in the function call. For instance, in the function `main()` of the program `max1.cpp`, the statement

```
c = max( a, b );                    // function call
```

invokes the function `max()` with two integer parameters. Executing the call statement causes the control to be transferred to the first statement in the function body and after execution of the function body the control is returned to the statement following the function call. The `max()` returns the maximum of the parameters `a` and `b`. The return value is assigned to the local variable `c` in `main()`.

Function Parameters

The parameters specified in the function call are known as *actual parameters* and those specified in the function declarator are known as *formal parameters*. For instance, in `main()`, the statement

```
c = max( a, b );                    // function call
```

passes the parameters (actual parameters) `a` and `b` to `max()`. The parameters `x` and `y` are formal parameters. When a function call is made, a one-to-one correspondence is established between the actual and the formal parameters. In this case, the value of the variable `a` is assigned to the variable `x` and that of `b` is assigned to `y`. The scope of formal parameters is limited to its function only.

Function Return

Functions can be grouped into two categories: functions that do not have a return value (`void` functions) and functions that have a return value. The statements

```
return x;                           // function return
```

and

```
return y;                           // function return
```

in function `max()` are called function return statements. The caller must be able to receive the value returned by the function (but not mandatory). In the statement

```
c = max( a, b );                    // function call
```

the value returned by the function `max()` is assigned to the local variable `c` in `main()`. Figure 7.4 shows the function `max()` returning a value to the caller.

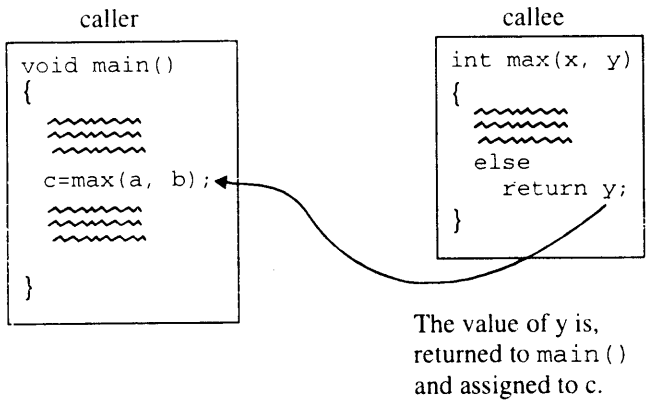

Figure 7.4: Function returning a value

The return statement in a function need not be at the end of the function. It can occur anywhere in the function body and as soon as it is encountered, execution control will be returned to the caller.

A function that does not return anything is indicated by the keyword `void`. It has the following form:

```
void FunctionName( ParameterList )
{
   statement(s);
   return;  // return is optional
}
```

In void functions, the use of `return` statement is optional.

Elimination of the Function Prototype

The function declaration can be eliminated by defining the function before calling it. The program `max2.cpp` illustrates this concept.

```
// max2.cpp: maximum of two integer numbers
#include <iostream.h>
int max( int x, int y )          // function definition
{
   // all the statements enclosed in braces forms body of the function
   if( x > y )
      return x;                  // function return
   else
      return y;                  // function return
}
void main()                       // function caller
{
   int a, b, c;
   cout << "Enter two integers <a, b>: ";
   cin >> a >> b;
   c = max( a, b );               // function call
```

```
   cout << "max( a, b): " << c << endl;
}
```

Run

```
Enter two integers <a, b>: 20 10
max( a, b): 20
```

The definition of `max()` occurs before it is invoked in `main()`, eliminating the need for a function prototype. In the case of a program having a large number of functions, the programmer has to arrange the functions, such that they are defined before they are called by any other function.

7.3 Passing Data to Functions

The entity used to convey the message to a function is the function argument. It can be a numeric constant, a variable, multiple variables, user defined data type, etc.

Passing Constants as Arguments

The program `chart1.cpp` illustrates the passing of a numeric constant as an argument to a function. This constant argument is assigned to the formal parameter which is processed in the function body.

```
// chart1.cpp: Percentage chart by passing numeric value
#include <iostream.h>
void PercentageChart( int percentage );
void main()
{
   cout << "Sridevi :  ";
   PercentageChart( 50 );
   cout << "Rajkumar:  ";
   PercentageChart( 84 );
   cout << "Savithri:  ";
   PercentageChart( 79 );
   cout << "Anand   :  ";
   PercentageChart( 74 );
}
void PercentageChart( int percentage )
{
   for( int i = 0; i < percentage/2; i++ )
      cout << '\xCD';      // double line character (see ASCII table)
   cout << endl;
}
```

Run

```
Sridevi :  ==========================
Rajkumar:  ========================================
Savithri:  ====================================
Anand   :  ==================================
```

In `main()`, the statement

```
PercentageChart( 84 );
```

invokes the function `PercentageChart` with the integer constant `84` to draw a chart. It draws a

horizontal line, made up of the double-line graphic character (`'\xCD'`) on the screen.

In the function definition, the variable name `percentage` is placed between the parentheses following the function name `PercentageChart`. The invocation of this function by the statement

```
PercentageChart( 84 );
```

ensures that the numeric constant `84` is assigned to the variable `percentage` as shown in Figure 7.5.

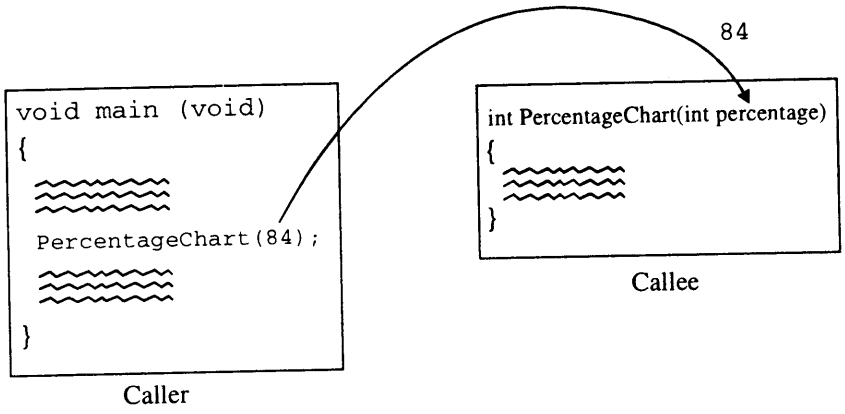

Figure 7.5: Passing value to a function

Passing Variables as Arguments

Similar to constants, variables can also be passed as arguments to a function. The program `chart2.cpp` illustrates the mechanism of passing a variable as an argument to a function.

```
// chart2.cpp: Percentage chart by passing variables
#include <iostream.h>
void PercentageChart( int percentage );
void main()
{
   int m1, m2, m3, m4;
   cout << "Enter percentage score of Sri, Raj, Savi, An: ";
   cin >> m1 >> m2 >> m3 >> m4;
   cout << "Sridevi :  ";
   PercentageChart( m1 );
   cout << "Rajkumar:  ";
   PercentageChart( m2 );
   cout << "Savithri:  ";
   PercentageChart( m3 );
   cout << "Anand   :  ";
   PercentageChart( m4 );
}
void PercentageChart( int percentage )
{
   for( int i = 0; i < percentage/2; i++ )
      cout << '\xCD';      // double line character (see ASCII table)
   cout << endl;
}
```

Run

```
Enter percentage score of Sri, Raj, Savi, An: 55 92 83 67
Sridevi :  ==========================
Rajkumar:  ==============================================
Savithri:  ========================================
Anand   :  ================================
```

In `main()`, the statement

```
PercentageChart( m2 );
```

invokes the function `PercentageChart`. It draws a horizontal line, made up of the double-line graphic character (`'xCD'`) on the screen. It ensures that the contents of the variable `m2` is assigned to the variable `percentage` as shown in Figure 7.6. Note that the names of the parameters in the calling and called functions can be the same or different, since the compiler treats them as different variables.

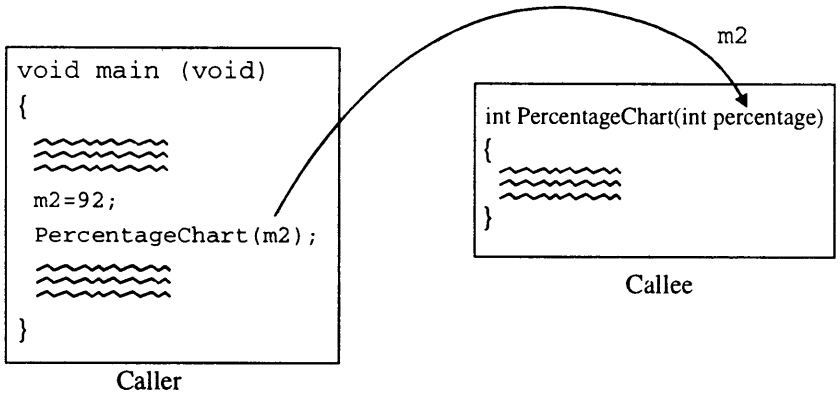

Figure 7.6: Variable used as argument

Passing Multiple Arguments

C++ imposes no limitation on the number of arguments that can be passed to a function. The program `chart3.cpp` passes two arguments to the function `PercentageChart()`, whose purpose is to draw various style charts on the screen.

```
// chart3.cpp: Percentage chart by passing multiple variables
#include <iostream.h>
void PercentageChart( int percentage, char style );
void main()
{
   int m1, m2, m3, m4;
   cout << "Enter percentage score of Sri, Raj, Savi, An: ";
   cin >> m1 >> m2 >> m3 >> m4;
   cout << "Sridevi :  ";
   PercentageChart( m1, '*' );
   cout << "Rajkumar:  ";
```