

```

PercentageChart( m2, '\xCD' );
cout << "Savithri: ";
PercentageChart( m3, '~');
cout << "Anand : ";
PercentageChart( m4, '!' );
}
void PercentageChart( int percentage, char style )
{
    for( int i = 0; i < percentage/2; i++ )
        cout << style;
    cout << endl;
}

```

Run

```

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

```

The process of passing two parameters is similar to passing a single parameter. The value of the first *actual* parameter in the *caller* (calling function) is assigned to the first *formal* parameter in the *callee* (called function), and the value of the second actual parameter is assigned to the second formal parameter, as shown in Figure 7.7. Of course, more than two parameters can be passed in the same way.

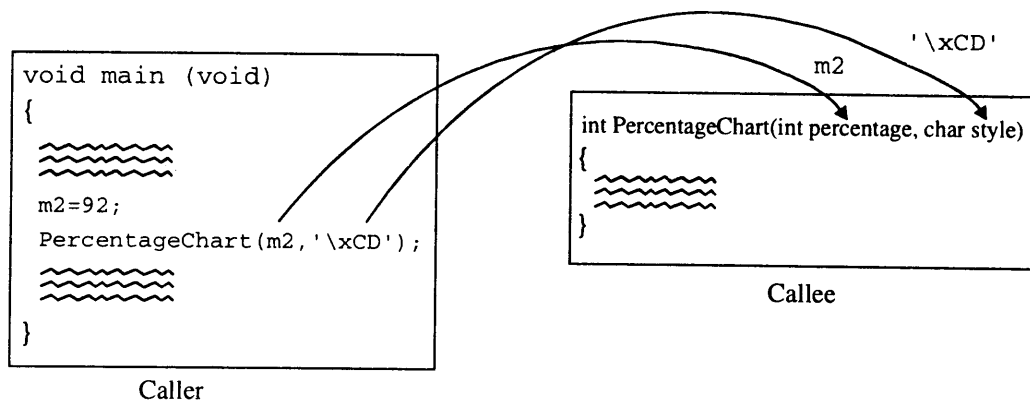


Figure 7.7: Multiple arguments passed to a function

7.4 Function Return Data Type

The return value can be a constant, a variable, a user-defined data structure, a general expression (reducible expressions), a pointer to a function or a function call (in which case the call must return a value). C++ does not place any restriction on the type of return value, except that it cannot be an array (a pointer to an array can be returned. A function can return an array that is a part of a structure).

202 Mastering C++

```
// ifact.cpp: factorial computation Returns a long integer value
#include <iostream.h>
long fact( int n )
{
    long result;
    if( n == 0 )
        result = 1; // factorial of zero is one
    else
    {
        result = 1;
        for( int i = 2; i <= n; i++ )
            result = result * i;
    }
    return result;
}
void main( void )
{
    int n;
    cout << "Enter the number whose factorial is to be found: ";
    cin >> n;
    cout << "The factorial of " << n << " is " << fact(n) << endl;
}
```

Run

```
Enter the number whose factorial is to be found: 5
The factorial of 5 is 120
```

The definition before `main()` indicates that the function `fact` takes an integer argument and returns a long datatype. It ensures that the correct value is returned by defining the appropriate data type (i.e., a long variable) and placing it in the return statement. Suppose that the variable `result` was defined as an integer, the compiler performs the necessary type conversion (i.e., to type long) and returns a value of type long, irrespective of the data variable to which the return value is assigned.

A function with a return value can be placed as an individual statement (i.e., the return value *need* not be assigned to any variable(s)). An example is given below.

```
int SumTwo( int n1, int n2 ) // n1 and n2 are the parameters
{
    return n1 + n2;
}
```

When a function has nothing specific to return or take, it is indicated by `void`. Typically, such functions are called **void functions**. The following is the prototype of a void function:

```
void func( void );
```

However, the keyword `void` is optional. C++ maintains strict type checking and an empty argument list is interpreted as the absence of any parameters.

Limitation of return

A key limitation of the `return` statement is that it can be used to return only *one item* from a function. An alternative method to overcome this limitation is to use parameters as a media of communication between calling and called functions.

7.5 Library Functions

Library functions are shipped along with the compilers. They are predefined and pre-compiled into library files, and their prototypes can be found in the files with `.h` (called header files) as their extension in the `include` directory. The definitions are available in the form of object codes in the files with `.lib` (called library files) as their extension in the `lib` directory. In order to make use of a library function, include the corresponding header file. Once the header file is included, any function available in that library can be invoked. The linker will add such functions to a calling program by extracting them from an appropriate function library. Some of the library calls are `sqrt()`, `pow()` (declared in the header file `math.h`), `strlen()`, `strcat()`, `strcpy()`, and `strncpy()` (declared in `string.h`). In case of user defined functions, the prototype and definitions of the functions must be a part of a program module. The program `namelen.cpp` illustrates the use of library functions.

```
// namelen.cpp: use of string library functions
#include <iostream.h>
#include <string.h>           // string function header file
void main()
{
    char name[ 20 ];
    cout << "Enter your name: ";
    cin >> name;
    int len = strlen( name ); // strlen returns the length of name
    cout << "Length of your name = " << len;
}
```

Run

```
Enter your name: Rajkumar
Length of your name = 8
```

Note that, the statement

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

informs the compiler to include the prototypes of the string related functions. The statement

```
int len = strlen( name );
```

invokes the library function `strlen` and assigns the length of the string stored in the variable `name` to the variable `len`.

The calls may be mathematical, such as `sin()`, `cos()`, `log10()` or may even include functions to round a value or truncate a resultant value. The program `maths.cpp` accesses mathematical functions.

```
// maths.cpp: Use of library function calls to round and truncate a result
#include <iostream.h>
#include <math.h>
void main(void)
{
    float num, num1, num2;
    cout << "Enter any fractional number: ";
    cin >> num;
    num1 = ceil( num ); // rounds up
```

```

    num2 = floor( num ); // rounds down
    cout << "ceil( " << num << " ) = " << num1 << endl;
    cout << "floor( " << num << " ) = " << num2 << endl;
}

```

Run1

```

Enter any fractional number: 2.9
ceil( 2.9 ) = 3
floor( 2.9 ) = 2

```

Run2

```

Enter any fractional number: 2.1
ceil( 2.1 ) = 3
floor( 2.1 ) = 2

```

Library functions improve the program design, reduce debugging and testing time, thereby reducing the amount of work needed for the development of the program. These functions are certainly better programmed, tested, and well proved. Hence, the use of library functions increases the program reliability and reduces the complexity.

7.6 Parameter Passing

Parameter passing is a mechanism for communication of data and information between the calling function (caller) and the called function (callee). It can be achieved either by passing the value or address of the variable. C++ supports the following three types of parameter passing schemes:

- ◆ Pass by Value
- ◆ Pass by Address
- ◆ Pass by Reference (only in C++)

The parameters used to transfer data to a function are known as *input-parameters* and those used to transfer the result to the caller are known as *output-parameters*. The parameters used to transfer data in both the directions are called *input-output parameters*.

Parameters can be classified as formal parameters and actual parameters. The formal parameters are those specified in the function declaration and function definition. The actual parameters are those specified in the function call. The following conditions must be satisfied for a function call:

- ◆ the number of arguments in the function call and the function declarator must be the same.
- ◆ the data type of each of the arguments in the function call should be the same as the corresponding parameter in the function declarator statement. However, the names of the arguments in the function call and the parameters in the function definition can be different.

Pass by Value

The default mechanism of parameter passing is called pass by value. Pass-by-value mechanism does not change the contents of the argument variable in the calling function (caller), even if they are changed in the called function (callee); because the content of the actual parameter in a caller is copied to the formal parameter in the callee. The formal parameter is stored in the local data area of the callee. Changes to the parameter within the function will effect only the copy (formal parameters), and will have no effect on the actual argument. It is illustrated in the program `swap1.cpp`. Most of the functions discussed earlier fall under the category *pass-by-value* parameter passing.

```

// swap1.cpp: swap integer values by value
#include <iostream.h>
void swap( int x, int y )
{
    int t; // temporary used in swapping
    cout<<"Value of x and y in swap before exchange: "<< x <<" "<< y << endl;
    t = x;
    x = y;
    y = t;
    cout<<"Value of x and y in swap after exchange: "<< x <<" " << y << endl;
}
void main()
{
    int a, b;
    cout << "Enter two integers <a, b>: ";
    cin >> a >> b;
    swap( a, b );
    cout << "Value of a and b on swap( a, b ) in main(): " << a << " " << b;
}

```

Run

```

Enter two integers <a, b>: 10 20
Value of x and y in swap before exchange: 10 20
Value of x and y in swap after exchange: 20 10
Value of a and b on swap( a, b ) in main(): 10 20

```

In main(), the statement

```
swap( x, y )
```

invokes the function swap() and assigns the contents of the actual parameters a and b to the formal parameters x and y respectively. In the swap() function, the input parameters are exchanged, however it is not reflected in the caller; actual parameters a and b do not get modified (see Figure 7.8).

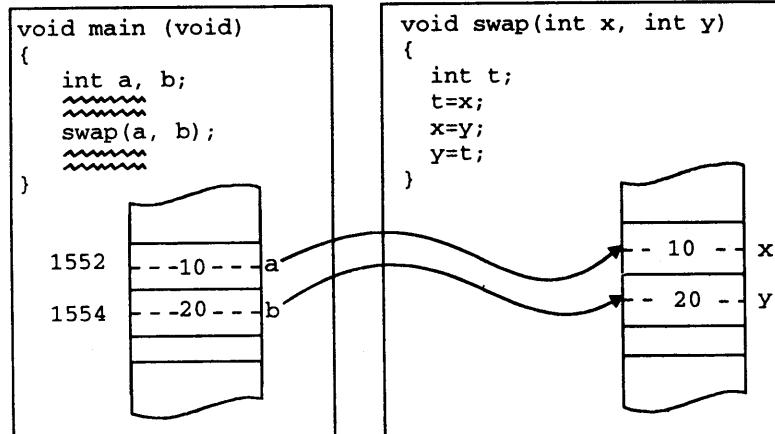


Figure 7.8: Parameter passing by value

Pass by Address

C++ provides another means of passing values to a function known as pass-by-address. Instead of passing the value, the address of the variable is passed. In the function, the address of the argument is copied into a memory location instead of the value. The de-referencing operator is used to access the variable in the called function.

```
// swap2.cpp: swap integer values by pointers
#include <iostream.h>
void swap( int * x, int * y )
{
    int t; // temporary used in swapping
    t = *x;
    *x = *y;
    *y = t;
}
void main()
{
    int a, b;
    cout << "Enter two integers <a, b>: ";
    cin >> a >> b;
    swap( &a, &b );
    cout << "Value of a and b on swap( a, b ): " << a << " " << b;
}
```

Run

Enter two integers <a, b>: 10 20
 Value of a and b on swap(a, b): 20 10

In main(), the statement

```
swap( &x, &y )
```

invokes the function swap and assigns the address of the actual parameters a and b to the formal parameters x and y respectively.

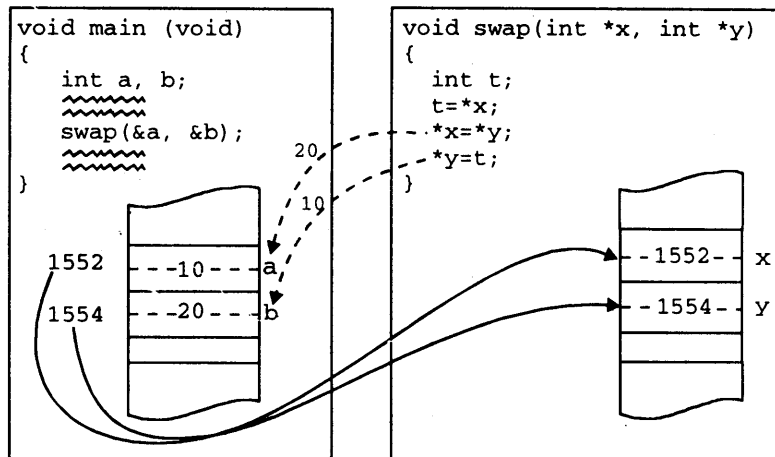


Figure 7.9: Parameter passing by address

In `swap()`, the statement

```
t = *x;
```

assigns the contents of the memory location pointed to by the pointer (address) stored in the variable `x` (It is effectively accessing the contents of the actual variable `a` in the caller. Similarly, the parameter `y` holds the address of the parameter `b`. Any modification to the memory contents using these addresses will be reflected in the caller; the actual parameters `a` and `b` get modified (see Figure 7.9).

Pass by Reference

Passing parameters by reference has the functionality of pass-by-pointer and the syntax of call-by-value. Any modifications made through the formal pointer parameter is also reflected in the actual parameter. Therefore, the function body and the call to it is identical to that of call-by-value, but has the effect of call-by-pointer.

To pass an argument by reference, the function call is similar to that of call by value. In the function declarator, those parameters, which are to be received by reference must be preceded by the `&` operator. The reference type formal parameters are accessed in the same way as normal value parameters. However, any modification to them will also be reflected in the actual parameters. The program `swap3.cpp` illustrates the mechanism of passing parameters by reference.

```
// swap3.cpp: swap integer values by reference
#include <iostream.h>
void swap( int & x, int & y )
{
    int t;    // temporary used in swapping
    t = x;
    x = y;
    y = t;
}
void main()
{
    int a, b;
    cout << "Enter two integers <a, b>: ";
    cin >> a >> b;
    swap( a, b );
    cout << "Value of a and b on swap( a, b ): " << a << " " << b;
}

```

Run

```
Enter two integers <a, b>: 10 20
Value of a and b on swap( a, b ): 20 10

```

In `main()`, the statement

```
swap( a, b );
```

is translated into

```
swap( &a, &b );
```

internally during compilation. The function declarator

```
void swap( int &a, int &b )
```

indicates that the formal parameters are of reference type and hence, they must be bound to the memory

location of the actual parameter. Thus, any access made to the reference formal parameters in the `swap()` function refers to the actual parameters. The following statements in the body of the `swap()` function:

```
t = x;
x = y;
y = t;
```

(as treated by the compiler) have the following interpretation internally:

```
t = *x; // store the value pointed by x into t
*x = *y; // store the value pointed by y into location pointed by x
*y = t; // store the value hold by 't' into location pointed by y
```

This is because, the formal parameters are of reference type and therefore the compiler treats them similar to pointers but does not allow the modification of the pointer value (cannot be made to point to some other variable). Changes made to the formal parameters `x` and `y` reflect on the actual parameters `a` and `b` (see Figure 7.10).

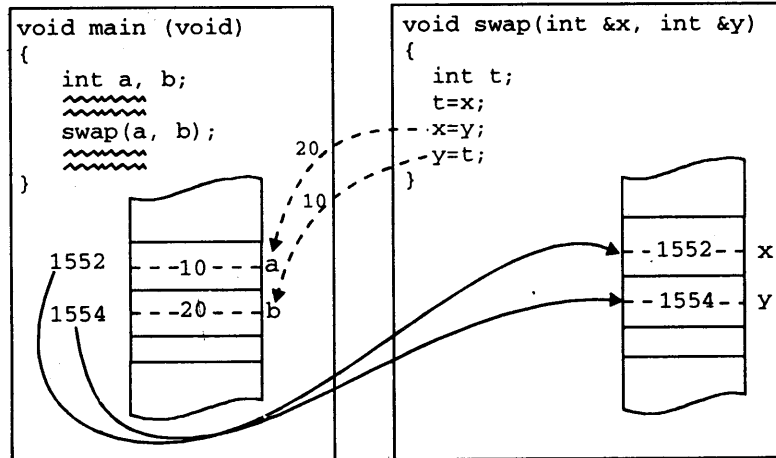


Figure 7.10: Parameter passing by reference

The following points can be noted about reference parameters:

- A reference can never be null, it must always refer to a legitimate object (variable).
- Once established, a reference can never be changed so as to make it point to a different object.
- A reference does not require any explicit mechanism to dereference the memory address and access the actual data value.

Note

Procedures can be implemented using functions. A function with no return value can be treated similar to a procedure of Pascal. The main difference between using functions and procedures in C++ (or C) is that function can be placed on the right side of the '=' (assignment) and on either side of '==' (equal) operator. Procedures (function with no return values) cannot be used with these operators. The return value from the function can be directly passed to `cout` for display, whereas procedures cannot be used in the `cout` statement.

Niceties of Parameter Passing

Pass by address/reference is also used when the size of the user defined data-structure is large, since a large number of arguments cannot be accommodated in the limited stack space. Consider the following declaration:

```
struct LargeStruct
{
    char Name[30];
    unsigned int Age, Sex;
    char Address[50];
    enum MartialStatus { Married, Unmarried } Ms;
};
```

If a variable of the above structure type is passed by value, 85 bytes of data movement between the caller space and a function stack space is required. If it is passed by address, it just requires 4 bytes movement and thus reduces the function context switching overhead.

7.7 Return by Reference

A function that returns a reference variable is actually an alias for the referred variable. This method of returning references is used in operator overloading to form a cascade of member function calls specified in a single statement. For example,

```
cout << i << j << endl;
```

is a set of cascaded calls that returns a reference to the object `cout`. The program `ref.cpp` illustrates the function return value by reference.

```
// ref.cpp: return variable by reference
#include <iostream.h>
int & max( int & x, int & y );      // prototype
void main()
{
    int a, b, c;
    cout << "Enter two integers <a, b>: ";
    cin >> a >> b;
    max( a, b ) = 425;
    cout<<"The value of a and b on execution of max(a,b) = 425; ..." << endl;
    cout << "a = " << a << " b = " << b;

}
int & max( int & x, int & y )      // function definition
{
    // all the statements enclosed in braces form body of the function
    if( x > y )
        return x;                // function return
    else
        return y;                // function return
}
```

Run1

Enter two integers <a, b>: 1 2

The value of a and b on execution of `max(a, b) = 425; ...`
`a = 1 b = 425`

Run2

Enter two integers <a, b>: 2 1
 The value of a and b on execution of `max(a, b) = 425; ...`
`a = 425 b = 1`

In `main()`, the statement

```
max( a, b ) = 425;
```

invokes the function `max`. It returns the reference to the variable holding the maximum value and assigns the value 425 to it (see *Run2*). Since the return type of the `max()` is `int &`, it implies that the call to `max()` can appear on the left-hand side of an assignment statement. Therefore, the above statement is valid and assigns 425 to a if it is larger, otherwise, it is assigned to b.

7.8 Default Arguments

Normally, a function call should specify all the arguments used in the function definition. In a C++ function call, when one or more arguments are omitted, the function may be defined to take default values for the omitted arguments by providing the *default values* in the function prototype.

Parameters without default arguments are placed first, and those with default values are placed later (because of the C++ convention of storing the arguments on the stack from right to left). Hence the feature of default arguments allows the same function to be called with fewer arguments than defined in the function prototype.

To establish a default value, the function prototype or the function definition (when functions are defined before being called) must be used. The compiler checks the function prototype/declarator with the arguments in the function call to provide default values (if available) to those arguments, which are omitted. The arguments specified in the function call explicitly always override the default values specified in the function prototype/declarator. In a function call, all the trailing missing arguments are replaced by default arguments as shown in Figure 7.11.

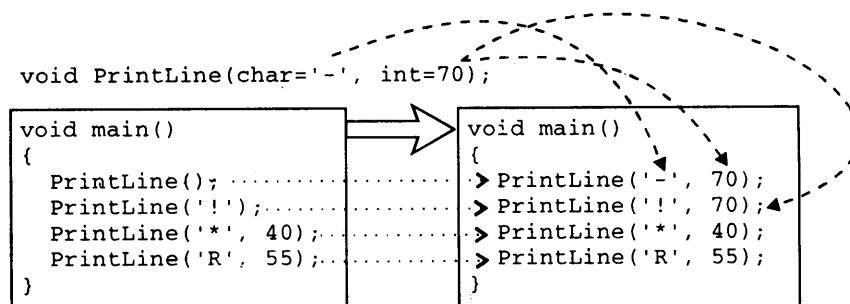


Figure 7.11: Preprocessor handling missing arguments at function call using default arguments

When a function is called by omitting some arguments, they are supplied by the compiler implicitly. The code of the program by no means becomes shorter or more efficient, but it provides high flexibility

212 Mastering C++

1. Therefore, the old code referring to this function need not be modified and new statements can be added without affecting the functionality. The program `defarg2.cpp` extends the capability of `defarg1.cpp` program.

```
// defarg2.cpp: extending the functionality without modifying old calls
#include <iostream.h>
void PrintLine( char = '-', int = 70, int = 1 );
void main()
{
    PrintLine();          // uses both default arguments
    PrintLine( '!' );    // assumes 2nd argument as default
    PrintLine( '*', 40 ); // ignores default arguments
    PrintLine( 'R', 55 ); // ignores default arguments
    // new code, Note: old code listed above is unaffected
    PrintLine( '&', 25, 2 );
}
void PrintLine( char ch, int RepeatCount, int nLines )
{
    int i, j;
    for( j = 0; j < nLines; j++ )
    {
        cout << endl;
        for( i = 0; i < RepeatCount; i++ )
            cout << ch;
    }
}
```

Run

```
-----
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
*****
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
```

The following statements in the above two programs

```
PrintLine();          // uses both default arguments
PrintLine( '!' );    // assumes 2nd argument as default
PrintLine( '*', 40 ); // ignores default arguments
PrintLine( 'R', 55 ); // ignores default arguments
```

are the same. Though the functionality of `PrintLine()` is enhanced in the program `defarg2.cpp`, the old code referring to it remains unaffected in terms of its functionality; the compiler supplies the last argument as 1, and thereby the new function does the same operation as that of the old one. Thus, the C++ feature of default arguments can be potentially utilized in extending a function without modifying the old code.

A default argument can appear either in the function prototype or definition. Once it is defined, it cannot be redefined. It is advisable to define default arguments in the function prototype so that it is known to the compiler at the time of compilation. Variable names may be omitted while assigning default values in the prototype.

7.9 Inline Functions

Function calls involve branching to a specified address, and returning to the instruction following the function call. That is, when the program executes a function call instruction, the CPU stores the memory address of the instruction following the function call, copies the arguments of the function call onto the stack, and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register, and returns control to the calling function. This constitutes an overhead in the execution time of the program. This overhead is relatively large if the time required to execute a function is less than the context switch time.

C++ provides an alternative to normal function calls in the form of **inline functions**. Inline functions are those whose function body is inserted in place of the function call statement during the compilation process. With the inline code, the program will not incur any context switching overhead. The concept of inline functions is similar to *macro functions* of C. Hence, inline functions enjoy both the flexibility and power offered by normal functions and macro functions respectively.

An inline function definition is similar to an ordinary function except that the keyword `inline` precedes the function definition. The syntax for defining an inline function is shown in Figure 7.12..

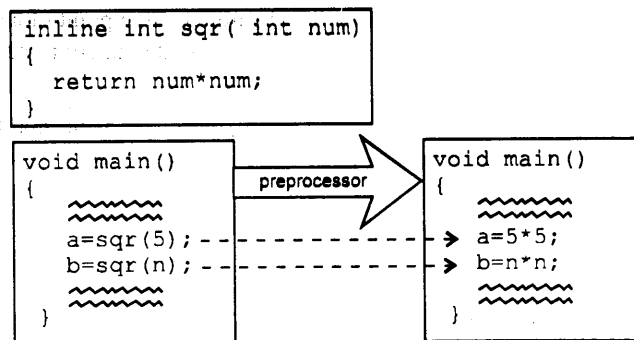


Figure 7.12: Inline function and its expansion

The significant feature of inline functions is: there is no explicit function call and body is substituted at the point of inline function call, thereby, the run-time overhead for function linkage mechanism is reduced. The program `square.cpp` uses inline function to compute the square of a number.

```

// square.cpp: square of a number using inline function
#include <iostream.h>
inline int sqr( int num )
{
    return num*num;
}
void main()
{
    float n;
    cout << "Enter a number: ";
    cin >> n;
    cout << "Its Square = " << sqr( n ) << endl;
}
  
```

214 Mastering C++

```
    cout << "sqr( 10 ) = " << sqr( 10 );  
}
```

Run

```
Enter a number: 5  
Its Square = 25  
sqr( 10 ) = 100
```

In main, the statement

```
    cout << "Its Square = " << square( num );
```

invokes the inline function `square(..)`. It will be suitably replaced by the instruction(s) of the body of the function `square(..)` by the compiler. The execution time of the function `square(..)` is less than the time required to establish a linkage between the caller (calling function) and callee (called function). Execution of a normal function call involves the operation of saving actual parameter and function return address onto the stack followed by a call to the function. On return, the stack must be cleaned to restore the original status. This process is costly when compared to having square computation instructions within a caller's body. Thus, inline functions enjoy the flexibility and modularity of functions and at the same time achieve computational speedup. Functions having small body do not increase the code size, although they are physically substituted at the point of a call; there is no code for function linkage mechanism. Hence, it is advisable to declare the functions having a small function body as inline functions.

The compiler has the option to treat the inline function definition as normal functions (a warning message is displayed). The compiler does not allow large segments of code to be grouped as inline functions. The compiler does not treat functions with loops as inline. Programs with inline functions execute faster than programs containing normal functions (non inline) at the cost of increase in the size of the executable code.

7.10 Function Overloading

Function polymorphism, or function overloading is a concept that allows multiple functions to share the same name with different argument types. Function polymorphism implies that the function definition can have multiple forms. Assigning one or more function body to the same name is known as *function overloading* or *function name overloading*.

The program `swap4.cpp` illustrates the need for function overloading. It has multiple functions for swapping numbers of different data types but with different names.

```
// swap4.cpp: multiple swap functions with different names  
#include <iostream.h>  
void swap_char( char & x, char & y )  
{  
    char t; // temporary used in swapping  
    t = x;  
    x = y;  
    y = t;  
}
```

```

void swap_int( int & x, int & y )
{
    int t;    // temporary used in swapping
    t = x;
    x = y;
    y = t;
}
void swap_float( float & x, float & y )
{
    float t; // temporary used in swapping
    t = x;
    x = y;
    y = t;
}
void main()
{
    char ch1, ch2;
    cout << "Enter two Characters <ch1, ch2>: ";
    cin >> ch1 >> ch2;
    swap_char( ch1, ch2 );
    cout << "On swapping <ch1, ch2>: " << ch1 << " " << ch2 << endl;
    int a, b;
    cout << "Enter two integers <a, b>: ";
    cin >> a >> b;
    swap_int( a, b );
    cout << "On swapping <a, b>: " << a << " " << b << endl;
    float c, d;
    cout << "Enter two floats <c, d>: ";
    cin >> c >> d;
    swap_float( c, d );
    cout << "On swapping <c, d>: " << c << " " << d;
}

```

Run

```

Enter two Characters <ch1, ch2>: R K
On swapping <ch1, ch2>: K R
Enter two integers <a, b>: 5 10
On swapping <a, b>: 10 5
Enter two floats <c, d>: 20.5 99.5
On swapping <c, d>: 99.5 20.5

```

The above program has three different functions:

```

void swap_char( char & x, char & y )
void swap_int( int & x, int & y )
void swap_float( float & x, float & y )

```

performing the same activity, but on different data types. Logically, all the three functions display the value of the input parameters. It has names such as `swap_char`, `swap_int`, `swap_float`, etc., making the task of programming difficult and creating the need to remember function names, which perform the same operation. In C++, this difficulty is circumvented by using the feature of overloading the function.

In C++, two or more functions can be given the same name provided the signature (parameters count or their data types) of each of them is unique either in the number or data type of their arguments. It is possible to define several functions having the same name, but performing different actions. It helps in reducing the need for unusual function names, making the code easier to read. The functions must only differ in the argument list. For example

```
swap( int, int ); // prototype
swap( float, float ); // prototype
```

From user's view point, there is only one operation which performs swapping numbers of different data types.

All the functions performing the same operation must differ in terms of the input argument data-types or number of arguments. The program `swap5.cpp` illustrates the benefits of function overloading.

```
// swap5.cpp: multiple swap functions, function overloading
#include <iostream.h>
void swap( char & x, char & y )
{
    char t; // temporarily used in swapping
    t = x;
    x = y;
    y = t;
}
void swap( int & x, int & y )
{
    int t; // temporarily used in swapping
    t = x;
    x = y;
    y = t;
}
void swap( float & x, float & y )
{
    float t; // temporarily used in swapping
    t = x;
    x = y;
    y = t;
}
void main()
{
    char ch1, ch2;
    cout << "Enter two Characters <ch1, ch2>: ";
    cin >> ch1 >> ch2;
    swap( ch1, ch2 ); // compiler calls swap( char &a, char &b );
    cout << "On swapping <ch1, ch2>: " << ch1 << " " << ch2 << endl;
    int a, b;
    cout << "Enter two integers <a, b>: ";
    cin >> a >> b;
    swap( a, b ); // compiler calls swap( int &a, int &b );
    cout << "On swapping <a, b>: " << a << " " << b << endl;
}
```



```

float c, d;
cout << "Enter two floats <c, d>: ";
cin >> c >> d;
swap( c, d ); // compiler calls swap( float &a, float &b );
cout << "On swapping <c, d>: " << c << " " << d;

```

Run

```

Enter two Characters <ch1, ch2>: R K
On swapping <ch1, ch2>: K R
Enter two integers <a, b>: 5 10
On swapping <a, b>: 10 5
Enter two floats <c, d>: 20.5 99.5
On swapping <c, d>: 99.5 20.5

```

In the above program, three functions named `swap()` are defined, which only differ in their argument data types: `char`, `int`, or `float`. In `main()`, when the statement

```
swap( ch1, ch2 );
```

is encountered, the compiler invokes the `swap()` function which takes character type arguments. This decision is based on the data type of the arguments. (see Figure 7.13).

```

void swap(float &x, float &y);
void swap(int &x, int &y);
void swap(char &x, char &y);

void main()
{
    char ch1, ch2;
    int a, b;
    float x, y;
    swap(ch1, ch2);
    swap(a, b);
    swap(x, y);
}

```

Figure 7.13: Function overloading

It is interesting to note the way in which the C++ compiler implements function overloading. Although the functions share the same name in the source text (as in the example above, `swap`), the compiler (and hence the linker) uses quite different names. The conversion of a name in the source file to an internally used name is called *name mangling*. It can be performed as follows: the C++ compiler might convert the name `void swap(int &, int &)` to the internal name say `vshowI`, while an analogous function with a `char*` argument might be called `vswapCP`. The actual names, which are internally used, depend on the compiler and are transparent to the programmer.

Another typical example program of function overloading is illustrated in `show.cpp`.

```

// show.cpp: display different types of information with same function
#include <iostream.h>

```

```

void show( int val )
{
    cout << "Integer: " << val << endl;
}
void show( double val )
{
    cout << "Double: " << val << endl;
}
void show( char *val )
{
    cout << "String: " << val << endl;
}
int main ()
{
    show( 420 );                // calls show( int val );
    show( 3.1415 );            // calls show( double val );
    show( "Hello World\n!" ); // calls show( char *val );
    return( 0 );
}

```

Run

```

Integer: 420
Double: 3.1415
String: Hello World
!

```

The following remarks can be made on function overloading:

- ◆ The use of more than one function with the same name, but having different actions should be avoided. In the above example, the functions `show()` are somewhat related (they print information on the screen). However, it is also possible to define two functions, say `lookup()`, one of which would find a name in a list, while the other would determine the video mode. In this case, the two functions have nothing in common except their name. It would, therefore, be more practical to use names such as `findname()` and `getvidmode()`, which suggest the action they perform.
- ◆ C++ does not permit overloading of functions differing only in their return value. The reason is that it is always the programmer's choice to inspect or ignore the return value of a function. For instance, the fragment

```
printf("Hello World!\n");
```

holds no information concerning the return value of the function `printf()`. (The return value in this case is an integer, which states the number of printed characters. This return value is practically never inspected). Two functions `printf()`, which would only differ in their return types and hence they are not distinguished by the compiler.

- ◆ Function overloading can lead to surprises. For instance, imagine the usage of statements

```
show( 0 );
show( NULL );
```

where there are multiple overloaded functions as in the program `show.cpp`. The zero could be interpreted here as a `NULL` pointer to a `char`, i.e., a `(char*) 0`, or as an integer with the value zero. C++ will invoke the function expecting an integer argument, which might not be what the user expects.

7.11 Function Templates

C++ allows to create a single function possessing the capabilities of several functions, which differ only in the data types. Such a function is known as *function template* or *generic function*. It permits writing one source declaration that can produce multiple functions differing only in the data types. The syntax of function template is shown in Figure 7.14.

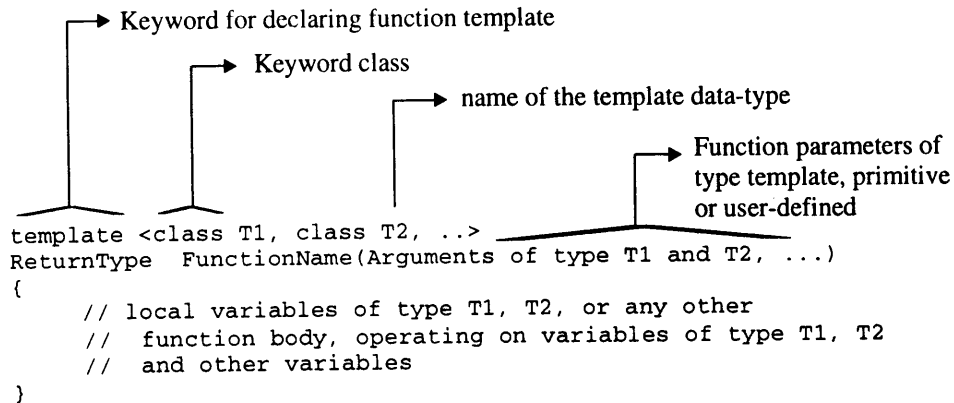


Figure 7.14: Syntax of function template

The program `swap5.cpp` has functions with the same code pattern (same function body but operating on different data types). The program `swap6.cpp` illustrates, declaring a single function template from which all those functions having the same pattern of code, but operating on different data types can be created.

```

// swap6.cpp: multiple swap functions, function overloading
#include <iostream.h>

template <class T>
void swap( T & x, T & y )
{
    T t; // temporarily used in swapping, template variable
    t = x;
    x = y;
    y = t;
}

void main()
{
    char ch1, ch2;
    cout << "Enter two Characters <ch1, ch2>: ";
    cin >> ch1 >> ch2;
    swap( ch1, ch2 ); // compiler creates and calls swap( char &x, char &y );
    cout << "On swapping <ch1, ch2>: " << ch1 << " " << ch2 << endl;
    int a, b;
    cout << "Enter two integers <a, b>: ";
    cin >> a >> b;
}

```

```

swap( a, b ); // compiler creates and calls swap( int &x, int &y );
cout << "On swapping <a, b>: " << a << " " << b << endl;
float c, d;
cout << "Enter two floats <c, d>: ";
cin >> c >> d;
swap( c, d ); // compiler creates and calls swap( float &x, float &y );
cout << "On swapping <c, d>: " << c << " " << d;
}

```

Run

```

Enter two Characters <ch1, ch2>: R K
On swapping <ch1, ch2>: K R
Enter two integers <a, b>: 5 10
On swapping <a, b>: 10 5
Enter two floats <c, d>: 20.5 99.5
On swapping <c, d>: 99.5 20.5

```

In main(), when the compiler encounters the statement

```
swap( ch1, ch2 );
```

calling swap template function with char type variables, it internally creates a function of type

```
swap( char &a, char &b );
```

The compiler automatically identifies the data type of the arguments passed to the template function, creates a new function and makes an appropriate call. The process by which the compiler handles function templates is totally invisible to the user. Similarly, the compiler converts the following calls

```

swap( a, b ); // compiler creates and calls swap( int &x, int &y );
swap( c, d ); // compiler creates and calls swap( float &x, float &y );

```

into equivalent functions and calls them based on their parameter data types.

For more details on function templates, refer to the chapter: *Generic Programming with Templates*.

7.12 Arrays and Functions

The arrays are passed by reference or by address. To pass an array to a function, it is sufficient to pass the address of the first element of the array. The program sort.cpp illustrates the concept of passing array type parameters to a function.

```

// sort.cpp: function to sort elements of an array
#include <iostream.h>
enum boolean { false, true };
void swap( int & x, int & y )
{
    int t; // temporary used in swapping
    t = x;
    x = y;
    y = t;
}
void BubbleSort( int * a, int size )
{
    boolean swapped = true;

```

```

    for( int i = 0; (i < size - 1) && swapped; i++ )
    {
        swapped = false;
        for( int j = 0; j < ( size - 1 ) - i; j++ )
            if( a[ j ] > a[ j + 1 ] )
            {
                swapped = true;
                swap( a[ j ], a[ j + 1 ] );
            }
    }
}

void main( void )
{
    int a[25];
    int i, size;
    cout << "Program to sort elements..." << endl;
    cout << "Enter the size of the integer vector <max=25>: ";
    cin >> size;
    cout << "Enter the elements of the integer vector..." << endl;
    for( i = 0; i < size; i++ )
        cin >> a[i];
    BubbleSort( a, size );
    cout << "Sorted Vector:" << endl;
    for( i = 0; i < size; i++ )
        cout << a[i] << " ";
}

```

Run

```

Program to sort elements...
Enter the size of the integer vector <max=25>: 5
Enter the elements of the integer vector...
8
6
9
3
2
Sorted Vector:
2 3 6 8 9

```

In main(), the statement

```
BubbleSort( a, size );
```

invokes the sorting function by passing the address of the array variable a and the value of the variable size to it. Hence, any modification made to the elements of the array a will be reflected in the caller.

7.13 C++ Stack

The medium for communication between a caller and the callee is the stack, which is used to store function parameters, return address, local variables, etc. When the function is invoked, the information such as return address and parameters, are pushed onto the stack by the function linkage mechanism;

these values are pushed onto or popped from the stack using the C convention for parameter passing. The argument values are pushed in order, from right to left. When they are popped out, the topmost value stored in the stack will be passed to the first parameter in the function parameter list. The order of storing the function parameters in the stack when the statement

```
func( a, b, c, d );
```

is invoked is shown in Figure 7.15. Note that, the Pascal convention of parameter passing is to push parameters from left to right when a function is invoked. Knowledge of parameter passing convention is essential while doing mixed language programming.

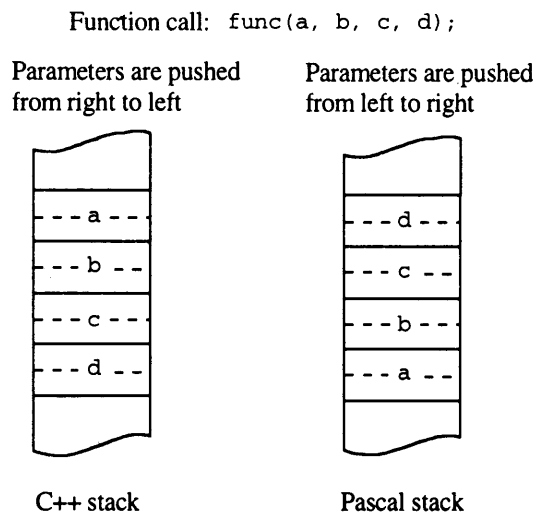


Figure 7.15: Parameter passing and Stack

The program `funcstk.cpp` demonstrates the concept of storing and retrieving the elements from the stack.

```
// funcstk.cpp: C++ convention of using stack
#include <iostream.h>
void Func( j, k )
{
    cout<<"In the function the argument values are " << j<< " .. "<<k<< endl;
}
int main( void )
{
    int i = 99;
    Func( ++i, i );
}
```

Run

In the function the argument values are 100 .. 99

The output of the program is not 100 .. 100 as expected, because of the C convention for passing

parameters. In the function call, first the value of right-most parameter *i*, which is 99 will be pushed onto the stack, and will be followed by `++i`; i.e., 100. Hence, the stack will have 99 at the bottom and 100 at the top. Hence, the statement

```
Func( ++i, i );
```

assigns the value 100 and 99 to the formal parameters *j* and *k* respectively.

7.14 Scope and Extent of Variables

Every variable in a program has some memory associated with it. Memory for variables are allocated and released at different points in the program. For example, in case of normal local variables defined in functions, memory is allocated when the function starts execution and released when the function returns. A variable defined outside all function bodies is called a global variable. Its extent is the entire life-span of the program. *The period of time during which the memory is associated with a variable is called the extent of the variable.* Consider the following function

```
void func()
{
    int i;
    i = 10;
}
```

Allocation of memory to the integer variable *i* is the process of deciding the memory locations to be occupied by *i*. The memory of such local variables is allocated in the program stack when the function `func()` is invoked. Naturally, the memory that was allocated to *i* is released when the function terminates, and that memory space is available for use. Identifiers defined in a function are not accessible outside that function and hence, their extent is limited to life of that function. However, there are exceptions (static variables). For instance, consider the following segment of a program code:

```
void func()
{
    int i;
    i = 10;
}
void main()
{
    i = 20;
    func();
    i = 30;
}
```

When this program is compiled, the statements,

```
i = 20;
i = 30;
```

lead to compilation errors; the variable *i* is not visible inside the `main()`. So the definition of the identifier *i* is valid only inside the `func()`. *The region of source code over which the definition of an identifier is visible is called the scope of the identifier.* The scope of the variable *i* defined in `func()` is limited to this function only. If the statement

```
int i;
```

is defined in the beginning of `main()`, then no errors occur, but nevertheless, the variable *i* in the

`func()` and that in function `main()` are different. Modifications to one variable do not affect the other variables. Note that the scope of the variable defined in `main()` is limited to `main()` only, whereas its extent is entire life-span (execution time) of the program. The program `variable.cpp` illustrates the scope and extent of local and global variables.

```
// variable.cpp: scope and extent of different variable
#include <iostream.h>
int g = 100; // global variable
void func1()
{
    int g = 50; // local variable
    cout << "Local variable g in func1(): " << g << endl;
}
void func2()
{
    cout << "In func2() g is visible, since it is global." << endl;
    cout << "Incrementing g in func..." << endl;
    g++; //accesses global variable
}
void main()
{
    cout << "In main g is visible here, since it is global.\n";
    cout << "Assigning 20 to g in main...\n";
    g = 20; // accesses global variable
    cout << "Calling func1...\n";
    func1();
    cout << "func1 returned. g is " << g << endl;
    cout << "Calling func2...\n";
    func2();
    cout << "func2 returned. g is " << g << endl;
}
```

Run

```
In main g is visible here, since it is global.
Assigning 20 to g in main...
Calling func1...
Local variable g in func1(): 50
func1 returned. g is 20
Calling func2...
In func2() g is visible, since it is global.
Incrementing g in func...
func2 returned. g is 21
```

The global variable `g` is visible to all functions (entire file) and its *extent* is the entire execution time of the program. The scope and extent of local variable `g` of `func1()` is limited to its function body.

The scope of a variable can conform to a block, a function, a file, or an entire program (in case of multimodule file). The variables defined within a block can be accessed only within that block. The program `block1.cpp` illustrates the block scope of variables.


```

// block.cpp: illustration of the variables scope in blocks
#include <iostream.h>
int main( void )
{
    int i = 144;
    cout << "i = " << i;
    {
        /* nested block*/
        int k = 12;
        cin >> k;
        i = i % k ;
    }
    if( i == 0 )
        cout << " i is a divisor of " << k; // Error: k undefined in main()
    return 0;
}

```

Reference to variable *k* in the main block results in a compile-time error: *Undefined symbol k in the function main();* the variable *k* is declared inside the nested block within *main()*. The memory space for the variable *k* is allocated when the execution of the block starts, and released when execution reaches the end of the nested block. When a variable is accessed, the compiler first checks for its existence in the current block, and then moves outwards if it does not exist in the current block; this process continues until the global definition. The function can access the identifiers in the parameter list, the local definitions and the global definitions (if any).

7.15 Storage Classes

The period of time during which memory is associated with a variable is called the extent of the variable. It is characterized by storage classes. The storage class of a variable indicates the allocation of storage space to the variable by the compiler. Storage classes define the extent of a variable. C++ supports the following four types of storage classes:

- ◆ auto
- ◆ register
- ◆ extern
- ◆ static

The syntax for defining variables with explicit storage class is shown in Figure 7.16. The storage classes except *extern* are used for defining variables; *extern* is used for declaration of variables. The scope and extent of *auto* and *register* storage class is the same. The scope of *static* variables is limited to its block (maximum to a file), but its extent is throughout the execution time of the program (does not matter whether it is local or global type).

auto, register,
static, or extern

StorageClass DataType Variable1,....;

Figure 7.16: Storage classes and variable declaration

Declaration Versus Definition

A declaration informs the compiler about the existence of the data or a function some where in the program. A definition allocates the storage location. In C++, a piece of data or function can be declared in several different places, but there must only be one definition. Otherwise, the linker will complain (generates multiple definition error) while uniting all the object modules, if it encounters more than one definition for the same function or piece of data. Almost all C and C++ programs require declarations. Therefore, it is essential for the programmer to understand the correct way to write a declaration. As far as data is concerned, except `extern` storage class, all others define data i.e., they not only direct the compiler, but also allocate resource for a variable.

Auto Variables

By default, all the variables are defined as auto variables. They are created when the function/block is entered and destroyed when the function/block is terminated. The memory space for local auto variables is allocated on the stack. The global auto variables are visible to all the modules of a program, and hence, they cannot be defined many times unlike the declarations.

Register Variables

The allocation of CPU (processor) registers to variables, speeds up the execution of a program; memory is not referred when such variables are accessed. The number of variables, which can be declared as register are limited (typically two or three), within any function or as global variables (else they are treated as auto variables). A program that uses register variables executes faster when compared to a similar program without register variables. It is possible to find out the allocation of register variables only by executing and comparing the timing performance of the program (perceptible in large programs). It is the responsibility of the compiler to allot register variables. In case the compiler is unable to do so, these variables are treated as auto variables. It is advisable to define frequently used variables, such as loop indices, as register variables. It is illustrated in the program `regvar.cpp`.

```
// regvar.cpp: use of register variable as loop index
#include <iostream.h>
#include <string.h>
void main()
{
    char name[30];
    register int i;      // register variable
    cout << "Enter a string: ";
    cin >> name;
    cout << "The reverse of the string is: ";
    for( i = strlen( name )-1; i >= 0; i-- )
        cout << name[i];
}
```

Run1

Enter a string: mahatma
The reverse of the string is: amtaham

Run2

Enter a string: malayalam

The reverse of the string is: malayalam

Static Variables

The static storage class allows to define a variable whose scope is restricted to either a block, a function, or a file (but not all files in multimodule program) and extent is the life-span of a program. The memory space for local static and global variables is allocated from the *global heap*. Static variables that are defined within a function remember their values from the previous call (i.e., the values to which they are initialized or changed before returning from the function). The static variables defined outside all functions in a file are called *file static variables*. They are accessible only in the file in which they are defined. The program `count.cpp` illustrates the use of function static local variables.

```
// count.cpp: use of static variables defined inside functions
#include <iostream.h>
void PrintCount( void )
{
    static int Count = 1; // Count is initialized only on the first call
    cout << "Count = " << Count << endl;
    Count = Count + 1;    // The incremented value of Count is retained
}
void main( void )
{
    PrintCount();
    PrintCount();
    PrintCount();
}
```

Run

```
Count = 1
Count = 2
Count = 3
```

The output of the program is a sequence of numbers starting with 1, rather than a string of 1's. The initialization of static variable `Count` is performed only in the first instance of the function call. In successive calls to the function, the variable `Count` has the same value as it had before the termination of the most recent call. However, these static variables are not accessible from other parts of the program.

Extern global variables are global to the file in which they are defined. They are used when the same global variable is referenced in each one of the files and these variables must be independent of each other across files. The use of global variables is not recommended, since they do not allow to achieve function independence which is one of the basic ideas of modular programming.

Extern Variables

When a program spans across different files, they can share information using global variables. Global variables must be defined only once in any of the program module and they can be accessed by all others. It is achieved by declaring such variables as *extern* variables. It informs the compiler that such variables are defined in some other file. Consider a program having the following files:

```

// file1.cpp: module one defining global variable
int done; // global variable definition
void func1()
{
    ....
}
void disp()
{
    ....
}
// file2.cpp: module two of the project
extern int done; // global variable declaration
void func3
{
    ....
    ....
}

```

In file1.cpp, the statement

```
int done;
```

defines the variable done as a global variable. In file2.cpp, the statement

```
extern int done;
```

declares the variable done and indicates that it is defined in some other file. Note that the definition of the variable done must appear in any one of the modules, whereas extern declaration can appear in any or all modules of a program. When the linker encounters such variables, it binds all references to the same memory location. Thus, any modification to the variable done is visible to all the modules accessing it.

If the global variable done is defined as static, it can be again defined in other modules since the linker treats each as a different variable. Such global static variables have scope restricted to a file and extent is equal to the entire life-span of the program. The auto and static global variables are used mainly in managing large multimodule software project. Note that, the memory space for global variable is allocated from the global heap memory.

7.16 Functions with Variable Number of Arguments

C++ functions such as `vfprintf()` and `vprintf()` accept variable argument lists in addition to taking a number of fixed (known) parameters. The `va_arg`, `va_end`, and `va_start` macros provide access to these argument lists in the standard form. They are used for stepping through a list of arguments when the called function does not know the number and types of the arguments being passed. The header file `stdarg.h` declares one type (`va_list`) and three macros (`va_start`, `va_arg`, and `va_end`).

The syntax of macros handling variable number of arguments are the following:

```

#include <stdarg.h>
void va_start(va_list ap, lastfix);

```

```

type va_arg(va_list ap, type);
void va_end(va_list ap);

```

va_list: This array holds information needed by `va_arg` and `va_end`. When a called function takes a variable argument list, it declares a variable `ap` of type `va_list`.

va_start: This routine (implemented as a macro) sets `ap` to point to the first of the variable arguments being passed to the function. `va_start` must be used before the first call to `va_arg` or `va_end`. The macro `va_start` takes two parameters: `ap` and `lastfix`. `ap` is a pointer to the variable argument list. `lastfix` is the name of the last fixed parameter passed to the caller.

va_arg: This routine (also implemented as a macro) expands to an expression that has the same type and value as the next argument being passed (one of the variable arguments). The variable `ap` to `va_arg` should be the same `ap` that `va_start` initialized. Note that because of default promotions, `char`, `unsigned char`, or `float` types cannot be used with `va_arg`.

When `va_arg` is used first time, it returns the first argument in the list. Every successive use of `va_arg`, returns the next argument in the list. It does this by first dereferencing `ap`, and then incrementing `ap` to point to the following item. `va_arg` uses the type to perform both the dereferencing and to locating the following item. Each time `va_arg` is invoked, it modifies `ap` to point to the next argument in the list.

va_end: This macro helps the called function to perform a normal return. `va_end` might modify `ap` in such a way that it cannot be used unless `va_start` is recalled. `va_end` should be called after `va_arg` has read all the arguments; failure to do so might cause a program to behave erratically.

Return Value: `va_start` and `va_end` return no values; `va_arg` returns the current argument in the list (the one that `ap` is pointing to).

The syntax of function receiving variable number of arguments is:

```

ReturnType Func( arg1, [arguments], ... );

```

It is same as the normal function except for the last three dots, which indicates that the function is of type variable arguments. The program `add.cpp` illustrates the use of variable number of arguments.

```

// add.cpp: variable number of arguments to a function
#include <iostream.h>
#include <stdarg.h>
int add( int argc, ... )
{
    int num, result;
    va_list args;
    va_start( args, argc );    // link to variable arguments
    result = 0;
    for(int i=0; i < argc; i++)
    {
        num = va_arg( args, int );    // get argument value
        result += num;
    }
    va_end( args );    // end of arguments
    return result;
}

```

230 Mastering C++

```
void main()
{
    int sum1, sum2, sum3;
    sum1 = add( 3, 1, 2, 3 );
    cout << "sum1 = " << sum1 << endl;
    sum2 = add( 1, 10 );
    cout << "sum2 = " << sum2 << endl;
    sum3 = add( 0 );
    cout << "sum3 = " << sum3 << endl;
}
```

Run

```
sum1 = 6
sum2 = 10
sum3 = 0
```

The function declarator (prototype)

```
int add( int argc, ... )
```

indicates that it takes one known argument and the remaining are unknown number of arguments. The three dots indicate that the function takes variable arguments, to which a chain has to be built. In add() function, the statement

```
va_list args;
```

creates a pointer variable named args. The macro call statement

```
va_start( args, argc ); // link to variable arguments
```

links variable arguments to the variable args. The variable args is the last known argument and those that follow are variable arguments. The statement

```
num = va_arg( args, int ); // get argument value
```

accesses the argument of type integer and assigns to the variable num. Later, args is updated to point to the next argument. The statement

```
va_end( args ); // end of arguments
```

indicates the end of access to variable arguments using args. In main(), the statement

```
sum1 = add( 3, 1, 2, 3 );
```

invokes the function add() and the first argument is a known argument indicating the number of variable arguments.

The last argument in the list of variable number of arguments must be established by the user. Another way of indicating the end of variable arguments is illustrated in the program sum.cpp.

```
// sum.cpp: variable arguments example
#include <iostream.h>
#include <stdarg.h>
// calculate sum of a 0 terminated list
void sum(char *msg, ...)
{
    int total = 0;
    va_list ap;
    int arg;
```

```

    va_start(ap, msg);
    while ((arg = va_arg(ap, int)) != 0) {
        total += arg;
    }
    cout << msg << total;
    va_end(ap);
}

int main(void)
{
    sum("The total of 1+2+3+4 is ", 1,2,3,4,0);
    return 0;
}

```

Run

The total of 1+2+3+4 is 10

In `main()`, the statement

```
sum("The total of 1+2+3+4 is ", 1,2,3,4,0);
```

invokes the variable argument function. The function `sum()` is designed such that when a zero valued argument is encountered, it is understood that no more arguments exists for further processing. Hence, the last argument 0 (zero) in this case, is the end-of-argument indicator. The programmer has full freedom for selecting suitable end-of-argument indicator.

7.17 Recursive Functions

Many of the scientific operations are expressed using recurrence relations. C++ allows the programmers to express such a relation using functions. A function that contains a function call to itself, or a function call to a second function which eventually calls the first function is known as a recursive function. The recursive definition for computing the factorial of a number can be expressed as follows:

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{fact}(n-1), & \text{otherwise} \end{cases}$$

Recursion, as the name suggests, revolves around a function recalling itself. Recursive functions are those, in which there is atleast one function call to itself (there can be more than one call to itself as in the *tower of hanoi* algorithm). The recursive approach of problem solving substitutes the given problem with another problem of the same form in such a way that the new problem is simpler than the original.

Two important conditions which must be satisfied by any recursive function are:

1. Each time a function calls itself it must be nearer, in some sense, to a solution.
2. There must be a decision criterion for stopping the process or computation.

Recursive functions involve the overhead of saving the return address, formal parameters, local variables upon entry, and restore these parameters and variables on completion.

Factorial of a Number

The program `rfact.cpp` computes the factorial of a number. It has a recursive function `fact()` which implements the above stated definition of recursion.

```
// rfact.cpp: factorial of a number using recursion
#include <iostream.h>
void main( void )
{
    int n;
    long int fact( int ); // prototype
    cout << "Enter the number whose factorial is to be found: ";
    cin >> n;
    cout << "The factorial of " << n << " is " << fact(n) << endl;
}
long fact( int num )
{
    if( num == 0 )
        return 1;
    else
        return num * fact( num - 1 );
}
```

Run

```
Enter the number whose factorial is to be found: 5
The factorial of 5 is 120
```

Tower of Hanoi

Tower of hanoi is a historical problem, which can be easily expressed using recursion. There are N disks of decreasing size stacked on one needle, and two other empty needles. It is required to stack all the disks onto a second needle in the decreasing order of size. The third needle can be used as a temporary storage. The movement of the disks must conform to the following rules:

1. Only one disk may be moved at a time
2. A disk can be moved from any needle to any other
3. At no time, a larger disk rests upon a smaller one.

The program `hanoi.cpp` implements the tower of hanoi problem. The physical model of a tower of hanoi problem is shown in Figure 7.17.

```
// hanoi.cpp: Tower of hanoi simulation using recursion
#include <iostream.h>
void main( void )
{
    unsigned int nvalue;
    char source = 'L', intermediate = 'C', destination = 'R';
    void hanoi( unsigned int, char, char, char );
    cout << "Enter number of disks: ";
    cin >> nvalue;
    cout << "Tower of Hanoi problem with " << nvalue << " disks" << endl;
    hanoi( nvalue, source, intermediate, destination );
}
void hanoi( unsigned n, char left, char mid, char right )
{
    if( n != 0 )
    {
```



```

// Move n-1 disks from starting needle to intermediate needle
hanoi( n-1, left, right, mid );
// Move disk n from start to destination
cout<< "Move disk " << n << " from " << left<<" to " << right <<endl;
// Move n-1 disks from intermediate needle to destination needle
hanoi( n-1, mid, left, right );
}
}

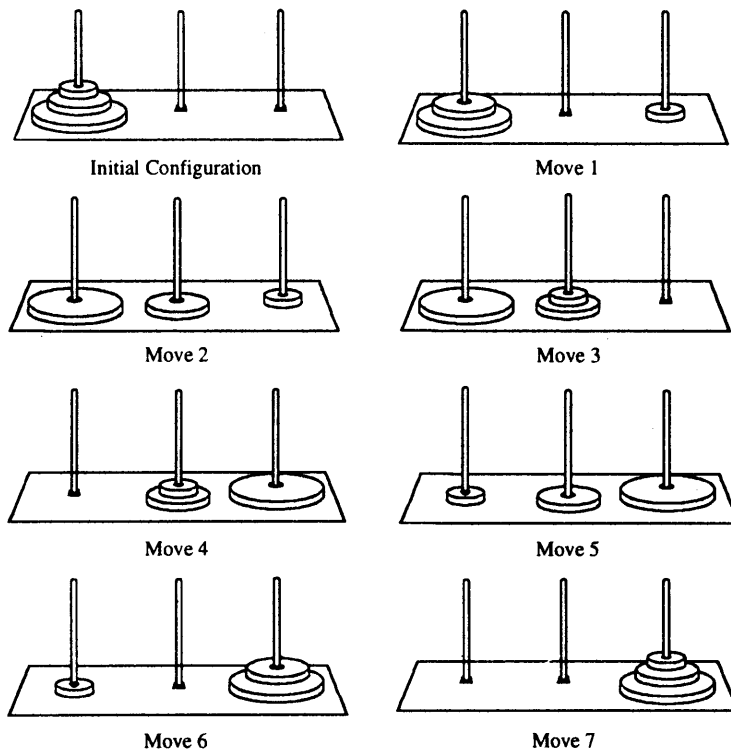
```

Run

```

Enter number of disks: 3
Tower of Hanoi problem with 3 disks.
Move disk 1 from L to R
Move disk 2 from L to C
Move disk 1 from R to C
Move disk 3 from L to R
Move disk 1 from C to L
Move disk 2 from C to R
Move disk 1 from L to R

```

**Figure 7.17: Tower of Hanoi**

7.18 Complete Syntax of `main()`

The function `main()` takes three input parameters called command-line arguments. These are passed from the point of program execution (usually operating system shell or command interpreter). The general format of the `main()` function is shown in Figure 7.18

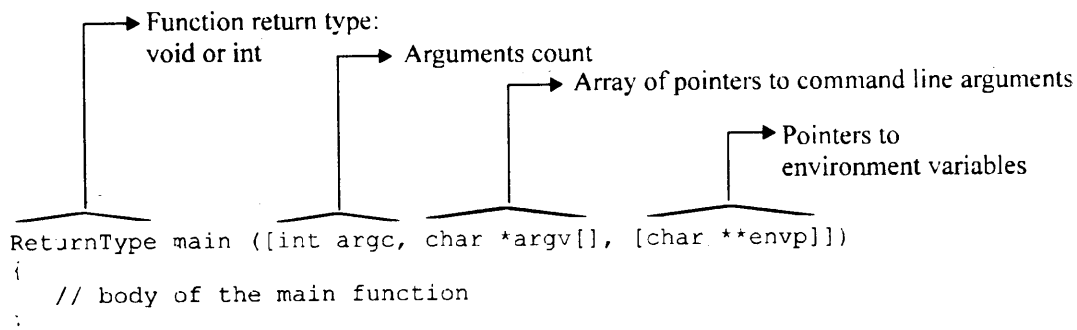


Figure 7.18: Syntax of the main function

The return type of the `main` function must be either `int` or `void`. It is normally used to indicate the status of the program termination. The command-line arguments have the following meaning:

argc: argument count, holds the value of the number of arguments passed to the `main()` function and its value is always positive.

argv: argument vector, holds pointers to the arguments passed from the command line. The meaning of various elements of the `argv` vector is as follows:

```

argv[ 0 ] = pointer to the name of the executable program file (command)
argv[ 1 ] .. argv[ argc - 1 ] = pointers to argument strings

```

envp: environment parameter, holds pointers to environment variables set in the operating system during the program execution. It includes path and environment parameters. It is optional and not a ANSI specification.

When the command `disp hello` is issued at the system prompt, the arguments are set as follows:

```

argc = 2
argv[ 0 ] = "disp"
argv[ 1 ] = "hello"

```

The program `args.cpp` prints the list of arguments passed to it. To execute this program, issue the command `args Hello World` at the system prompt.

```

// args.cpp: printing command line arguments
#include <iostream.h>
void main( int argc, char *argv[] )
{
    int i;
    cout << "Argument Count = " << argc;
    cout << "\nProgram Name = " << argv[ 0 ];
    cout << "\nArgument Vectors Are:\n";
    for ( i = 0; i < argc; i++ )
        cout << argv[i] << "\n";
}

```

Run

```
Argument Count = 3
Program Name = D:\CPP_SRC\MC2CPP.C02\ARGS.EXE
Argument Vectors Are:
D:\CPP_SRC\MC2CPP.C02\ARGS.EXE
Hello
World
```

Program Execution Status

Normally, after the complete execution of the program, it exits from the `main()` function itself. However, programs can be terminated from anywhere within the program. The return type of the main function can be used by the system to decide whether the program terminates with successful execution or not. The return statement in `main()`

```
return 0; // program return type
```

or the `exit()` statement anywhere in the program

```
exit( 0 )
```

terminates the program with the program execution status as zero. The general convention is that, the return value 0 is treated as a successful execution of the program and nonzero value is interpreted as unsuccessful execution of the program. The method of identifying this return value from outside the program (from where it is invoked), depends on the operating system environment in which the program is executed. For instance, under MS-DOS operating system, the system sets the environment variable `errorlevel` to the value returned by the programmer. The user can inspect the value held by the `errorlevel` variable to decide the status of program execution. The program `fullmain.cpp` displays the command line arguments and environment variables.

```
// fullmain.cpp: prints command line arguments and environment variables
#include <iostream.h>
int main( int argc, char **argv, char **envp )
{
    cout << "The number of command line arguments is: " << argc << endl;
    cout << "The command line arguments are as follows" << endl;
    for( int i = 0; i < argc; i++ )
        cout << "argv[" << i << " ] : " << argv[i] << endl;
    cout << "The environment variables are:" << endl;
    i = 0;
    while( *envp[i] )
        cout << envp[i++] << endl;
    return 0;
}
```

Run

```
The number of command line arguments is: 3
The command line arguments are as follows
argv[0] : C:\CPP_SRC\FUNCTION.C07\FULLMAIN.EXE
argv[1] : Hello
argv[2] : World
The environment variables are:
COMSPEC=C:\COMMAND.COM
PROMPT=$p$g
PATH=C:\BC4\BIN;C:\EXCEEDW\PATHWAY;C:\BC4\BIN;C:\WINDOWS;C:\DOS;C:\PATHWAY;
```

Review Questions

- 7.1 What is modular programming and what are its benefits? Explain the same with a C++ example.
- 7.2 Explain different components of a C++ program with a suitable example program.
- 7.3 What are the differences between actual parameters and formal parameters?
- 7.4 What are caller and callee? List the various components causing the overhead of function invocation.
- 7.5 What are library functions? Explain how they ease program development. What are the different categories of functions supported by C++ library?
- 7.6 What is parameter passing? Explain parameter passing schemes supported by C++.
- 7.7 Develop a function to sort numbers using bubble sort technique. Write a driver function also.
- 7.8 What are the differences between parameter passing by value and passing by address?
- 7.9 What are the benefits of pass by reference method of parameter passing over pass by pointer?
- 7.10 What are default arguments? Write a program to compute tax. A *tax compute* function takes two arguments: amount and tax percentage. Default tax percentage is 15% of income.

- 7.11 State whether the following statements are valid or not? Give reasons.

```
tax_amount( int amount, int percentage = 15 ); // prototype
tax_amount( , 5 );
show( char ch = 'A', int count = 3 ); // prototype
show( , 2 );
show( , );
show();
```

- 7.11 What are inline functions? Write an inline function for finding minimum of two numbers.
- 7.12 What is function overloading? Write overloaded functions for computing area of a triangle, a circle, and a rectangle. Develop a driver function.
- 7.13 What are function templates? Write a template based program for sorting numbers.
- 7.14 What is the difference between parameter passing in C++ and Pascal? What is the result of:
`sum = add(i++, a[i]); // if i=1 and a[] = { 5, 10, 15, 20 }`
- 7.15 Define terms: scope and extent. Explain different storage classes supported by C++. Also explain there scope and extent.
- 7.16 Write a program having a variable argument function to multiply input numbers.
- 7.17 What are recursive functions? Write a program to find the gcd of two numbers using the following Euclid's recursive algorithm.

$$\text{gcd}(m, n) = \begin{cases} \text{gcd}(n, m) & \text{if } n > m \\ m & \text{if } n = 0 \\ \text{gcd}(n, m\%n), & \text{otherwise} \end{cases}$$

- 7.18 Write a program for adding integer parameters passed as command line arguments.
- 7.19 Write a program to generate fibonacci series using the following recursive algorithm:

$$\text{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2), & \text{otherwise} \end{cases}$$

- 7.20 Implement a recursive binary search using *divide and conquer* technique.

8

Structures and Unions

8.1 Introduction

Structures combine logically related data items into a single unit. The data items enclosed within a structure are known as members and they can be of the same or different data types. Hence, a structure can be viewed as a heterogeneous user-defined data type. It can be used to create variables, which can be manipulated in the same way as variables of standard data types. It encourages better organization and management of data in a program.

8.2 Structure Declaration

The declaration of a structure specifies the grouping of various data items into a single unit without assigning any resources to them. The syntax for declaring a structure in C++ is shown in Figure 8.1.

```
struct StructureName
{
    DataType member1;
    DataType member1;
    ....
    DataType memberN;
};
```

The diagram shows the C++ structure declaration syntax. A callout box labeled 'keyword' points to the word 'struct'. Another callout box labeled 'structure name' points to 'StructureName'. A large right-facing curly brace groups the member declarations (DataType member1, ..., DataType memberN) and is labeled 'structure members'.

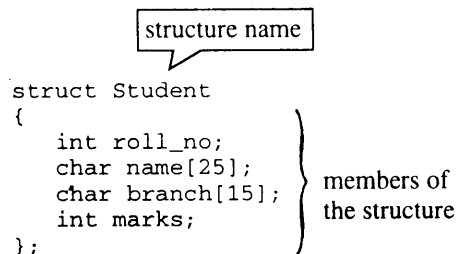
Figure 8.1: Structure declaration

The structure declaration starts with the structure header, which consists of the keyword `struct` followed by a tag. The tag serves as a structure name, which can be used for creating structure variables. The individual members of the structure are enclosed between the curly braces and they can be of the same or different data types. The data type of each variable is specified in the individual member declarations. Like all data structure declarations, the closing brace is terminated with a semicolon.

Consider a student database consisting of student roll number, name, branch, and total marks scored. A structure declaration to hold this information is shown below:

```
struct Student
{
    int roll_no;
    char name[25];
    char branch[15];
    int marks;
};
```

The data items enclosed between flower brackets in the above structure declaration are called *structure elements* or *structure members*. Student is the name of the structure and is called *structure tag*. Note that, some members of Student structure are integer type and some are character array type. The description of various components of the structure Student is shown in Figure 8.2.



```

struct Student
{
    int roll_no;
    char name[25];
    char branch[15];
    int marks;
};

```

Figure 8.2: Declaration of structure Student

The individual members of a structure can be variables of built-in data types, pointers, arrays, or even other structures. All member names within a particular structure must be different. However, member names may be the same as those of variables declared outside the structure. The individual members cannot be initialized inside the structure declaration. For example, the following declaration is invalid:

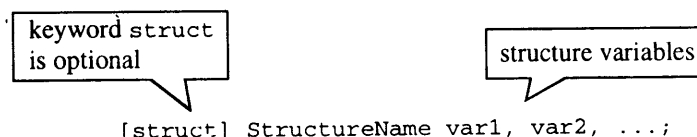
```

struct Student
{
    int roll_no = 0; // Error: initialization not allowed here
    char name[25];
    char branch[15];
    int marks;
};

```

8.3 Structure Definition

The declaration of a structure will not serve any purpose without its definition. It only acts as a blueprint for the creation of variables of type `struct` (structure). The structure definition creates structure variables and allocates storage space for them. Structure variables can be created at the point of structure declaration itself, or by using the structure tag explicitly as and when required. The most commonly used syntax for structure definition is shown in Figure 8.3.



```

[struct] StructureName var1, var2, ...;

```

Figure 8.3: Syntax of structure definition

The use of the keyword `struct` in the structure definition statement is optional. The following statements create variables of the structure Student declared earlier:

```

struct Student s1;
or
Student s1;

```

Figure 8.4 shows the storage organisation of the members of the structure Student.

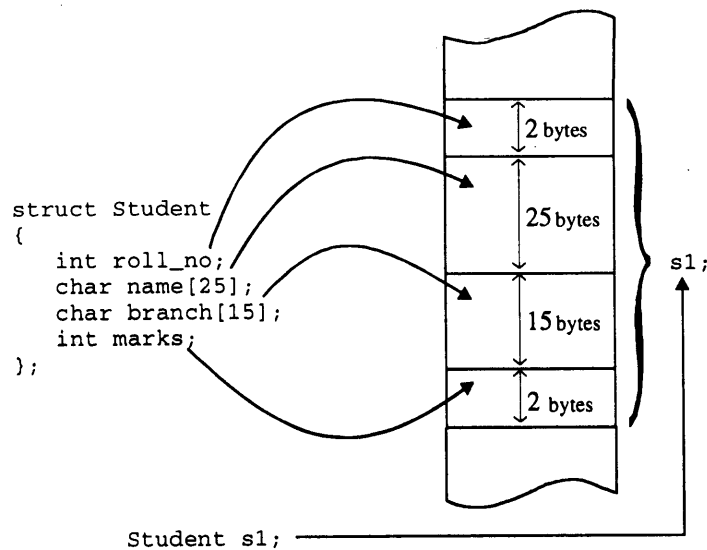


Figure 8.4: Storage organisation when structure variable is defined

The structure variables can be created during the declaration of a structure as follows:

```

struct Student
{
    int roll_no;
    char name[25];
    char branch[15];
    int marks;
} s1;

```

In the above declaration, Student is the structure tag, while s1 is a variable of type Student. If variables of this structure type are not defined later in the program, then the tag name Student can be omitted as shown below:

```

struct
{
    int roll_no;
    char name[25];
    char branch[15];
    int marks;
} s1;

```

It is not a good practice, to have both declaration and definition in the same statement.

Multiple variables of a structure can be created using a single statement as follows:

```
struct Student s1, s3, s4;
```

or

```
Student s1, s3, s4;
```

All these instances are allocated separate memory locations and hence, each one of them are independent variables of the same structure type as shown in Figure 8.5.

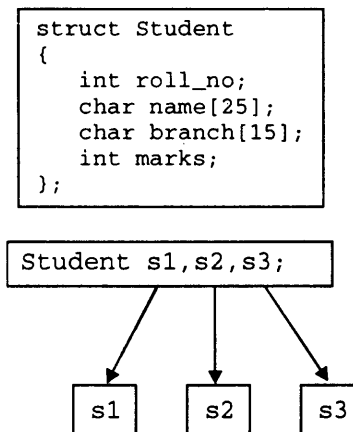


Figure 8.5: Variables of type Student

8.4 Accessing Structure Members

C++ provides the period or dot (.) operator to access the members of a structure independently. The dot operator connects a structure variable and its member. The syntax for accessing members of a structure variable is shown in Figure 8.6.

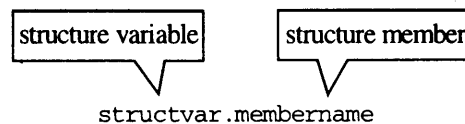


Figure 8.6: Accessing a structure member using dot operator

Here, `structvar` is a structure variable and `membername` is one of its members. Thus, the dot operator must have a structure variable on its left and a legal member name on its right. Consider the following statement:

```
Student s1;
```

Each member of the structure variable `s1` can be accessed using the dot operator as follows:

```
s1.roll_no    will access s1's roll_no
s1.name       will access s1's name
s1.branch     will access s1's branch
s1.marks      will access s1's marks
```


The following are valid operations on the structure variable s1:

```
s1.roll_no = 5;
cin >> s1.roll_no;
strcpy( s1.name, "Mangala" );
cout << s1.name;
strcpy( s1.branch, "Computer" );
```

Accessing members of a structure using structure tag is not allowed. Hence, a statement such as

```
Student.roll_no = 5; // Error: Student is not a structure variable
```

is invalid; structure name Student is a data type like int, and not a variable. Just as int = 10 is invalid, Student.roll_no = 5 is invalid.

The program student1.cpp illustrates the various concepts discussed in the earlier sections such as structure declaration, definition, and accessing members of a structure.

```
// student1.cpp: processing of student data using structures
#include <iostream.h>
// structure declaration
struct Student
{
    int roll_no;
    char name[25];
    char branch[15];
    int marks;
};
void main()
{
    Student s1; // structure definition
    cout << "Enter data for student..." << endl;
    cout << "Roll Number ? ";
    cin >> s1.roll_no; // accessing structure member
    cout << "Name ? ";
    cin >> s1.name;
    cout << "Branch ? ";
    cin >> s1.branch;
    cout << "Total Marks <max-325> ? ";
    cin >> s1.marks;
    cout << "Student Report" << endl;
    cout << "-----" << endl;
    // process student data
    cout << "Roll Number: " << s1.roll_no << endl;
    cout << "Name: " << s1.name << endl;
    cout << "Branch: " << s1.branch << endl;
    cout << "Percentage: " << s1.marks*(100.0/325) << endl;
}
```

Run

```
Enter data for student...
Roll Number ? 5
Name ? Mangala
```

242 Mastering C++

```
Branch ? Computer
Total Marks <max-325> ? 290
Student Report
-----
Roll Number: 5
Name: Mangala
Branch: Computer
Percentage: 89.230769
```

Precedence of the DOT operator

The dot operator is a member of the highest precedence group, and its associativity is from left to right. Hence, the expression such as `++stvar.membern` is equivalent to `++(stvar.membern)`, implying that the unary operator will act only on a particular member of the structure and not the entire structure.

8.5 Structure Initialization

Similar to the standard data types, structure variables can be initialized at the point of their definition. Consider the following structure declaration:

```
struct Student
{
    int roll_no;
    char name[25];
    char branch[15];
    int marks;
};
```

A variable of the structure `Student` can be initialized during its definition as follows:

```
Student s1 = { 5, "Mangala", "Computer", 290 };
```

The initial values for the components of the structure are placed in curly braces and separated by commas. The members of the variable `s1`, `roll_no`, `name`, `branch`, and `marks` are initialized to 5, "Mangala", "Computer", and 290 respectively (see Figure 8.7).

The program `days.cpp` illustrates the initialization of the members of a structure at the point of a structure variable definition.

```
// days.cpp: structure members initialization at the point of definition
#include <iostream.h>
// structure declaration
struct date
{
    int day;
    int month;
    int year;
};
void main()
{
    date d1 = { 14, 4, 1971 };
    date d2 = { 3, 7, 1996 };
```

```

cout << "Birth date: ";
cout << d1.day << "-" << d1.month << "-" << d1.year;
cout << endl << "Today date: ";
cout << d2.day << "-" << d2.month << "-" << d2.year;
}

```

Run

```

Birth date: 14-4-1971
Today date: 3-7-1996

```

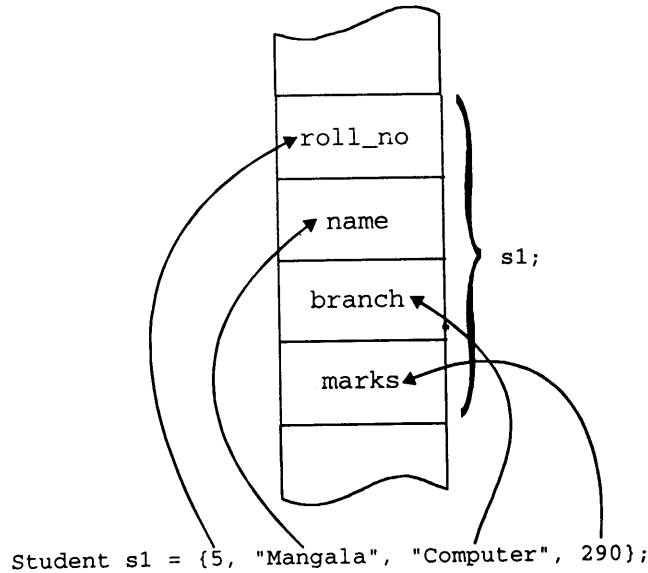


Figure 8.7: Structure members' initialization during definition

8.6 Nesting of Structures

A member of a structure may itself be a structure. Such nesting enables building of very powerful data structures. The `Student` structure can be enhanced to accommodate the date of birth of a student. The new member `birthday` is a structure of type `date` by itself as shown below:

```

struct date
{
    int day;
    int month;
    int year;
};
struct Student
{
    int roll_no;
    char name[25];
    struct date birthday;
    char branch[15];
    int marks;
};

```

The structure to be embedded must be declared before its use. Another way of declaring a nested structure is to embed member structure declaration within the declaration of a new structure as follows:

```
struct Student
{
    int roll_no;
    char name[25];
    struct date
    {
        int day;
        int month;
        int year;
    } birthday;
    char branch[15];
    int marks;
};
```

The embedded structure `date` is declared within the enclosing structure declaration. A variable of type `Student` can be defined as follows:

```
Student s1;
```

The year in which the student `s1` was born can be accessed as follows:

```
s1.birthday.year
```

The following are the some of the valid operations on the variable `s1`:

```
s1.roll_no = 5;
cin >> s1.roll_no;
s1.birthday.day = 2;
s1.birthday.month = 2;
s1.birthday.year = 1972;
```

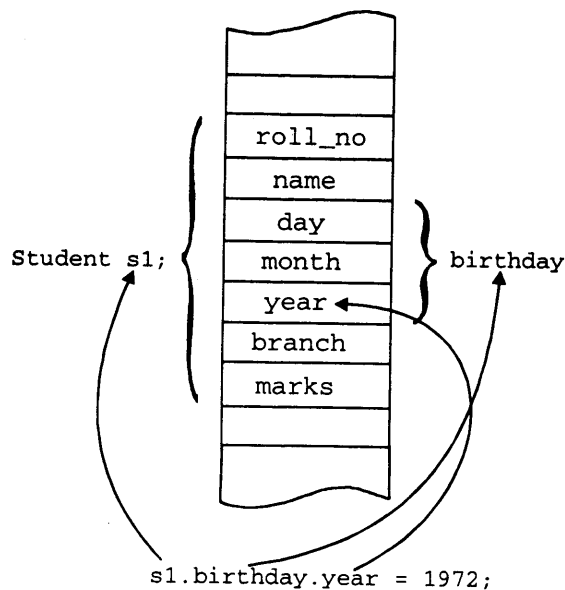


Figure 8.8: Accessing members of nested structures

The dot operator accessing a member of the nested structure `birthday` using the statement

```
s1.birthday.year = 1972;
```

is shown in Figure 8.8. The program `student2.cpp` illustrates the declaration, definition, and processing of nested structure members.

A statement such as

```
s1.date.day = 2; // error
```

is invalid, because a member of the nested structure must be accessed using its variable name.

```
// student2.cpp: processing of student data using structures
#include <iostream.h>
// structure declaration
struct date
{
    int day;
    int month;
    int year;
};
struct Student
{
    int roll_no;
    char name[25];
    struct date birthday; // structure within a structure
    char branch[15];
    int marks;
};
void main()
{
    Student s1; // structure definition
    cout << "Enter data for student..." << endl;
    cout << "Roll Number ? ";
    cin >> s1.roll_no; // accessing structure member
    cout << "Name ? ";
    cin >> s1.name;
    cout << "Enter date of birth <day month year>: ";
    cin >> s1.birthday.day >> s1.birthday.month >> s1.birthday.year;
    cout << "Branch ? ";
    cin >> s1.branch;
    cout << "Total Marks <max-325> ? ";
    cin >> s1.marks;
    cout << "Student Report" << endl;
    cout << "-----" << endl;
    // process student data
    cout << "Roll Number: " << s1.roll_no << endl;
    cout << "Name: " << s1.name << endl;
    cout << "Birth day: ";
    cout<<s1.birthday.day << "-" << s1.birthday.month << "-" << s1.birthday.year;
    cout << endl << "Branch: " << s1.branch << endl;
    cout << "Percentage: " << s1.marks*(100.0/325) << endl;
```

Run

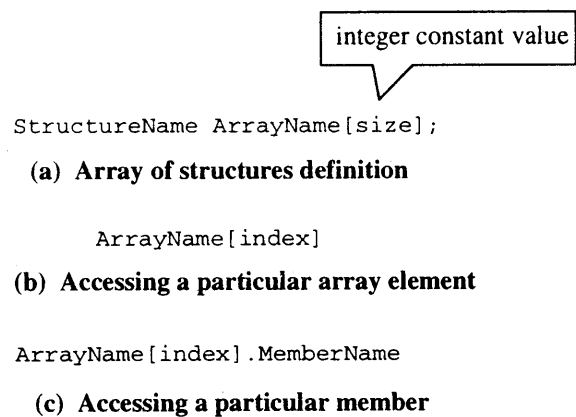
```

Enter data for student...
Roll Number ? 9
Name ? Savithri
Enter date of birth <day month year>: 2 2 1972
Branch ? Electrical
Total Marks <max-325> ? 295
Student Report
-----
Roll Number: 9
Name: Savithri
Birth day: 2-2-1972
Branch: Electrical
Percentage: 91.076923

```

8.7 Array of Structures

It is possible to define an array of structures; each array element is similar to a variable of that structure. The syntax for defining an array of structures and accessing its members using an index, is shown in Figure 8.9.

**Figure 8.9: Array of structures and member access**

The following examples illustrate the concepts of defining arrays of structures and manipulating their members. Consider the structure declaration given below:

```

struct Student
{
    int roll_no;
    char name[25];
    struct date birthday;
    char branch[15];
    int marks;
};

```

An array of the above structure can be defined as follows:

```
Student s[10];
```

The variable `s` is a 10 element array of structures of the type `Student`. The 5th structure can be accessed as follows:

```
s[4]; // arrays are numbered from 0 to n-1
```

The following statements access members of the structure array elements:

```
s[4].name; // access the name of 5th structure
s[0].marks[5]; // access 6th character of 1st structure
&s[2].name // address of 3rd s structure member name
```

Another method of defining an array of structures is as follows:

```
struct Student
{
    int roll_no;
    char name[25];
    struct date birthday;
    char branch[15];
    int marks;
} s[10];
```

More than one array of structure variables can be defined in a single statement as follows:

```
Student class1[10], class2[15];
```

It defines two arrays of structure variables `class1` and `class2` of size 10 and 15 respectively. Each element of the `class1` will be a structure of type `Student`. The program `student3.cpp` illustrates the method of processing of an array of structures.

```
// student3.cpp: processing of student data using structures
#include <iostream.h>
struct Student
{
    int roll_no;
    char name[25];
    char branch[15];
    int marks;
};
void main()
{
    // data definitions of 10 students
    Student s[10];
    int n;
    cout << "How many students to be processed <max-10>: ";
    cin >> n;
    // read student data
    for( int i = 0; i < n; i++ )
    {
        cout << "Enter data for student " << i+1 << "... " << endl;
        cout << "Roll Number ? ";
        cin >> s[i].roll_no;
        cout << "Name ? ";
```

248 Mastering C++

```
    cin >> s[i].name;
    cout << "Branch ? ";
    cin >> s[i].branch;
    cout << "Total Marks <max-325> ? ";
    cin >> s[i].marks;
}
cout << "Students Report" << endl;
cout << "-----" << endl;
// process student data
for( i = 0; i < n; i++ )
{
    cout << "Roll Number: " << s[i].roll_no << endl;
    cout << "Name: " << s[i].name << endl;
    cout << "Branch: " << s[i].branch << endl;
    cout << "Percentage: " << s[i].marks*(100.0/325) << endl;
}
}
```

Run

```
How many students to be processed <max-10>: 2
Enter data for student 1...
Roll Number ? 5
Name ? Mangala
Branch ? Computer
Total Marks <max-325> ? 290
Enter data for student 2...
Roll Number ? 9
Name ? Shivakumar
Branch ? Electronics
Total Marks <max-325> ? 250
Students Report
-----
Roll Number: 5
Name: Mangala
Branch: Computer
Percentage: 89.230769
Roll Number: 9
Name: Shivakumar
Branch: Electronics
Percentage: 76.923077
```

Initialization of Array of Structures

An array of structures can be initialized in the same way as a single structure and hence, the discussion regarding the initialization of a single structure is still relevant. This is illustrated by the following example:

```
Student s[5] = {
    2, "Tejaswi", "CS", 200,
    3, "Laxmi H", "IT", 215,
    5, "Bhavani", "Electronics", 250,
    7, "Anil", "Civil", 215,
```



```
9, "Savithri", "Electrical", 290
```

The variable `s` is an array of 5 elements of type `Student`. Thus, structure element `s[0]` will be assigned the first set of values, `s[1]` the second set of values, etc. Note that there are 5 sets of values in the initialization, which are placed in different rows for clarity. The values are separated by commas and enclosed within braces, with the closing brace being followed by a semicolon. To improve the readability of the program code, it is advisable to enclose the individual **sets of values within braces** as shown below:

```
Student s[5] = {
    { 2, "Tejaswi", "CS", 200 },
    { 3, "Laxmi", "IT", 215 },
    { 5, "Bhavani", "Electronics", 250 },
    { 7, "Anil", "Civil", 215 },
    { 9, "Savithri", "Electrical", 290 }
};
```

The program `student4.cpp` illustrates the initialization of an array of structures at the point of its definition.

```
// student4.cpp: array of structures and their initialization
#include <iostream.h>
struct Student
{
    int roll_no;
    char name[25];
    char branch[15];
    int marks;
};
int const STUDENTS_COUNT = 5;
void main()
{
    // data definitions of 10 students
    Student s[ STUDENTS_COUNT ] = {
        { 2, "Tejaswi", "CS", 285 },
        { 3, "Laxmi", "IT", 215 },
        { 5, "Bhavani", "Electronics", 250 },
        { 7, "Anil", "Civil", 215 },
        { 9, "Savithri", "Electrical", 290 }
    };
    cout << "Students Report" << endl;
    cout << "-----" << endl;
    // process student data
    for( int i = 0; i < STUDENTS_COUNT; i++ )
    {
        cout << "Roll Number: " << s[i].roll_no << endl;
        cout << "Name: " << s[i].name << endl;
        cout << "Branch: " << s[i].branch << endl;
        cout << "Percentage: " << s[i].marks*(100.0/325) << endl;
    }
}
```

Run

```

Students Report
-----
Roll Number: 2
Name: Tejaswi
Branch: CS
Percentage: 87.6923
Roll Number: 3
Name: Laxmi
Branch: IT
Percentage: 66.1538
Roll Number: 5
Name: Bhavani
Branch: Electronics
Percentage: 76.9231
Roll Number: 7
Name: Anil
Branch: Civil
Percentage: 66.1538
Roll Number: 9
Name: Savithri
Branch: Electrical
Percentage: 89.2308

```

Operations Involving the Assignment Operator

The individual structure member can be used in an assignment statement just like any other ordinary variable. It is illustrated in the following statements:

```

s[1].marks = 290;      // marks set to 290
s[1].marks += 5;      // marks is incremented by 5

```

Notice that only the individual structure members are accessed, and not the entire structure. If the structure member is itself a structure, then the embedded structure's member is accessed as follows:

```
s[1].birthday.day
```

It accesses the member `day` of the structure variable `birthday` embedded in the 2nd element of the array of structure variable `s`. The assignment operator can also be used to copy variables of the same structure. For instance, the statement,

```
s1 = s2;
```

copies contents of `s2` to `s1`, which are variables of the `student` structure. It is performed by copying each member transparently. Array of structure elements can also be copied as follows:

```

s[2] = s[1];
s[i] = s[j];

```

If a structure has members of type pointers, then only the address stored in that pointer member is copied and hence, such members still point to that pointed to by the source variable. In such a situation, make sure that memory is allocated, and explicitly copy the elements pointed to by the pointers. If this is not done, it might result in a dangling reference. It happens when the destination variable releases memory and the source variable continues to exist. (*Dangling reference*: it refers to a situation when a pointer to the memory item continues to exist, but memory allocated to that item is released. *Garbage memory*: it indicates that the memory item continues to exist but the pointer to it is lost; it happens when memory is not released explicitly.)