OBJECT ORIENTED PROGRAMMING IN C++

Module 2

Structures

Array is a data structure whose elements are all of the same data type. The structure is a data structure whose individual elements can differ in type. Thus, a single structure might contain integer elements, floating-point elements and character elements. Pointers, arrays and other structures can also be included as elements within a structure. The individual structure elements are referred to as members. We will see how structures are defined, and how their individual members are accessed and processed within a program.

Closely associated with the structure is the union, which also contains multiple members. Unlike a structure, however, the members of a the union share the same storage area, even though the individual members may differ in type. Thus, a union permits several different data items to be stored in the same portion of the computer's memory at different times.

Defining a Structure : Structure declarations are somewhat more complicated than array declarations, since a structure must be defined in terms of its individual members. In general terms, the composition of a structure may be defined as

```
struct tag
{
        member 1;
        member 2;
        member m;
} g;
```

In this declaration, struct is a required keyword; tag is a name that identifies structures of this type (structures having this composition); and member 1, member 2, . . . , member m are individual member declarations.

The individual members can be ordinary variables, pointers, arrays, or other structures. The member names within a particular structure must be distinct from one another, though a member name can be the same as the name of a variable that is defined outside of the structure. A storage class, however, cannot be assigned to an individual member, and individual members cannot be initialized within a structure type declaration. Once the composition of the structure has been defined, individual structure type variables can declare as follows.

```
storage-class struct tag variable 1, variable 2,…, variable n;
```

where

storage-class is an optional storage class specifier,
struct is a required keyword,
tag is the type name that is used in the structure declaration,
variable 1, variable 2,…., variable n are structure variables of type tag

In C++ we can omit struct and simply write :

```
tag variable;
```

e.g.

Structure definition.

```
struct account
 {
        int accnum;
        char acctype;
        char name[25];
        float balance;
```

```
};
```

Structure declaration.

```
struct account oldcust; [or]
account newcust;
```

Processing a Structure : The members of structure are usually processed individually as separate entities. Therefore, we must be able to access the individual structure members. A structure member can be accessed by writing

variable. member

Where variable refers to the name of a structure-type variable, and member refers to the name of a member within the structure. Notice the period (.) that separates the variable name from the member name. This period is an operator; it is a member of the highest precedence group, and its associativity is left - to - right.

Structure definition.

```
struct account
{
        int accnum;
        char acctype;
        char name[25];
        float balance;
};
```

Structure declaration.

```
struct account oldcust; [or]
account newcust;
```

e.g.

```
newcust.balance = 100.0
cout << oldcust.name;
```

We can initialize the members as follows :

e.g.

```
account customer = { 100, 'w', 'David', 6500.00};
```

We cannot copy one structure variable into another. If this has to be done then we have to do memberwise assignment.

Nested Structures

We can also have nested structures as shown in the following example :

```
struct  date
{
        int dd, mm, yy;
};

struct account
{
        int accnum;
```

```
        char acctype;
        char name[25];
        float balance;
        struct date d1;
};
```

Now if we have to access the members of date then we have to use the following method.

```
account c1;

c1.d1.dd=21;
```

Passing Structure to a Function

A structure can be passed as a function argument just like any other variable. If we are only interested in one member of a structure, it is probably simpler to just pass that member. When a structure is passed as an argument, each member of the structure is copied. This can prove expensive where structures are large or functions are called frequently. Passing and working with pointers to large structures may be more efficient in such cases.

e.g.:

```
# include <stdio.h>
struct customer
{
        int id;
        char name[15];
};
void func_struct(struct customer);

main()
{
        struct customer cus1={101,"George"};
        func_struct(cus1);
}

void func_struct(struct customer temp)
{
        printf("%d",temp.id);
        printf("%s",temp.name);
}
```

Array of Structures

Array of Structures is a collection of similar data types, which are declared as structures. It is useful to store different and large number of structure variables. To declare an array of structures, you must first define a Structure and then declare an array variable of that type.

For example,

```
        struct customer
        {
                int id;
                char name[15];
        };
        struct customer c[100]; .
```

In the above example, the array of Structures is used to find the details of 100 different customers. This statement provides space in memory for 100 structures of the type struct customer. The syntax to reference to each

element of the array c is similar to ordinary arrays of *ints* and *chars*. i.e. the first customer's name will be referenced as c[1].name.

In an array of structures, all the elements of the array are stored in adjacent memory locations. The elements of the array are in structures. And since all structure elements are stored in adjacent memory locations, it's easy to understand its arrangement in memory. For example, in memory c[0] 's id, name would be immediately followed by c[1] 's id and name and so on.

Structure is a collection of dissimilar data types whereas array is a collection of similar data types. Then the array of structure means an array of similar data types which themselves are a collection of dissimilar data types.

Self-Referential Structures

It is sometimes desirable to include within a structure one member that is a pointer to the parent structure type. In general terms, this can be expressed as

```
struct tag
{
        member 1;
        member 2;
        . . . . .
        struct tag *name;
} ;
```

where name refers to the name of a pointer variable. Thus, the structure of type tag will contain a member that points to another structure of type tag. Such structures are known as self-referential structures.

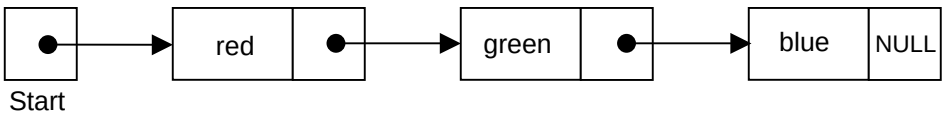
A C program contains the following structure declaration.

```
struct list_element
{
        char item[40];
        struct list_element *nextj
} ;
```

This is a structure of type list_element. The structure contains two members: a 40-element character array, called item, and a pointer to a structure of the same type (i.e., a pointer to a structure of type list_element), called next. Therefore this is a self-referential structure. .

Self-referential structures are very useful in applications that involve linked data structures, such as lists and trees. The basic idea of a linked data structure is that each component within the structure includes a pointer indicating where the next component can be found. Therefore, the relative order of the components can easily be changed simply by altering the pointers. In addition, individual components can easily be added or deleted again by altering the pointers. As a result, a linked data structure is not confined to some maximum number of components. Rather, the data structure can expand or contract in size as required.

Figure shown below illustrates a linked list containing three components. Each component consists of two data items: a string, and a pointer that references the next component within the list. Thus, the first component contains the string “red”, the second contains “green” and the third contains “blue”. The beginning of the list is indicated by a separate pointer, which is labeled “start”. Also, the end of the list is indicated by a special pointer, called “NULL”.



Enumerated Constants

Enumerated constants enable the creation of new types and then define variables of these types so that their values are restricted to a set of possible values.

e.g.

```
enum Colour{RED, BLUE, GREEN, WHITE, BLACK};
```

Colour is the name of an enumerated data type. It makes RED a symbolic constant with the value 0, BLUE a symbolic constant with the value 1 and so on.

- Every enumerated constant has an integer value. If the program doesn't specify otherwise, the first constant will have the value 0, the remaining constants will count up by 1 as compared to their predecessors.

- Any of the enumerated constant can be initialised to have a particular value, however, those that are not initialised will count upwards from the value of previous variables.

  e.g.

  ```
  enum Colour{RED = 100, BLUE, GREEN = 500, WHITE, BLACK = 1000};
  ```

The values assigned will be RED = 100,BLUE = 101,GREEEN = 500,WHITE = 501,BLACK = 1000

- You can define variables of type Colour, but they can hold only one of the enumerated values. In our case RED,BLUE,GREEEN,WHITE,BLACK .

- You can declare objects of enum types.

  e.g.

  ```
  enum Days{SUN, MON, TUE, WED, THU, FRI, SAT};
  Days day;
  Day = SUN;
  Day = 3;              // error int and day are of different types
  Day = hello;          // hello is not a member of Days.
  ```

- Even though enum symbolic constants are internally considered to be of type unsigned int we cannot use them for iterations.

  e.g.

  ```
  enum Days{SUN, MON, TUE, WED, THU, FRI, SAT};
  for(enum i  = SUN; i<SAT; i++)        //not allowed.
  ```

  There is no support for moving backward or forward from one enumerator to another.

- However whenever necessary, an enumeration is automatically promoted to arithmetic type.

  e.g.

  ```
  if( MON > 0)
   {
          cout << ”ghjf”;
   }

  int num = 2*MON;
  ```

  These are allowed.

Unions

A union is also like a structure , except that only one variable in the union is stored in the allocated memory at a time. It is a collection of mutually exclusive variables , which means all of its member variables share the same

physical storage and only one variable is defined at a time. The size of the union is equal to the largest member variables. A union is defined as follows :

```
union tag
 {
        type memvar1;
        type memvar2;
        type memvar3;
        :
        :
 };
```

A union variable of this data type can be declared as follows,

```
Union tag variable_name;
```

In C++ we can omit struct and simply write :

```
tag variable_name;
```

e.g.

```
union utag
 {
        int num;
        char ch;
 };

union utag filed;
utag fil;
```

The above union will have two bytes of storage allocated to it. The variable num can be accessed as field.sum and ch is accessed as field.ch. At any time, only one of these two variables, can be referred to. Any change made to one variable affects another.

Thus unions use memory efficiently by using the same memory to store all the variables, which may be of different types, which exist at mutually exclusive times and are to be used in the program only once.

Functions

A program is made up of one or more functions, with one of these being main(). Function is a self – contained block of program that performs a particular task.

Reasons for using functions :

1. Avoids rewriting the same code over and over.
2. It is easier to write programs and keep track of what they are doing. Codes can be checked independently and makes it error free for the correct execution of programs.

Function Definition

The C code that describes what a function does is called the function definition.

A function definition has the following form

```
Type    function name( parameter list)
{
        Declarations
        Statements;
```

```
}
```

Everything before the first brace comprises the header of the function definition and everything between the braces comprises the body of the function definition.

The type of the function depends on the type of value that the function returns if any. The type **void** can be used if a function does not return a value.

Function name must not be library routine or operating system commands.

Parameter list contains valid variable names separated by commas. Parameters are identifiers used within the body of the function. The parameters in function definition are called formal parameters to emphasize their role as place holders for actual values that are passed to the function, when it is called.

An example of a function definition

```
main()
{
        message();
} /* header */


message()
{
        /*bodystarts here*/
        printf ("Example for function definition\n");
        return;
}
```

The program will print the following output, **Example for function definition**

message() is the function here. In all c programs, program execution begins in main(). When program control encounters message(), the function is invoked and program control is passed to it. After printing the message(), the program control passes back to the calling environment, which in above example is main(). When main() runs out of function calls, the, program ends.

The return statement

The return statement is used for two purposes. Once the return statement is executed the program control will be immediately passed back to the calling environment. If the return statement contains an expression then the value of the expression is passed back to the calling environment as well. This value will be converted, if necessary to the type of the function as specified in the function definition. If no type is explicitly declared, the type is implicitly “int”.

A return statement has one of the following forms:

```
return;
return expression;
```

Some examples are

```
return(3);
return(a*b);
```

The expression being returned can be enclosed in parentheses, but it's not necessary.

Function Prototype

Functions should be declared before they are used. ANSI C provides for the new function declaration syntax called the function prototype. A function prototype tells the compiler the number and type of arguments that are to be passed to the function and the type of the value that is to be returned by the function.

An example is

```
double sqrt(double);
```

This tells the compiler that sqrt() is a function that takes a single argument of type double and returns a double. The general form of function prototype.

type *function_name****(parameter type*** *list****);***

Parameter Passing Mechanism

Parameters are syntactically identifiers, and they are used within the body of the function. Sometimes the parameters in a function definition are called formal parameters to emphasize their role as placeholders for actual values that are passed to the function when it is called. Upon function invocation, the value of the argument correspond to a formal parameter is used within the body of executing function. Before beginning with the statements in the functions, it is necessary to declare the type of the arguments through type declaration statements.

1. Call by value

Functions are invoked by writing their name and an appropriate list of arguments within parenthesis. Typically, these arguments are in the, parameter list of the function definition. All arguments are passed "call-by-value". Whenever we called a function and passed something to it we have always passed the values of variables to the called function. This type of function calls are called "call-by-value".

The examples of call-by-value are shown below:

```
sum=cal sum( a, b );
f=fact(a);
```

If a variable is passed to a function, the stored value of that variable in that variable in the calling environment will not be changed.

Here is .an example:

```
#include<stdio.h> .
int compute_sum(int m);
int main( void)
{
        int n=3, sum;
        printf("%d\n",n); /*3 is printed */
        sum=compute_sum(n);
        printf("%d\n",n); /*3 is printed */
        printf("%d\n",sum); /*6 is printed */
        return 0;
}

int compute_sum(int n) /*sum the integers from 1 to n*/ 9
{
        int sum=0;
        for (;n>O;--n) /*stored value of n is changed*/
        sum+=n;
        return sum;
}
```

Even though n is passed to compute_sum() and the value of n in the body of that function is changed, the value of n in the calling environment remains unchanged. It is the value of n that is being passed, not n itself.

2. Call by reference

In "call-by- reference", instead of passing the value of a variable, the location number (or the address) of the variable is passed to a function. This will become "call-by-reference". It is a way to pass address (reference) of variables to a function that then allows the body of the function to make changes to the value of the variables in the calling environment.

Here is .an example:

```
#include<stdio.h> .
int compute_sum(int *n);
int main( void)
{
        int n=3, sum;
        printf("%d\n",n); /*3 is printed */
        sum=compute_sum(&n);
        printf("%d\n",n); /*3 is printed */
        printf("%d\n",sum); /*6 is printed */
        return 0;
}

int compute_sum(int *n) /*sum the integers from 1 to n*/ 9
{
        int sum=0;
        for (;*n>O;--*n) /*stored value of n is changed*/
        sum+=*n;
        return sum;
}
```

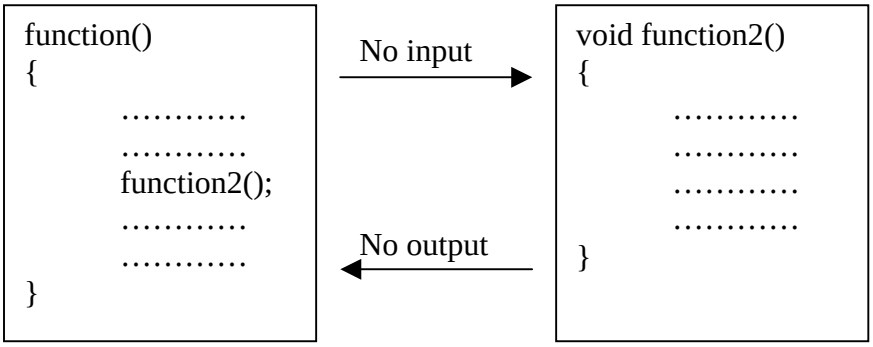
Category of functions

A function, depending on whether parameters or arguments are present or not and whether a value returned or not, may belong to one of the following categories.

1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and return values.

Functions with no arguments and no return values

When a function has no arguments, it does not receive any data from the calling function. Similarly, it does not return any value; the calling function does not receive any data from the called function.

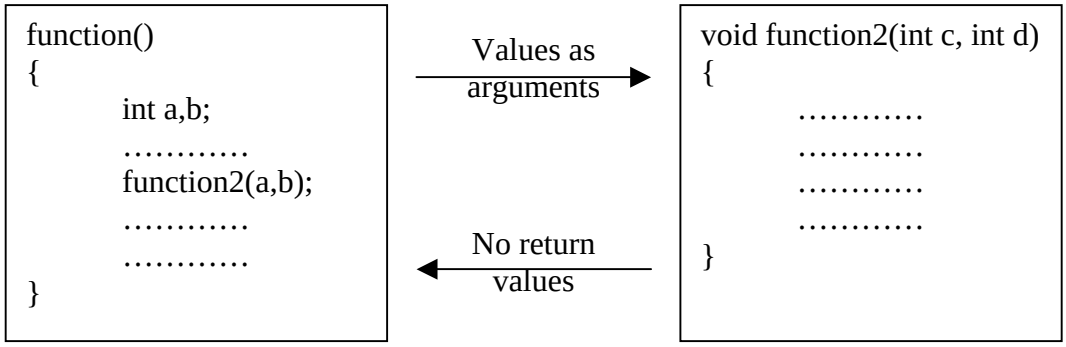


There is only transfer of control but not data. This function can only be used as an independent statement.

Functions with arguments and no return values

The calling function will read data from the terminal and pass it on to called function. This will work good as the calling function can check the validity of data, if necessary, before it is handed over to the called a function.

The nature of data communication between the calling function and called function with arguments with no return values is shown below.



Example

area(l,b,h);

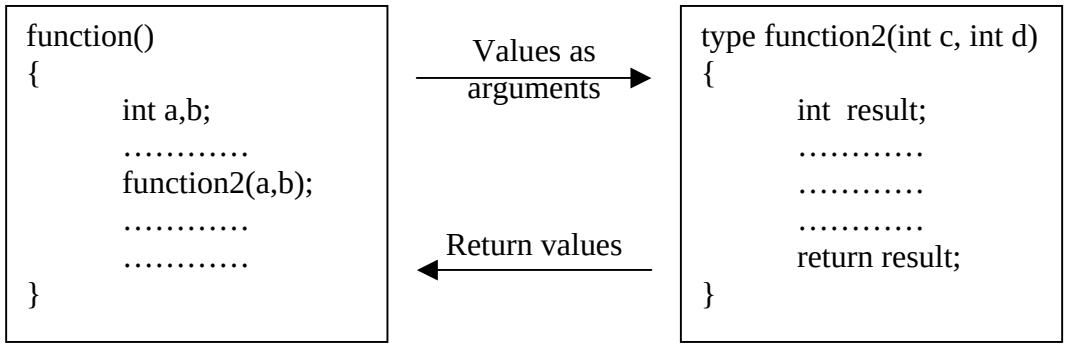
The arguments 1, b, h are called the formal parameters. The calling function can send values to these arguments using function calls. The function call area (3,4,5) would send the values 3,4,5 to the function area (l,b,h) and assign 3 to l, 4 to b, 5 to h. The values 3,4,5 are the actual parameters which become the values of the formal arguments inside the called function.

The actual and the formal arguments should match in number, and order. When a function call is made, only a copy of the values of actual arguments is passed into the called function.

Functions with arguments and return values

The previous category functions receive values from the calling function through arguments but do not send back any value. Rather it displays the result of calculations at the terminal. To ensure a higher degree of portability between programs, a function should generally be coded without involving any I/O operations. If the function returns a value, it can be used in different ways in different programs.

The nature of the two way data communication between the calling and the called function



The following events occur; when a function call is executed

(i) The function call transfers the control along with the copies of the values of actual parameters.

(ii) The called function is executed line by line until the return statement is encountered. At this point, the value is passed back to main() or calling function.

Recursion

In C, it is possible for the function to call themselves. A function is called 'recursive' if a statement within the body of a function calls the same function. Sometimes called 'circular definition', recursion is thus the process of defining something in terms of itself.

For a clear understanding of recursive function we shall see an example for calculating value of an integer.

```
main()
{
        int a,factorial;
        printf("\n enter any number");
        scanf("%d",&a);.
        factorial =fact(a);
        printf ("factorial value =%fctorial);
}

fact (int x)
{
        int f =1,i;
        for(i=x;i>=;i++ )
                f=f*i;
        return f;
}
```

And here is the output

Enter any number (3 is given as input)
Factorial value =6

Now we will see the program using recursion

```
main()
{
        int a,fact;
        printf(enter any number");
        scanf("%d",&a);
        fact =fact_rec(a);
        printf("\nfactorial value =%d",fact);
}

fact_rec(int x)
{
        int f,i;
        if (x==1)
                return 1;
        else
                f=x*fact_rec(x-1);
        return (f);
}
```

When a recursive program is executed, the recursive function calls are not executed immediately. Rather, they are placed on a stack until the condition that terminates the recursion is encountered. The function calls are then executed in reverse order, as they are “popped” off the stack. Thus, when evaluating a factorial recursively, the function calls will proceed in the following order.

n! = n! * (n-1)!
(n-1)! = (n-1)*(n-2)!
(n-2)! = (n-2)*(n-3)!
……
2! = 2*1!
1! =1

The actual values will then be returned in the following reverse order

1! = 1
2! = 2*1! = 2*1 = 2
3! = 3*2! = 3*2 = 6
……
……
n! = n * (n-1)!

This reversal in the order of execution is a characteristic of all functions that are executed recursively.

The recursive function fact_rec() is analyzed as illustrated in the following table.

Function call	**Value returned**
fact_rec(1)	1
fact_rec(2)	2*fact_rec(1) or 2*1
fact_rec(3)	2*fact_rec(2) or 3*2* 1
fact_rec(4)	2*fact_rec(3) or 4*3*2*1

In the first run, when the value of x = 1, it checks the if condition, if (x=1) is satisfied and is returned through return statement, f =x*fact_rec(x-l). So this becomes f =2*fact_rec(1). We know that fact_rec(1) is 1, so the expression reduces to (2 * 1) or 2.

Recursive functions can be efficiently used to solve problems where the solution is expressed in terms of successively applying the same solution to subsets of the problem. When we use recursive functions, we must haye an if statement, somewhere to force the function to return without recursive call being executed. Otherwise, the function will never return.

Inline Functions

Imagine a c program, which reads disk records containing employee information. If this is a payroll application each employee record data is probably processed via a series of function calls. These function calls could be to print the entity data, to compute the salary, to compute the taxes to be withheld etc,. Each one of these calls inherently contains overhead that must be part of your overall program. In other words it takes code space and time to push actual arguments onto the stack, call the function, push some return value onto the stack and finally pop all of those values .

C++ provides the inline functions, a mechanism by which these explicit function calls can be avoided.

" An inline function by definition is a function whose code gets substituted in place of the actual call to that function."

Whenever the compiler encounters a call to that function it merely replaces it with the code itself thereby saving you all of the overhead. Such a function can be either a member of a class or a global function. Inline functions work best when they are small, straightforward bodies of code that are not called from too many different places within your program. The time saved will increase with the increase in number of calls.

Even if you request that the compiler make a function into an inline function, the compiler may or may not honor that request. It depends on the code of the function. For example, the compiler will not inline any function that contains a loop, static data member, or aggregate initializer list. If such cases case, a warning message will be issued.

The disadvantage with inline functions is that if the code itself ever needs to be modified, then all programs that use the these functions would then have to be recompiled. Furthermore, an inline function is a violation of implementation hiding.

How to write a global inline function

First, let's get away from member functions for a moment and consider a global function. To make a request that this function be inline :

- Precede the return type of the function with the keyword inline.
- Write the function body (the definition, not just the declaration) before any calls to that function.

Here is a program that uses an inline function to compute and return the absolute value of its input argument.

```
# include <stdio.h>

inline int abs(int x)
{
        return x <0? -x : x ;
}

void main( )
 {
          for (int i=-2 ; i<2 ; ++i)
          {
                  int value = abs(i) ;
                  printf ("Absolute value of %+d = %+d\n",i, value);
        }
}
```

The output is:

```
Absolute value of –2 =+2
Absolute value of –1 =+1
Absolute value of +0 =+0
Absolute value of +1 =+1
```

When the call to the abs ( ) function is encountered, the compiler, instead of making a function call, generates this assembly code.

Variables

Information stored in a variable can change in the course of the program. The type used in the definition describes the kind of information the symbol can store. *Variables are addressable.*

Rules for naming the variables:

- Can have characters, digits and underscore.
- Can start with character or underscore.
- Are case sensitive i.e., fan is different from Fan and FAN.
- Can't use keywords
- No limits on the length of the word.

e.g.

```
int poodle          //  Valid
int my_stud         //  Valid
int _mystud         //  Valid
int 4ever                   //  Invalid  starts with number
int double          //  Invalid  keyword
int Honk-Kong       //  Invalid  –(dash) not allowed as it is a special character.
```

Variables help us to reuse the codes. To find the value of power of some number we may write,

e.g.

To find the value of power of some number we may write,

```
cout << 2*2*2*2*2*2*2*2*2*2  for 2^8.     /* cout is equivalent  to  printf() in C. It prints the literals or values of
                                            variables.*/
```

If we want to find the power of some number or if we want to change the number we have to type the whole sentence again. We can generalize this operation by using 2 variables as follows.

e.g.

```
int val = 4, pow =10, cnt  ;
for ( cnt =1; cnt <= pow; ++cnt )
  {
        res = res*val;
  }
```

This is flexible compared to previous method but not reusable. By using functions we can make them reusable too.

e.g.

```
int  pow( int val , int exp )
  {
        for ( res =1 ; exp>0 ; --exp )
    {
      res = res * val;
      return res;
    }
  }
```

You can call this function any number of times with different arguments. **Variables can alternatively be called as Objects.**

There are two terms associated with a variable :

1. r-value (read value) : Data. This can be a literal or variable name.
2. l-value (location value) : Address in the memory location where data is stored.

e.g.

```
ch = ch1 - '0';
```

where ch is l-value,ch1 is r-value symbol,'0' is r-value literal.

You can have the same symbol in both l-value and r-value.

e.g.

```
ch = ch - '0';
```

```
0 = 1;        // Error, cannot have a literal as l-value.
```

salary + extra = newsal // Error, l value must be an addressable variable and not an expression.

Objects or variables can have only a single location. So, they can't be defined twice. However, they can be declared twice. Definition sets memory aside for a variable where as declaration just informs the compiler that this variable is defined somewhere in the program.

Some variables may be needed in two or more files. In such cases we can define it in one file and declare it in all the files that use this variable.

e.g.

```
// file fileno1.cpp
        string filename;
        :
        :

// file fileno2.cpp
        extern string filename;
        :
        :
```

The keyword *extern* informs the compiler that the filename is defined somewhere outside the existing file. If you need to declare many variables in different files you can declare once in a header file and include it in files where the declarations are needed.

The definition of an object(variable):

We can define a variable(set memory to the variable) in the following ways.

e.g.

```
double salary;
int month;
```

When more than one variable of same type are needed a comma can be used to declare multiple variables.

e.g.

```
double sal , wage;
int m , d , y ,
k , a ;          // Can span multiline too.
```

If the variables are declared globally, they are initialized to zero by default constructors(functions that are used to initialize the variables) that are inbuilt . If they are declared locally we have to initialize them as they may have some junk value stored in them.

e.g.

```
int x;   // Global , initialized to zero.
void main()
{
    int i;          //  Local , not initialized;
}
```

You can initialize the variables in the following ways.

e.g.

```
int  ival =100;
int ival(100);

double sal =10.0, wage = sal + 10;

int mon = 06,
day  = 19,
year =1975;
```

In C++, special constructors are inbuilt that support initialization.

e.g.

```
int ival = int();          // Sets ival = 0 ;
double dval = double() ;   // Sets dval to 0.0;
```

An object can be initialized with an arbitrary complex expression.

e.g.

```
double pr =199.99. disc = 0.16;
double sale( pr * disc );
```

Scope Rules and Storage Classes

The storage class determines the life of a variable in terms of its duration or its scope. There are four storage classes :

automatic
static
external
register

Automatic Variables

Automatic variables are variable which are defined within the functions. They lose their value when the function terminates. It can be accessed only in that function. All variables when declared within the function are , by default, are automatic. However, we can explicitly declare them by using the keyword *'automatic'.*

e.g.

```
void print()
 {
   auto int i =0;

        cout << " Value of i before incrementing is :<< i;
        i = i + 10;
        cout <<"\n"<< " Value of i after incrementing is :<<i;
 }

void main()
 {
        print();
        print();
        print();
 }
```

Output:

```
 Value of i before incrementing is : 0
 Value of i after incrementing  is : 10
 Value of i before incrementing is : 0
 Value of i after incrementing  is : 10
 Value of i before incrementing is : 0
 Value of i after incrementing  is : 10
```

Static Variables

Static variables have the same scope s automatic variables, but , unlike automatic variables, static variables retain their values over number of function calls. The life of a static variable starts, when the first time the function in which it is declared, is executed and it remains in existence, till the program terminates. They are declared with the keyword *static.*

e.g.

```
void print()
{
  static int i =0;

        cout << " Value of i before incrementing is :<< i;
        i = i + 10;
        cout <<"\n"<< " Value of i after incrementing is :<<i;

}

void main()
{
        print();
        print();
        print();
}
```

Output:

```
Value of i before incrementing is : 0
Value of i after incrementing  is : 10
Value of i before incrementing is : 10
Value of i after incrementing  is : 20
Value of i before incrementing is : 20
Value of i after incrementing  is : 30
```

It can be seen from the above example that the value of the variable is retained when the function is called again. It is allocated memory and is initialized only for the first time.

External Variables

Different functions of the same program can be written in different source files and can be compiled together. The scope of a global variable is not limited to any one function , but is extended to all the functions that are defined after it is declared. However, the scope of a global variable is limited to only those functions, which are in the same file scope. If we want to use a variable defined in another file, we can use *extern* to declare them.

e.g.

```
// FILE 1 – g is global and can be used only in main() and // // fn1();
int g = 0;
void main()
{
        :
        :
}

void fn1()
{
        :
        :
}


// FILE 2 If the variable declared in file 1 is required to be used in file 2 then it is to be declared as an extern.
extern int g = 0;
void fn2()
{
        :
        :
}
```

```
void fn3()
 {
        :
 }
```

Register Variable

Computers have internal registers, which are used to store data temporarily, before any operation can be performed. Intermediate results of the calculations are also stored in registers. Operations can be performed on the data stored in registers more quickly than on the data stored in memory. This is because the registers are a part of the processor itself. If a particular variable is used often – for instance, the control variable in a loop, can be assigned a register, rather than a variable . This is done using the keyword *register.* However, a register is assigned by the compiler only if it is free, otherwise it is taken as automatic. Also, global variables cannot be register variables.

e.g.

```
void loopfn()
 {
  register int i;

        for(i=0; i< 100; i++)
         {
                cout << i;
         }
 }
```