CHAPTER 11

USER DEFINED FUNCTIONS

Function Definition

- **Using functions we can structure our programs in a more modular way, accessing all the potential that structured programming can offer to us in C++.**

  A function is a group of statements that is executed when it is called from some point of the program. The following is its format:

Function Definition

```
type name ( parameter1, parameter2, ...)
{
      statements
}
```

where:

- type is the data type specifier of the data returned by the function.
- name is the identifier by which it will be possible to call the function.
- parameters (as many as needed): Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: int x) and which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas.
- statements is the function's body. It is a block of statements surrounded by braces { }.

Here you have the first function example:

```
// function example
#include <iostream>
int addition (int a, int b)
{
  int r;
  r=a+b;
  return (r);
}
int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
  return 0;
}
```

WORKING OF FUNCTION

- We can see how the main function begins by declaring the variable z of type int. Right after that, we see a call to a function called addition. Paying attention we will be able to see the similarity between the structure of the call to the function and the declaration of the function itself some code lines above:

  The parameters and arguments have a clear correspondence. Within the main function we called to addition passing two values: 5 and 3, that correspond to the int a and int b parameters declared for function addition

WORKING OF FUNCTION

- At the point at which the function is called from within main, the control is lost by main and passed to function addition. The value of both arguments passed in the call (5 and 3) are copied to the local variables int a and int b within the function.

  Function addition declares another local variable (int r), and by means of the expression r=a+b, it assigns to r the result of a plus b. Because the actual parameters passed for a and b are 5 and 3 respectively, the result is 8.

  The following line of code:

- *return* (r);

WORKING OF FUNCTION

- finalizes function addition, and returns the control back to the function that called it in the first place (in this case, main). At this moment the program follows its regular course from the same point at which it was interrupted by the call to addition. But additionally, because the return statement in function addition specified a value: the content of variable r (return (r);), which at that moment had a value of 8. This value becomes the value of evaluating the function call.

So being the value returned by a function the value given to the function call itself when it is evaluated, the variable z will be set to the value returned by addition (5, 3), that is 8. To explain it another way, you can imagine that the call to a function (addition (5,3)) is literally replaced by the value it returns (8).

The following line of code in main is:

```
cout << "The result is " << z;
```

FUNCTION PROTOTYPE

A Function Prototype is a declaration of the function that tells the program about the type of the value returned by the function and the type of argument.

FUNCTION PROTOTYPE

DECLARING FUNCTIONS

Until now, we have defined all of the functions before the first appearance of calls to them in the source code. These calls were generally in function main which we have always left at the end of the source code. If you try to repeat some of the examples of functions described so far, but placing the function main before any of the other functions that were called from within it, you will most likely obtain compiling errors. The reason is that to be able to call a function it must have been declared in some earlier point of the code, like we have done in all our examples.

FUNCTION PROTOTYPE

But there is an alternative way to avoid writing the whole code of a function before it can be used in main or in some other function. This can be achieved by declaring just a prototype of the function before it is used, instead of the entire definition. This declaration is shorter than the entire definition, but significant enough for the compiler to determine its return type and the types of its parameters.

FUNCTION PROTOTYPE

Its form is:

type name (argument_type1, argument_type2, ...);

It is identical to a function definition, except that it does not include the body of the function itself (i.e., the function statements that in normal definitions are enclosed in braces { }) and instead of that we end the prototype declaration with a mandatory semicolon (;).

FUNCTION PROTOTYPE

The parameter enumeration does not need to include the identifiers, but only the type specifiers. The inclusion of a name for each parameter as in the function definition is optional in the prototype declaration. For example, we can declare a function called protofunction with two int parameters with any of the following declarations:

int protofunction (*int* first, *int* second);

int protofunction (*int*, *int*);

FUNCTION PROTOTYPE

```
// declaring functions prototypes
#include <iostream>
void odd (int a);
void even (int a);
int main ()
{
  int i;
  do {
    cout << "Type a number (0 to exit): ";
    cin >> i;
    odd (i);
  } while (i!=0);
  return 0;
}
```

FUNCTION PROTOTYPE

```
void odd (int a)
{
  if ((a%2)!=0) cout << "Number is odd.\n";
  else even (a);
}
void even (int a)
{
  if ((a%2)==0) cout << "Number is even.\n";
  else odd (a);
}
```

FUNCTION PROTOTYPE

This example is indeed not an example of efficiency. I am sure that at this point you can already make a program with the same result, but using only half of the code lines that have been used in this example. Anyway this example illustrates how prototyping works. Moreover, in this concrete example the prototyping of at least one of the two functions is necessary in order to compile the code without errors.

FUNCTION PROTOTYPE

The first things that we see are the declaration of functions odd and even:

```
void odd (int a);
void even (int a);
```

This allows these functions to be used before they are defined, for example, in main, which now is located where some people find it to be a more logical place for the start of a program: the beginning of the source code.

Anyway, the reason why this program needs at least one of the functions to be declared before it is defined is because in odd there is a call to even and in even there is a call to odd. If none of the two functions had been previously declared, a compilation error would happen, since either odd would not be visible from even (because it has still not been declared), or even would not be visible from odd (for the same reason).

FUNCTION PROTOTYPE

Having the prototype of all functions together in the same place within the source code is found practical by some programmers, and this can be easily achieved by declaring all functions prototypes at the beginning of a program.

DEFAULT ARGUMENTS

When declaring a function we can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead. For example:

DEFAULT ARGUMENTS

```
// default values in functions
#include <iostream.h>
int divide (int a, int b=2)
{
  int r;
  r=a/b;
  return (r);
}
int main ()
{
  cout << divide (12);
  cout << endl;
  cout << divide (20,4);
  return 0;
}
```

DEFAULT ARGUMENTS

As we can see in the body of the program there are two calls to function divide. In the first one:

divide (12)

we have only specified one argument, but the function divide allows up to two. So the function divide has assumed that the second parameter is 2 since that is what we have specified to happen if this parameter was not passed (notice the function declaration, which finishes with int b=2, not just int b). Therefore the result of this function call is 6 (12/2).

DEFAULT ARGUMENTS

In the second call:

divide (20,4)

there are two parameters, so the default value for b (int b=2) is ignored and b takes the value passed as argument, that is 4, making the result returned equal to 5 (20/4).

CONSTANTS

By constant arguments meant that the function can not modify these arguments. If you pass constant values to the function, then the function can not modify the values as the values are constants.

In order to make the argument constant to the function we can use the keyword **const** as shown bellow,

```
int sum(const int a , const int b);
```

CALL BY VALUE

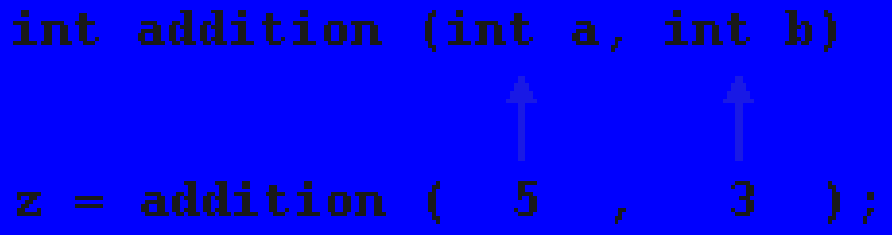
Until now, in all the functions we have seen, the arguments passed to the functions have been passed *by value*. This means that when calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves. For example, suppose that we called our first function addition using the following code:

```
int x=5, y=3, z;
z = addition ( x , y );
```

What we did in this case was to call to function addition passing the values of x and y, i.e. 5 and 3 respectively, but not the variables x and y themselves.

CALL BY VALUE

What we did in this case was to call to function addition passing the values of x and y, i.e. 5 and 3 respectively, but not the variables x and y themselves.



CALL BY VALUE

This way, when the function addition is called, the value of its local variables a and b become 5 and 3 respectively, but any modification to either a or b within the function addition will not have any effect in the values of x and y outside it, because variables x and y were not themselves passed to the function, but only copies of their values at the moment the function was called.

CALL BY VALUE

But there might be some cases where you need to manipulate from inside a function the value of an external variable. For that purpose we can use arguments passed by reference, as in the function duplicate of the following example:

CALL BY REFERENCE

The call by reference method uses a different mechanism. In place of passing value to the function being called.

A reference is an alias (i.e different name) for a predefined variable.

CALL BY REFERENCE

There might be some cases where you need to manipulate from inside a function the value of an external variable. For that purpose we can use arguments passed by reference, as in the function duplicate of the following example:

CALL BY REFERENCE

```
// passing parameters by reference
#include <iostream>
void duplicate (int& a, int& b, int& c)
{
  a*=2;
  b*=2;
  c*=2;
}
int main ()
{
  int x=1, y=3, z=7;
  duplicate (x, y, z);
  cout << "x=" << x << ", y=" << y << ", z=" << z;
  return 0;
}
```

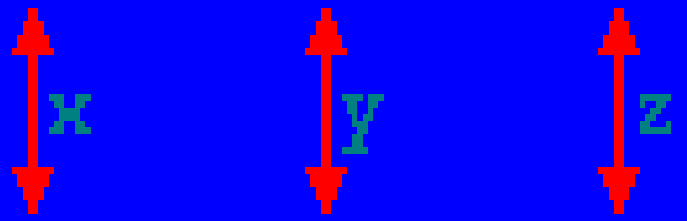
CALL BY REFERENCE

The first thing that should call your attention is that in the declaration of duplicate the type of each parameter was followed by an ampersand sign (&). This ampersand is what specifies that their corresponding arguments are to be passed *by reference* instead of *by value*.

When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the local variables will have an effect in their counterpart variables passed as arguments in the call to the function.

CALL BY REFERENCE

```
void duplicate (int& a,int& b,int& c)
```




```
    duplicate (   x   ,   y   ,   z   );
```

CALL BY REFERENCE

To explain it in another way, we associate a, b and c with the arguments passed on the function call (x, y and z) and any change that we do on a within the function will affect the value of x outside it. Any change that we do on b will affect y, and the same with c and z.

That is why our program's output, that shows the values stored in x, y and z after the call to duplicate, shows the values of all the three variables of main doubled.

CALL BY REFERENCE

If when declaring the following function:

void **duplicate (*int*& a, *int*& b, *int*& c)**

we had declared it this way:

void **duplicate (*int* a, *int* b, *int* c)**

CALL BY REFERENCE

```
// more than one returning value
#include <iostream>
using namespace std;
void prevnext (int x, int& prev, int& next)
{
  prev = x-1;
  next = x+1;
}
int main ()
{
  int x=100, y, z;
  prevnext (x, y, z);
  cout << "Previous=" << y << ", Next=" << z;
  return 0;
}
```

CALLING FUNCTION WITH ARRAYS

Arrays as parameters

At some moment we may need to pass an array to a function as a parameter. In C++ it is not possible to pass a complete block of memory by value as a parameter to a function, but we are allowed to pass its address. In practice this has almost the same effect and it is a much faster and more efficient operation.

In order to accept arrays as parameters the only thing that we have to do when declaring the function is to specify in its parameters the element type of the array, an identifier and a pair of void brackets []. For example, the following function:

CALLING FUNCTION WITH ARRAYS

void procedure (*int* arg[])

accepts a parameter of type "array of int" called arg. In order to pass to this function an array declared as:

CALLING FUNCTION WITH ARRAYS

int myarray [40];

it would be enough to write a call like this:.

procedure (myarray);

CALLING FUNCTION WITH ARRAYS

```
// arrays as parameters
#include <iostream>
using namespace std;
void printarray (int arg[], int length) {
  for (int n=0; n<length; n++)
    cout << arg[n] << " ";
  cout << "\n";
}
int main ()
{
  int firstarray[] = {5, 10, 15};
  int secondarray[] = {2, 4, 6, 8, 10};
  printarray (firstarray,3);
  printarray (secondarray,5);
  return 0;
}
```

CALLING FUNCTION WITH ARRAYS

As you can see, the first parameter (int arg[]) accepts any array whose elements are of type int, whatever its length. For that reason we have included a second parameter that tells the function the length of each array that we pass to it as its first parameter. This allows the for loop that prints out the array to know the range to iterate in the passed array without going out of range.

In a function declaration it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is:

base_type[][depth][depth]

for example, a function with a multidimensional array as argument could be:

void procedure (*int* myarray[][3][4])

CALLING FUNCTION WITH ARRAYS

First Way

```
#include <iostream.h>
void main()
{
int age[10];
cout<<“enter the elements”;
 for(i=0;i<n;i++)
cin>>age[i];
display(age);
}
```

```
void display (int a[10])
{
for (i=0;i<10;i++)
cout<<“\n”<<a[i];
}
```

See receiving parameter has been declared with its size

See for passing array, only its name (age) is passed as an argument

CALLING FUNCTION WITH ARRAYS

Second way

```
#include <iostream.h>
void main()
{
int age[10];
cout<<“enter the elements”;
 for(i=0;i<n;i++)
cin>>age[i];
display(age);
}
```

```
void display (int a[ ] )
{
for (i=0;i<10;i++)
cout<<“\n”<<a[i];
}
```

See receiving parameter has been declared without specifying its size

See for passing array, only its name (age) is passed as an argument

CALLING FUNCTION WITH ARRAYS

Third way

```
#include <iostream.h>
void main()
{
int age[10];
cout<<“enter the elements”;
 for(i=0;i<n;i++)
cin>>age[i];
display(age);
}
```

See receiving parameter has been declared as pointer of same base type

See for passing array, only its name (age) is passed as an argument

RECURSION

Recursivity is the property that functions have to be called by themselves. It is useful for many tasks, like sorting or calculate the factorial of numbers. For example, to obtain the factorial of a number (n!) the mathematical formula would be:

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

more concretely, 5! (factorial of 5) would be:

5! = 5 * 4 * 3 * 2 * 1 = 120

and a recursive function to calculate this in C++ could be:

RECURSION

```
// factorial calculator
#include <iostream>
long factorial (long a)
{
  if (a > 1)
   return (a * factorial (a-1));
  else
   return (1);
}
int main ()
{
  long number;
  cout << "Please type a number: ";
  cin >> number;
  cout << number << "! = " << factorial (number);
  return 0;
}
```

RECURSION

Notice how in function factorial we included a call to itself, but only if the argument passed was greater than 1, since otherwise the function would perform an infinite recursive loop in which once it arrived to 0 it would continue multiplying by all the negative numbers (probably provoking a stack overflow error on runtime).

This function has a limitation because of the data type we used in its design (long) for more simplicity. The results given will not be valid for values much greater than 10! or 15!, depending on the system you compile it.

RETURNING FROM A FUNCTION

As invoking a function is important, returning from a function is equally important as it not only terminates the functions execution but also passes the control back to the calling function.

THE RETURN STATEMENT

The return statement is useful in two ways. First an immediate exit from the function is caused as soon as the return statement is encountered and the control is passes back to statement following the called function in the calling code. The general format of return statement is ,

```
return(value);
```

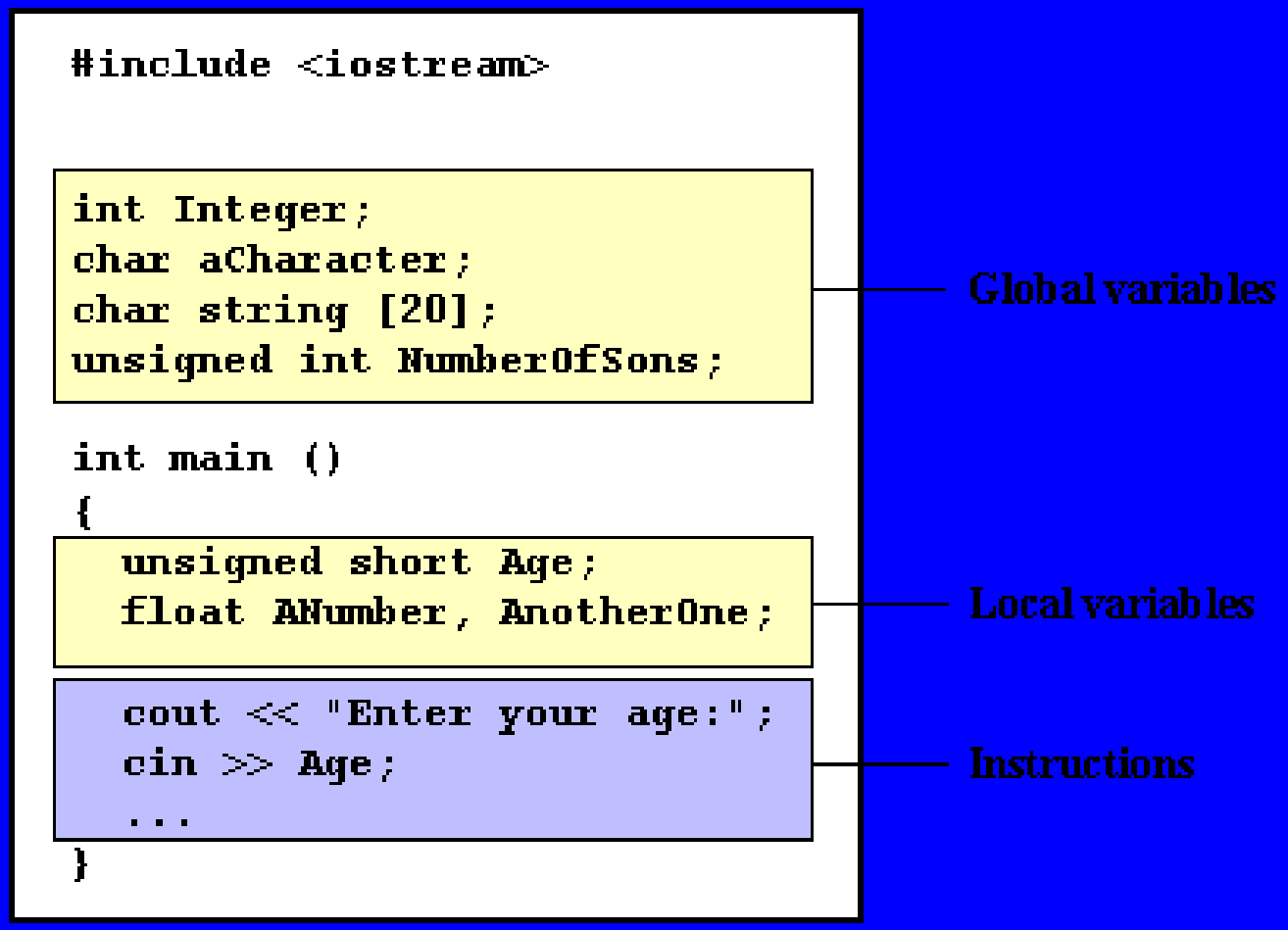
SCOPE OF VARIABLE

The scope of variables declared within a function or any other inner block is only their own function or their own block and cannot be used outside of them. For example, in the previous example it would have been impossible to use the variables a, b or r directly in function main since they were variables local to function addition. Also, it would have been impossible to use the variable z directly within function addition, since this was a variable local to the function main.

SCOPE OF VARIABLE

Therefore, the scope of local variables is limited to the same block level in which they are declared. Nevertheless, we also have the possibility to declare global variables; These are visible from any point of the code, inside and outside all functions. In order to declare global variables you simply have to declare the variable outside any function or block; that means, directly in the body of the program.

SCOPE OF VARIABLE



INLINE FUNCTION

The inline specifier indicates the compiler that inline substitution is preferred to the usual function call mechanism for a specific function. This does not change the behavior of a function itself, but is used to suggest to the compiler that the code generated by the function body is inserted at each point the function is called, instead of being inserted only once and perform a regular call to it, which generally involves some additional overhead in running time.

The format for its declaration is:

```
inline type name ( arguments ... )
{
        Statements
                                    }
```

and the call is just like the call to any other function. You do not have to include the inline keyword when calling the function, only in its declaration.

Most compilers already optimize code to generate inline functions when it is more convenient. This specifier only indicates the compiler that inline is preferred for this function.

STRUCTURES

In C++ language, custom data types can be created to meet users requirements in 5 ways: class, structure, union, enumeration and typedef.

Structures are one of the 2 important building blocks in the understanding of classes and objects. A structure is a collection of data and functions. In other words, we can say, a structure plus related functions make a class. Technically ,there is no difference between a structure and a class. In fact,a structure is a class declared with keyword struct and by default, all members are public in a structure whereas all members are private by default in the class.

STRUCTURES

Sometimes, some logically related elements need to be treated under one unit. For instance, the elements storing a student's information (e.g., rollno, name, class, marks, grade) need t be processed together under one roof. Similarly, elements keeping a date's information (e.g., day, month, and year) need to be processed together. To handle and serve to such situation, C++ offers structures.

Thus it can be said that ,

A C style structures is a collection of variables referenced under one name.

STRUCTURES

The following code fragment shows how to define a structure (say date). The keyword struct tells the compiler that a structure is being defined.

```
struct date {  short day;
               short month;
               short year;};
```

STRUCTURES

In the above definition, the date is a structure tag and it identifies this particular data structure and its type specifier. Also remember, at this point of time (i.e., after the above definition), no structure variable has been declared, that is , no memory space has bee reserved. Only the form of the data has been defined. To declare a structure variable having the data form as defined by date, we will write

```
Date joining_date;
```

This declares a structure variable joining_date of type date. Thus, the complete structure definition is as follows:

```
struct date
{
        short day ;
        short month;
        short year;
};
Date joining _date ;
```

Now, the structure joining date will be having its elements as day month and year. The c++ compiler automatically allocates sufficient memory to accommodate all of the element variable that make up a structure variable.

STRUCTURES

Technically ,there is no difference between a structure and a class. In fact, a structure is a class declared with keyword struct and by default, all members are public in a structure whereas all members are private by default in the class.

STRUCTURE

In simple terms structure can be defined as,

It is a collection of variables referenced under one name.

REFERENCING STRUCTURE ELEMENTS

The Structure variable name is followed by a period (.) and the element name references to that individual structure element.

The syntax of accessing structure element is,

structure-name.element-name

REFERENCING STRUCTURE ELEMENTS

The structure members are treated like other variables. Therefore , to print the year of joining date, we can write,

```
cout<<joining_date.year;
```

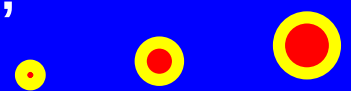
In the same fashion to read the members of the structure we can write,

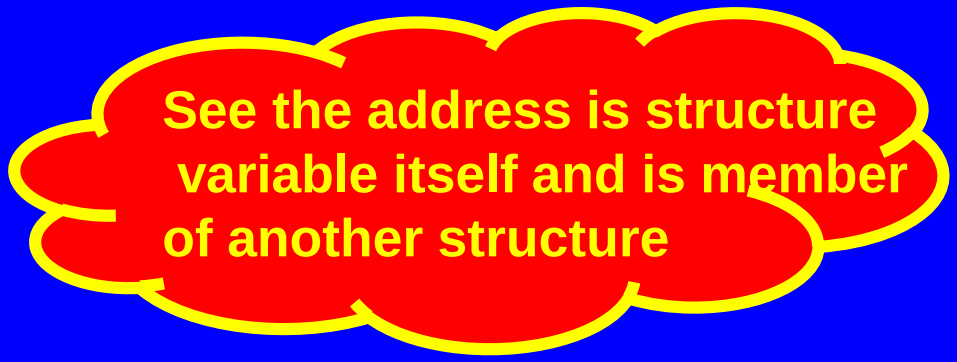
```
cout<<joining_date.day>>joining_date.month;
cout<<joining_date.year;
```

NESTED STRUCTURES

A structure can be nested inside another.
Following code fragment illustrates it;

```
struct addr
{
int houseno;
char area[26];
char city[26];
char state[26];
};
struct emp
{
int empno;
char name[26];
char desig[16];
addr address;
float basic;
}worker;
```





ACCESSING NESTED STRUCTURE MEMBERS

The members of the structures are accessed using dot operator. To access the city member of address structure which is an element of another structure worker we shall write

worker.address.city

To initialize houseno member of address structure, element of worker structure, we can write as follows,

worker.address.houseno=1694

ARRAYS OF STRUCTURE

Since array can contain similar elements the combination having structures within an array is an array of structure

To declare the array of structure, you must first define a structure and then array variable of that type. For instance, to store addresses of 100 members of council, you need to create an array

addr mem_addr[100];

This creates 100 sets of variables that are organised as defined in the structure.

To access we can write,

cout<<mem_addr[7].houseno;

This statement is to print house no position of 8

ARRAYS WITHIN STRUCTURE

A structure can have an array as one of its elements.

For example,

```
struct student
{
int rollno;
char name[20];
float marks[5];
};
student learning;
```

The above declared structure variable learner is of structure type student that contains an element which is an array of 5 floats to store marks of student 5 different subjects

To reference marks of 3rd subject of structure learner, we shall write,

learner.marks[2]

PASSING STRUCTURES TO FUNCTIONS

Structures can be passed to a functions **by value** as well as **by reference** method.

In **Call By Value**, the called function copies the passed structure then remains unaffected.

In **Call By Reference,** the called function declares a reference for the passed structure and refers to the original structure elements through its reference.

PROGRAM

```
#include <iostream.h>
struct emp
{
int empno;
char name[25];
double salary;
};
void reademp(Emp &e)
{
cout <<“Enter Employee No”; cin>>e.empno;
cout<<“Enter the Employee Name”; cin>>e.name;
cout<<“Enter the salary “ ; cin>>e.salary;
}
```

PROGRAM (contd...)

```
void showemp(Emp e)
{
cout<<“Employee Details”;
cout<<“Employee Number<<e.empno”;
cout<<“Name”<<e.name;
cout<<“Salary “<<e.salary;
}
void main()
{
emp e1;
reademp(e1);
showemp(e1);
}
```