Vidyalankar

F.E. Sem. I
CP – I : Notes

Students may refer the solved programs and theory from the book :

TATA MC-GRAW-HILL SERIES FOR MUMBAI UNIVERSITY
(Computer Programming 1)
Author : Kunal Pimparkhede

The chapter wise break up from book is as given below:

Chapter Number and Name	Page Numbers from Book	Weightage in University Exam
1. Introduction	1.1 to 1.20	-
2. Fundamentals of C++	2.1 to 2.31	10 to 15 Marks
3. Operators and Type Casting	3.1 to 3.44	
4. Decision Making Control statements	4.1 to 4.40	10 Marks
5. Iterative Control statements	5.1 to 5.76	20 Marks
6. Arrays	6.1 to 6.78	20 Marks
7. Functions	7.1 to 7.72	10 to 15 Marks
8. Pointers	8.1 to 8.58	20 Marks
9. Structures and Unions	9.1 to 9.57	5 to 10 Marks
10. Dymamic Memory Allocation	10.1 to 10.39	10 Marks
11. Classes and Objects	11.1 to 11.81	20 Marks
12. Constructors and Destructors	12.1 to 12.52	5 to 10 Marks
13. Operator and Function Overloading	13.1 to 13.58	10 Marks
14. Inheritance	14.1 to 14.82	10 to 20 Marks

Functions

Storage classes in C++

The place in the memory where a variable is stored after it is declared in the program is called as a "storage class" of that variable. C++ supports following storage classes for each of the variables declared in the program"

i) **auto** storage class
ii) **register** storage class
iii) **static** storage class
iv) **extern** storage class

The compiler allocates memory to each of the variables declared in the program depending upon the "storage class" specified at the time of declaration of the variables. Given below is the syntax of declaring variables by specifying the "storage class" for the variables.

<StorageClass> <DataType> <VariableName>;

i) auto storage class

This is the default storage class for all the variables in C++. The variables of type "auto" are stored in the main memory (RAM) of the computer system as shown in the figure 1. If an auto variable is defined inside the function, then it is called as a "local auto variable". The "local auto variables" are stored in a part of RAM called as **stack**. The "local auto variables" are created when the control of execution enters the function definition and they will be destroyed after the function execution

completes. The "local auto variables" will not be available outside the scope of the function in which they are defined .The statements below show the creation of "auto" variables "b" and "c" :

```
/*auto is a keyword in C++*/
auto int b =20; // b is a "auto" variable
/*By default all the variables are "auto" in C++*/
int c=30; // c is a "auto" variable
```

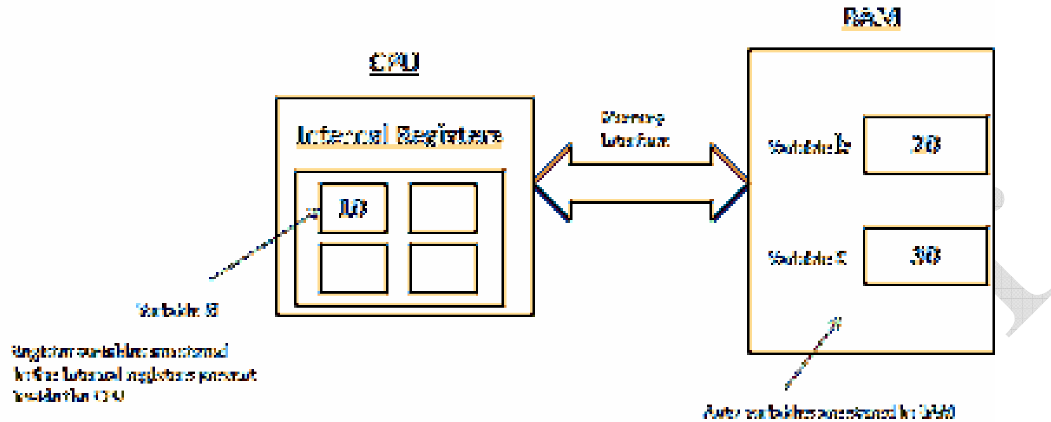

Fig. 1 : Register and auto storage classes

ii) register storage class

"Register" refers to the internal memory integrated inside the chip of the processor. The variables with a storage class specified as "register" are stored in the internal registers of the CPU as shown in the figure, and hence they can be accessed faster when compared to the "auto" variables. As the number of "registers present inside the processor memory are limited, we cannot create large number of register variables. The typical count of "register" variables can be 3 or 4 in a C++ program. C++ supports a keyword "register" to create register type variables. Given below is a C++ statement that creates a integer variable "a" with a register storage class:

```
register int a = 10;
```

iii) static storage class

The static variables are variables whose life span is throughout the C++ program. However, it is important to understand that the static variables can only be accessed within a function or a block in which they are defined. For example, let us consider the function "f1" which defines a static variable "c" with an initial value as zero as shown in the code.

```
#include<iostream.h>
void f1()
{
static int c=0;
c++;
cout<< "Function f1 is called "<<c<< " times"<<endl;
}
void main()
{
f1();
f1();
f1();
f1();
}
```

Output :

Function f1 is called 1 times
Function f1 is called 2 times
Function f1 is called 3 times
Function f1 is called 4 times

As seen from the above program, when the function "f1" is called for the first time the value of the static variable "c" is initialized to zero, indeed the default value of static variables is always zero. As, the static variable "c" is now stored in the global heap space its life time is throughout the C++ program. This means that, the "static" variables will be initialized only once when the function is called for the first time and the variable "c" will not be initialized each time when the function "f1" is called. Rather the value of variable "c" updated by previous function call will now be used by the new function call. Hence the output of the program is as shown below. The output clarifies that, each time we call a function, each of the function runs use the value of variable "c" that was last updated by the most recent execution of the function.

iv) extern storage class

The variable which is defined in some other "cpp" file is called as an external variable. We can access the value of such variables using the keyword "extern" as shown in the figure 2.

As seen from the figure, the variable "a" is declared as a global variable in the file "p1.cpp" with a value initialized as 10. The value of variable "a" can also be accessed in some other file say "p2.cpp" provided that the variable "a" is declared as "extern" in the file "p2.cpp". The keyword informs the compiler that the variable "a" is actually declared in some the file. Hence, we can now access the value of a variable which is defined in some other file using a keyword "extern".

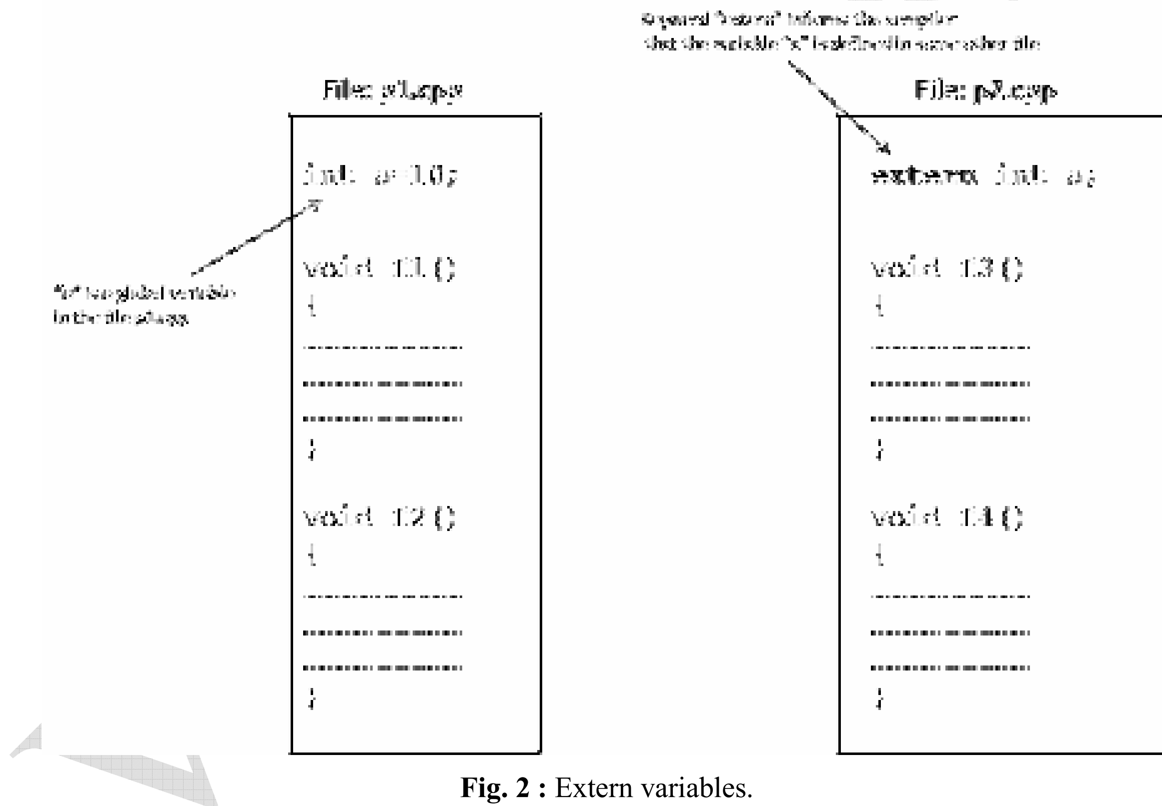

Fig. 2 : Extern variables.

Dynamic Memory Allocation

Linked List

Linked list is a data structure which is created as a collection of nodes such that the new nodes can be added to the linked list or existing nodes can be removed from the linked list at runtime of the program. This means that the number of nodes in the linked list can be changed at runtime as per the requirement of the incoming data. As the linked list is a chain of nodes which can "grow" or "shrink" at runtime of the program, it avoids the drawback with arrays because no maximum bound or capacity is associated with linked list. This is unlike arrays, where the maximum capacity of the array must be given at the time of creating it.

Each node of the linked list consists of following two members:
i) **data**: A integer data value
ii) **next**: A pointer to a next node present in the linked list. The pointer "next" will contain NULL for a last node of the linked list.

Hence a node of the linked list can be defined using a structure below:

```
struct Node
{
int data;
Node *next;
};
```

Given below are the functions to **insert, traverse and find** a specific node in the linked list. The given functions assume that there is a GLOBAL pointer named as "**head**" created such that the pointer "head" always points to the first node of the linked list.

Initially, the linked list contains no elements and hence the GLOBAL pointer "head" points to NULL as shown below:

```
Node *head=NULL;
```

- **Function to insert a node into a Linked List**

```
void insert(int x)
{
    if(head==NULL)
    {
    /*Node to be inserted is the first node*/
    head= new Node;
    head->data=x;
    head->next=NULL;
    }
else
    {
    /*Node to be inserted is the subsequent node*/
    Node *d = new Node;
    d->data=x;
    d->next=NULL;
    Node *temp;
    temp=head;
        while(temp->next!=NULL)
        {
            temp = temp -> next;
        }
        temp->next=d;
    }
}
```

- **Function to traverse a Linked List**

```
void traverse()
{
Node *temp;
temp=head;
cout<<"The elements of the linked list are as below:"<<endl;
/*Traverse the list until it ends*/
    while(temp!=NULL)
    {
        cout<<temp->data<<endl;
```

Vidyalankar

```
                temp = temp -> next;
                }
        }
```

- **Function to find a specific node in an existing Linked List**

```
void find(int x)
{
    Node *temp;
    temp=head;
    /*Traverse the list until the node is not found and end of the list is not reached*/
        while(temp!=NULL && temp->data!=x)
        {
            temp = temp -> next;
        }
        if(temp==NULL)
        {
        /*Node not found*/
        cout<<"Node not found"<<endl;
        }
    else
        {
        /*Node found*/
        cout<<"Element found"<<endl;
    }
}
```

Arrays

Multiplication of 2 matrices

Let the matrix "a" have r1-rows and c1-columns and matrix "b" have r2-rows and c2-columns. Also, we know that the multiplication of two matrices can only be performed if number of columns in the first matrix is same as the number of rows in the second matrix. Hence, we perform the multiplication of the matrices only if the value of c1 is same as r2 as shown in the code. Before starting the multiplication process we take the matrix "a" with r1 rows and c1 columns and matrix "b" with r2 rows and c2 columns as input from the user as seen in the code. Let the matrices "a" and "b" to be multiplied be as below:

	0	1	2
0	10	20	30
1	50	60	70
2	90	100	110

2D array "a" with 3 rows and 3 columns

	0	1	2
0	5	8	3
1	6	2	20
2	12	10	5

2D array "b" with 3 rows and 3 columns

So as to understand the process of multiplication let us evaluate the value Res[0][0] which is the first element of the resultant matrix. The value Res[0][0] will be evaluated as:

Res[0][0] = a[0][0]*b[0][0] + a[0][1]*b[1][0] +a[0][2]*b[2][0];

Substituting, the values from figure we have,

Res[0][0] = 10*5 +20*6 +30*12

Res[0][0] = 5400

Hence for element at ith row and jth column we can evaluate Res[i][j] as:

Res[i][j] = a[i][0]*b[0][j] + a[i][1]*b[1][j] +a[i][2]*b[2][j];

Vidyalankar

Given below is the C++ program to matrix multiplication:

```
#include<iostream.h>
void main()
{
    int a[100][100],b[100][100], Res[100][100];
    int r1,c1,r2,c2,i,j,k;
    cout<<" Enter the order of the matrix a";
    cin>>r1>>c1;
    cout<<" Enter the order of the matrix b";
    cin>>r2>>c2;
    if(c1==r2)
    {
        cout<<"Enter Matrix a"<<endl;
        for(i=0;i<=r1-1;i++)
        {
            for(j=0;j<=c1-1;j++)
            {
                cin>>a[i][j];
            }
        }
        cout<<"Enter Matrix b"<<endl;
        for(i=0;i<=r2-1;i++)
        {
            for(j=0;j<=c2-1;j++)
            {
                cin>>b[i][j];
            }
        }
        //perform multiplication
        cout<<"Multiplication is"<<endl;
        for(i=0;i<=r1-1;i++)
        {
            for(j=0;j<=c2-1;j++)
            {
                Res[i][j]=0;
                for(k=0;k<=c1-1;k++)
                {
                    Res[i][j]+=a[i][k]*b[k][j];
                }
                cout<<Res[i][j]<<"   ";
            }
            cout<<endl;
        }
    }
else
    {
        cout<<"Multiplication is not possible";
    }
}//end of main
```