Input output statements

Input output statements

- The most important input output statements are:

  cin, cout instructions

For output we use cout and the insertion operator << as in the following syntax

Syntax:

cout<<output list;

Ex:

cout<<“my name is adel”;

cout<<x<<y;

cout<<“x=”<<x;

Input output statements

For input we use cin and the extraction operator >> as in the following syntax

Syntax:

cin>>input list;

Ex:

cin>>x>>y;

cin>>a;

example

Ex1) Write a program to find the value of y where

Y=x+3b-c

solution

```
#include <iostream>
void main ()
{
  int x,b,c,y;
cout << "Give the value of x " ;
cin>>x;
cout << "Give the value of b " ;
cin>>b;
cout << "Give the value of c " ;
cin>>c;
y=x+3*b-c;

cout << "the value of y= "<<y ;

}
```

example

Ex2) Write a program to find the value of y where

Y=a-3b+c

C=3x+7

solution

```
#include <iostream>
void main ()
{
  int x,b,c,y,a;
cout << "Give the value of x " ;
cin>>x;
c=3*x+7;
cout << "Give the value of b " ;
cin>>b;
cout << "Give the value of a " ;
cin>>a;
y=a-3*b+c;

cout << "the value of y= "<<y ;

}
```

C - IF...ELSE STATEMENT

An **if** statement can be followed by an optional **else** statement, which executes when the boolean expression is false.

Syntax:

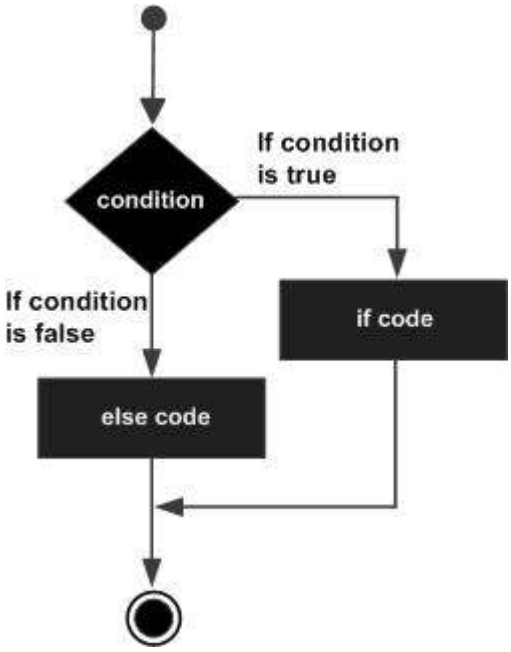
The syntax of an **if...else** statement in C programming language is:

```
if(boolean_expression)
{
   /* statement(s) will execute if the boolean expression is true */
}
else
{
  /* statement(s) will execute if the boolean expression is false */
}
```

If the boolean expression evaluates to **true**, then the **if block** of code will be executed, otherwise **else block** of code will be executed.

C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.

Flow Diagram:



Example:

```
#include <stdio.h>

int main ()
{
   /* local variable definition */
   int a = 100;

   /* check the boolean condition */
   if( a < 20 )
   {
       /* if condition is true then print the following */
       printf("a is less than 20\n" );
   }
```

```
   else
   {
       /* if condition is false then print the following */
       printf("a is not less than 20\n" );
   }
   printf("value of a is : %d\n", a);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
a is not less than 20;
value of a is : 100
```

The if...else if...else Statement

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.

When using if , else if , else statements there are few points to keep in mind:

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax:

The syntax of an **if...else if...else** statement in C programming language is:

```
if(boolean_expression 1)
{
   /* Executes when the boolean expression 1 is true */
}
else if( boolean_expression 2)
{
   /* Executes when the boolean expression 2 is true */
}
else if( boolean_expression 3)
{
   /* Executes when the boolean expression 3 is true */
}
else
{
   /* executes when the none of the above condition is true */
}
```

Example:

```
#include <stdio.h>

int main ()
{
   /* local variable definition */
   int a = 100;

   /* check the boolean condition */
   if( a == 10 )
   {
       /* if condition is true then print the following */
       printf("Value of a is 10\n" );
   }
   else if( a == 20 )
   {
       /* if else if condition is true */
```

```
        printf("Value of a is 20\n" );
    }
    else if( a == 30 )
    {
        /* if else if condition is true  */
        printf("Value of a is 30\n" );
    }
    else
    {
        /* if none of the conditions is true */
        printf("None of the values is matching\n" );
    }
    printf("Exact value of a is: %d\n", a );

    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
None of the values is matching
Exact value of a is: 100
```

Loops

```
# include<iostream.h>
void main()
{int x,m=1,i;
for (i=0;i<=10;i++)
{ cin>>x;
   If (x<0)
          if(m==1) m=x else if (x>m) m=x;
}
  cout<<m;
}
```

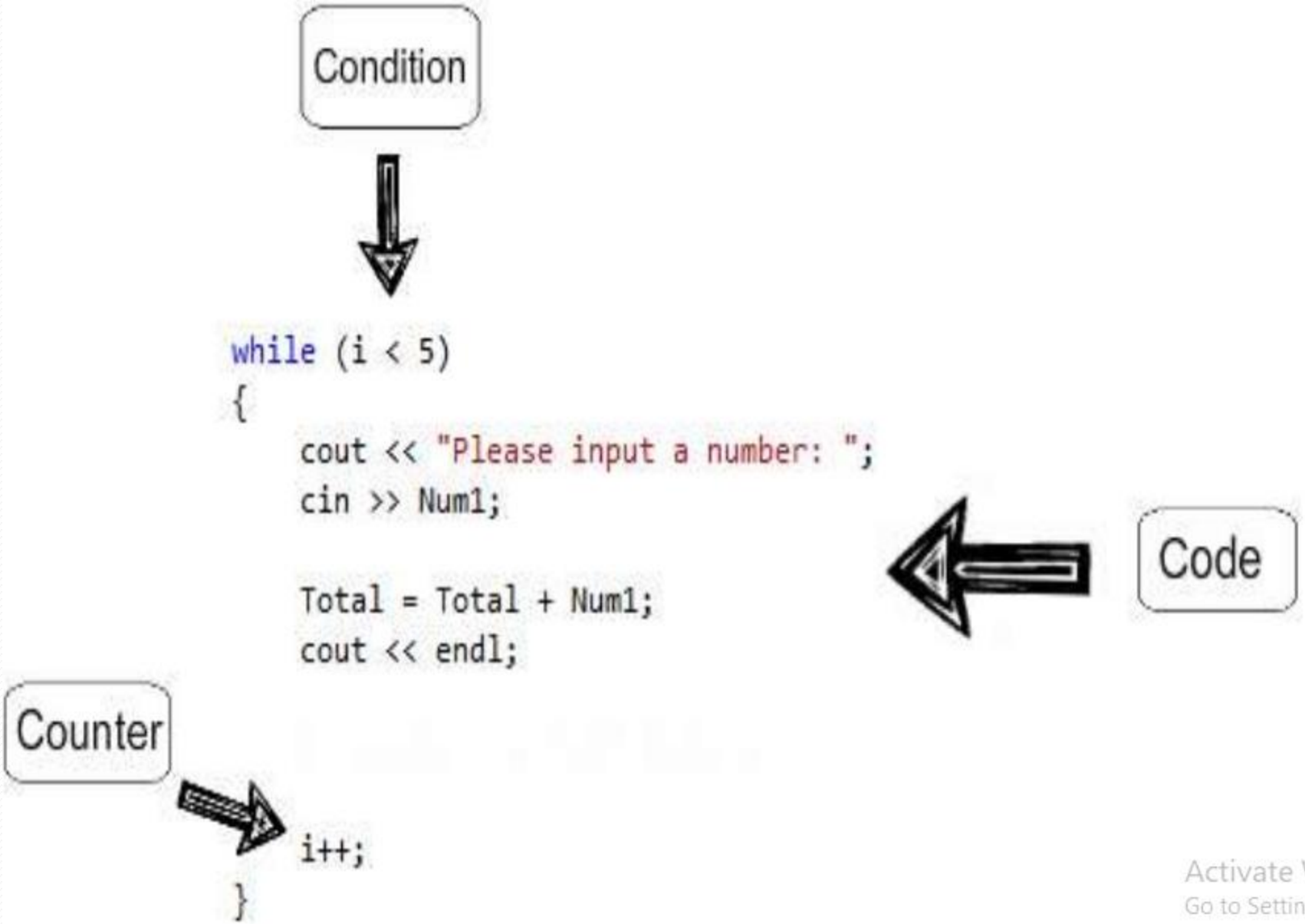
WHILE LOOP

- The syntax of **while** statement :

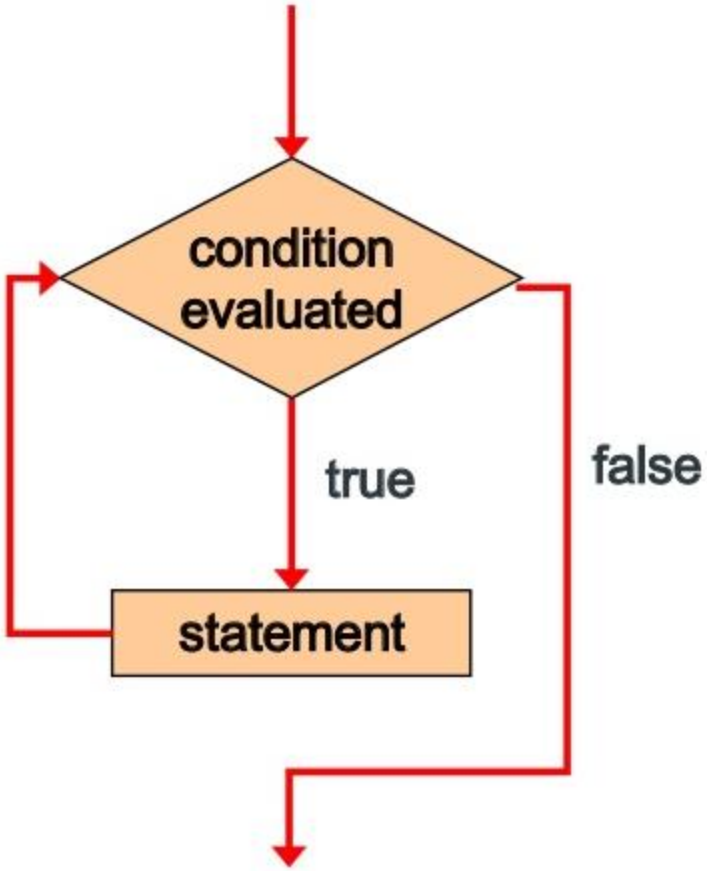
  while (**loop repetition condition**)

  statement

- **Loop repetition condition** is the condition which controls the loop.
- The ***statement*** is repeated as long as the loop repetition condition is **true**.
- A loop is called an **infinite loop** if the loop repetition condition is always true.


Condition
while (i < 5)
{
cout << "Please input a number: ";
cin >> Num1;
Total = Total + Num1;
cout << endl;
Code
Counter
i++;
}

Logic of a while Loop



Ex) Write a program to find the summation of group of numbers where the last number is 7

```
# include<iostream.h>
void main()
{int x,m;
 cin>>x;   m=x;
while(x!=7)
{ cin>>x;
   m+=x;
}
  cout<<m;
}
```

DO…WHILE LOOP

- The syntax of **do-while** statement in C:

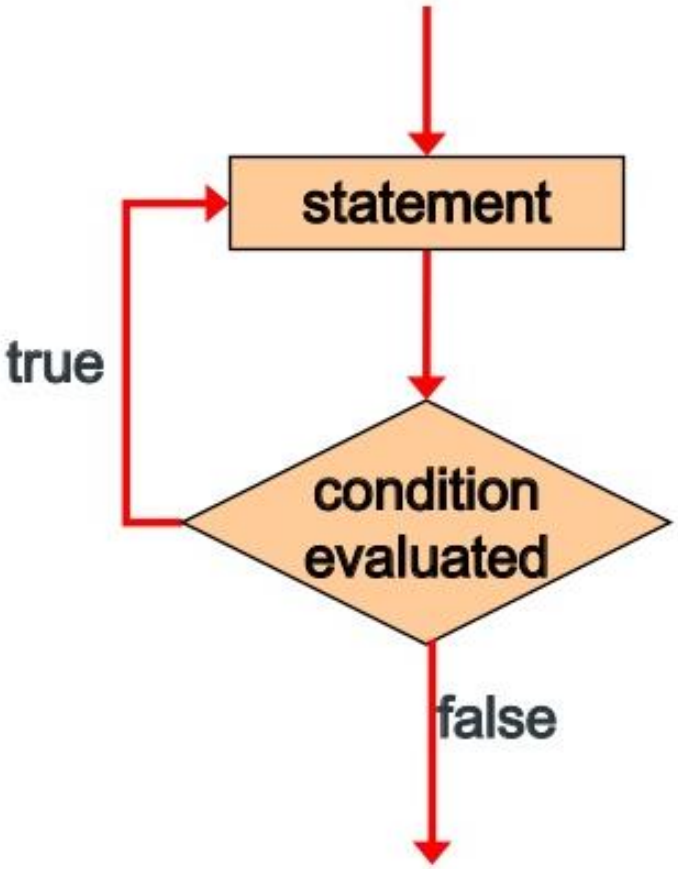
  do

  statement

  while (**loop repetition condition**);

- The ***statement*** is first executed.
- If the **loop repetition condition** is true, the ***statement*** is repeated.
- Otherwise, the loop is exited.

Logic of a do…while loop



Ex) Write a program to find how many nmber can be divided by 3 between 15 numbers

```
# include<iostream.h>
void main()
{int x,i=0,j=0;
do
i++;
  cin>>x;
  if (x%3==0)j++;
while(i<15)
  cout<<j;
}
```

Another solution to find the largest negative number between ten numbers

```
# include<iostream.h>
void main()
{int x,i=0,m;
do
  i++;  cin>>x;
  if (x<0)m=x;
while((x>0)&&(i<10))

while(i<10)
{i++;
  cin>>x;
  if(x<0) if (x>m)m=x;
}
  cout<<m;
}
```

Thank you

Arrays

Bubble sort

How to sort one dimension array ?

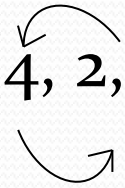
One of the most important algorthems to sort one dimension arrays is (bubble sort) which can be illustrated as in bellow :

Suppose that we have the following element:

[4, 2, 3, 6, 1]

And we want to sort it in ascending order to be

[1, 2, 3, 4, 6]



[4, 2, 3, 6, 1]

[2, 4, 3, 6, 1] [2, 3, 4, 1] [2, 3, 1] [1, 2]

[2, 3, 4, 6, 1] [2, 3, 4, 1] [2, 1, 3]

[2, 3, 4, 6, 1] [2, 3, 1, 4]

[2, 3, 4, 1, 6]

[1, 2, 3, 4, 6]

Bubble sort

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
int a[6],i,j,x,n;
Cout<<“how many element:”;
Cin>>n;
Cout<<“now give your elements: “;
for(i=0;i<n;i++)
Cin>>a[i];
for(i=0;i<n-1;i++)
for(j=0;j<n-i-1;j++)
if(a[j]>a[j+1])
{x=a[j]; a[j]=a[j+1];a[j+1]=x;
}
for(i=0;i<n;i++)
Cout<<a[i]<<‘\n’;
getch();
}
```

Two dimension array examples (1)

a[0,0]=2	a[0,1]=5	a[0,2]=7
a[1,0]=3	a[1,1]=0	a[1,2]=1
a[2,0]=7	a[2,1]=8	a[2,2]=2

Suppose the following

n=number of rows or columns

i=row sequence, j=column sequence

Then we can conclude the following

Location	condition
Primary diagonal	i=j
Upper Primary diagonal	i<j
Secondary diagonal	i+j=n-1
Under Secondary diagonal	i+j>n-1

The solution

Ex1) Write a program to read two dimension array then print it as rows and columns and find its summation

```
#include<iostream.h>
void main()
{int a[3][3],i,j,s=0;
for(i=0;i<3;i++)
for(j=0;j<3;j++)
cin>>a[i][j];

for(i=0;i<3;i++)
for(j=0;j<3;j++)
s+=a[i][j];

for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
  cout<<a[i][j]<<"   ";
  cout<<'\n';
}
cout<<'\n'<<s;
}
```

The solution

Ex2) Write a program to find the summation of the primary diagonal in two dimension array

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
int a[3][3],i,j,s=0;

for(i=0;i<3;i++)
for(j=0;j<3;j++)
cin>>a[i][j];

for(i=0;i<3;i++)
 s+=a[i][i];

cout<<"summation= "<<s;
getch();
}
```

Ex4) Write a program to find the multiplication of the following

1-first row
2- second column
3- second half of the array rows

The solution

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
int a[4][4],i,j,m1=1,m2=1,m3=1;

for(i=0;i<4;i++)
for(j=0;j<4;j++)
cin>>a[i][j];

for(i=0;i<4;i++)
for(j=0;j<4;j++)
{
if (i==0)m1*=a[i][j];
if (j==1)m2*=a[i][j];
if (i>1)m3*=a[i][j];

 }
```

```
for(i=0;i<4;i++)
{
for(j=0;j<4;j++)
  cout<<a[i][j]<<"  ";

  cout<<'\n';
}
cout<<"first row="<<m1<<'\n';
cout<<"second column="<<m2<<'\n';
cout<<"second half="<<m3<<'\n';

getch();

}
```

Ex3) Write a program to find the summation of the following

1-primary diagonal
2- secondary diagonal
3- the elements upper the primary diagonal
4-the elements under the secondary diagonal

The solution

```
#include<iostream.h>
#include<conio.h>

void main()
{
clrscr();
int a[3][3],i,j,s1=0,s2=0,s3=0,s4=0;

for(i=0;i<3;i++)
for(j=0;j<3;j++)
cin>>a[i][j];

for(i=0;i<3;i++)
for(j=0;j<3;j++)
{
if (i==j)s1+=a[i][j];
if (i+j==2)s2+=a[i][j];
if (i<j)s3+=a[i][j];
if (i+j>2)s4+=a[i][j];
 }
```

```
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
  cout<<a[i][j]<<"  ";

  cout<<'\n';
}
cout<<"primary diagonal="<<s1<<'\n';
cout<<"secondary diagonal="<<s2<<'\n';
cout<<"upper primary diagonal="<<s3<<'\n';
cout<<"under secondary diagonal="<<s4;
getch();
}
```

Two dimension Array examples (2)

Ex1) Write a program to create the following two dimension array

1	1	1
2	2	2
3	3	3

```
# include <iostream.h>
Void main()
{  int a[3][3] ,i,j;

  for (i=0;i<3;i++)
            for (j=0;i<3;i++)
             a[i][j]=i+1;
for (i=0;i<3;i++)
{
            for (j=0;j<3;j++)
             cout<<a[i][j]<<“    ”;
    Cout<<‘\n’;
}
}
```

Ex2) Write a program to create the following two dimension array

1	2	2
3	1	2
3	3	1

```
# include <iostream.h>
Void main()
{  int a[3][3] ,i,j;

  for (i=0;i<3;i++)
            for (j=0;i<3;i++)
             if (i==j) a[i][j]=1; else if (i<j) a[i][j]=2;else a[i][j]=3;

for (i=0;i<3;i++)
{
            for (j=0;j<3;j++)
             cout<<a[i][j]<<“    ”;
    Cout<<‘\n’;
}
}
```

Ex3) Write a program to create the following two dimension array

1	2	3
4	5	6
7	8	9

```
# include <iostream.h>
Void main()
{  int a[3][3] ,i,j,k=0;

  for (i=0;i<3;i++)
            for (j=0;i<3;i++)
             {k++;
             a[i][j]=k;
            }
for (i=0;i<3;i++)
{
            for (j=0;j<3;j++)
             cout<<a[i][j]<<“    ”;
    Cout<<‘\n’;
}
}
```

Ex4) Write a program to create the following two dimension array

1	4	6
8	1	12
14	16	1

```
# include <iostream.h>
Void main()
{  int a[3][3] ,i,j,k=0;

  for (i=0;i<3;i++)
            for (j=0;i<3;i++)
             {k+=2;
             if (i==j) a[i][j]=1; else a[i][j]=k;
            }
for (i=0;i<3;i++)
{
            for (j=0;j<3;j++)
             cout<<a[i][j]<<“    ”;
    Cout<<‘\n’;
}
}
```

Strings in c++

String as a value:

“abcd”

“a12bc”

“1234760”

“#%^&ERT!@#1234”

How to define a string:

Char s1[10];

Char s2[]="abcd";

Char * S;

All of these definitions are valid

Dealing a string as an array of characters:

```
char s[10];

for(i=0;i<10;i++)
  cin>>s[i];

for(i=0;i<10;i++)
  cout<<s[i];
```

Here we will read the string and print it as (character by character)

Dealing a string as one piece:

```
Char s [8];

cin>>s;

cout<<s;
```

Here we can read a string directly by one statement and so about printing it (without using index)

Using the above two types of dealing together for one string:

```
char s[10];

cin>>s;

for(i=0;i<10;i++)
  cout<<s[i];
```

Here we will read a string in one statement then print it as character by character

More specialized standard functions for string processing

gets(string name) // to read a string

puts(string name) // to print a string

strlen(string name) // to find a string length

Here we need to include libraries like
include<stdio.h>
#include<string.h>

Ex) Write a program to read a string then print it's characters in a separate lines

```
# include<stdio.h>
#include<string.h>
#include<iostream.h>
#include<conio.h>
void main()
{ clrscr();
char s[10];
   int i;
   gets(s);
   for(i=0;i<strlen(s);i++)
      cout<<s[i]<<'\n';
    getch();
 }
```

Ex) Write a program to read a string then print it's words in a separate lines

```
# include<stdio.h>
#include<string.h>
#include<iostream.h>
#include<conio.h>
void main()
{ clrscr();
char s[10];
  int i;
  gets(s);
  for(i=0;i<strlen(s);i++)
     if (s[i]==' ') cout<<'\n';
          else cout<<s[i];
   getch();
 }
```

More specialized standard functions for string processing

strcpy(s1,s2) // to copy s2 into s1

strcat(s1,s2) // return the concatination of s1 and s2strings

strcmp(s1,s2) // to compare string and return a value as follows :
‘value=0’ means strings are equal
‘value<0’ means s1<s2
‘value>0’ means s1>s2

More specialized standard functions for string processing

strchr(s,ch) // Returns a pointer to the first occurrence of character ch in string s

strstr(s1,s2) // Returns a pointer to the first occurrence of string s2 in string s1.

Structure in C++

What is structure:

C/C++ arrays allow you to define variables that combine several data items of the same kind but **structure** is another user defined data type which allows you to combine data items of different kinds.

Structures are used to represent a record, suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

- Title
- Author
- Subject
- Book ID

Defining a Structure:

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member, for your program. The format of the struct statement is this:

```
struct [structure tag]
{
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional.

How to declare the Book structure:

```
struct Books
{
    char   title[50];
    char   author[50];
    char   subject[100];
    int    book_id;
}book;
```

Another way to declare structures:

```
struct rec
{ char name[20];
int dgree1,degree2;
};
void main()
{ rec student1, student2;
….
….
```

How to access the structure field:

We have to write the struture name then dot then the field name as followes

```
struct rec
{ char name[20];
int dgree1,degree2;
};
void main()
{ rec student1, student2;
student1.degree1=80;
cin>>student1.name;
……
…..
```

To make structure more useful:

We can define an array where each element of it is a structure therefore we can save an information like a table of data base , suppose that we have the following table and we want to save it's information:

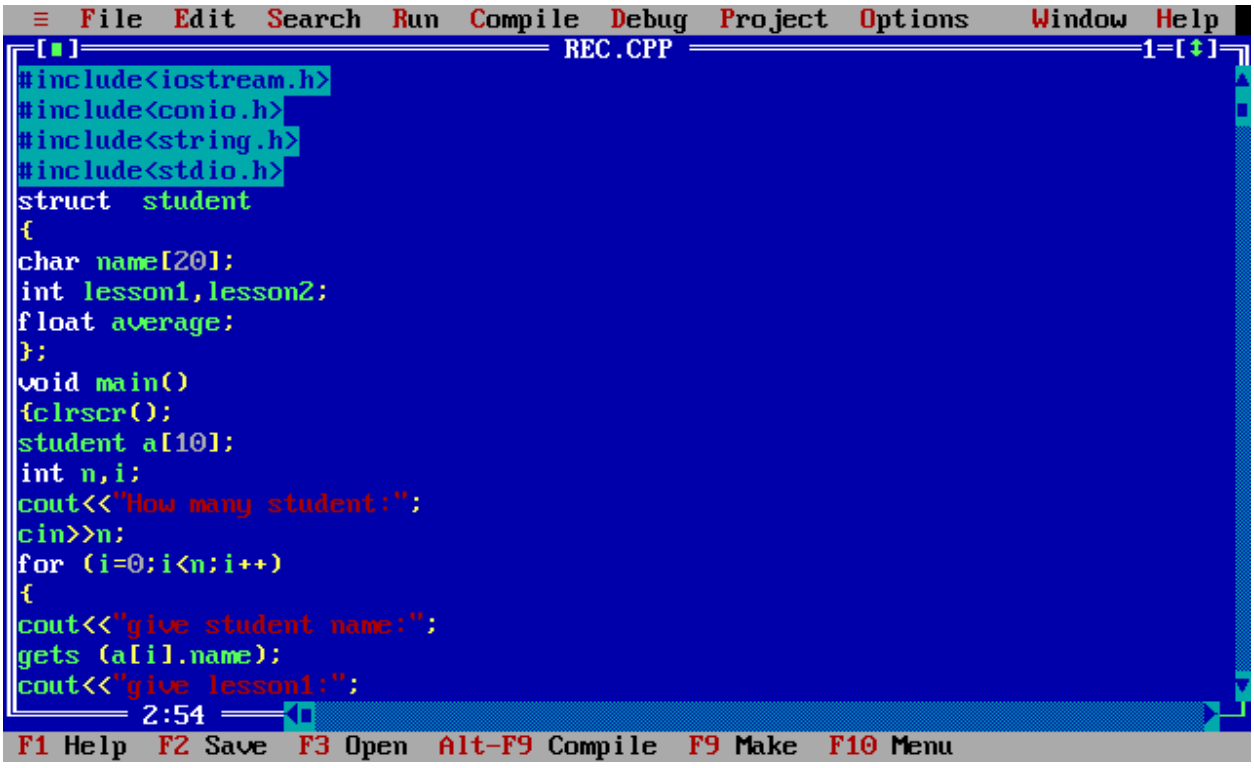
St_name	Lesson 1	Lesson 2	Average
Ahmed	79	81	80
Ali	60	80	70

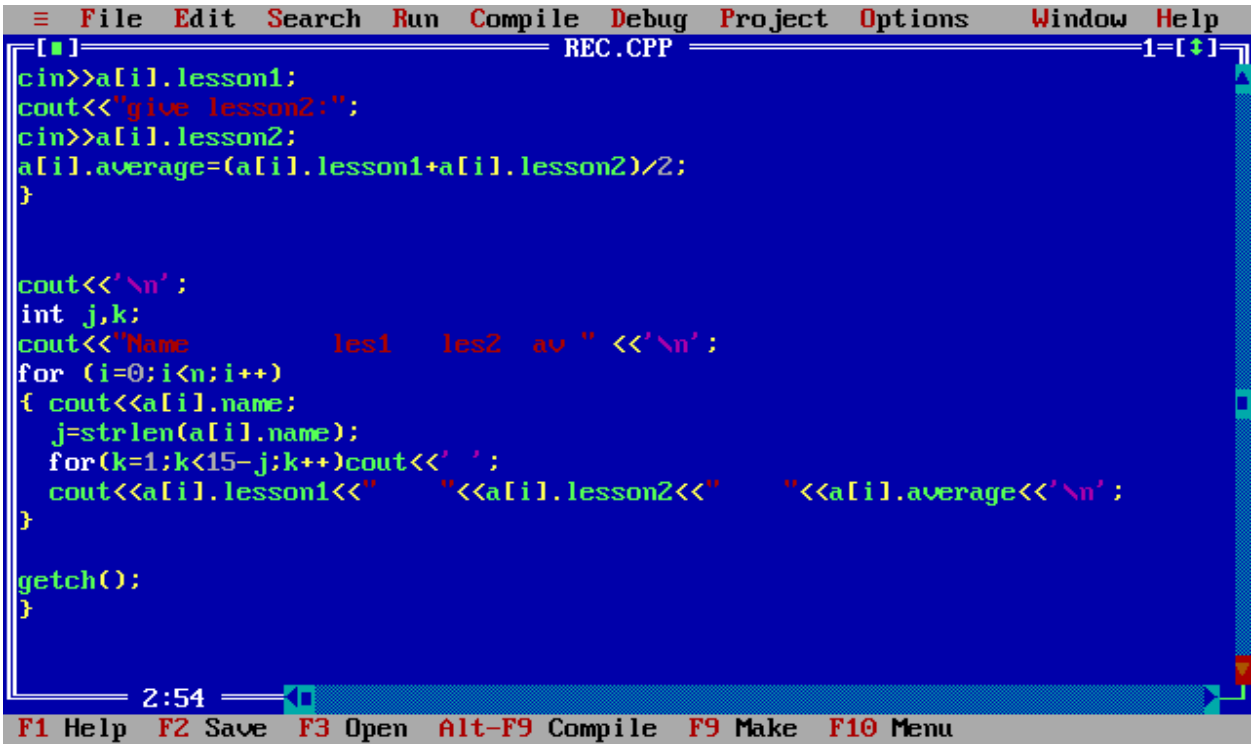
Another way to declare structures:

```
struct rec
{ char name[20];
int dgree1,degree2;
};
void main()
{ rec student1, student2;
….
….
```

Ex)Write a program to save an information about student like in the table below and find the average for each student

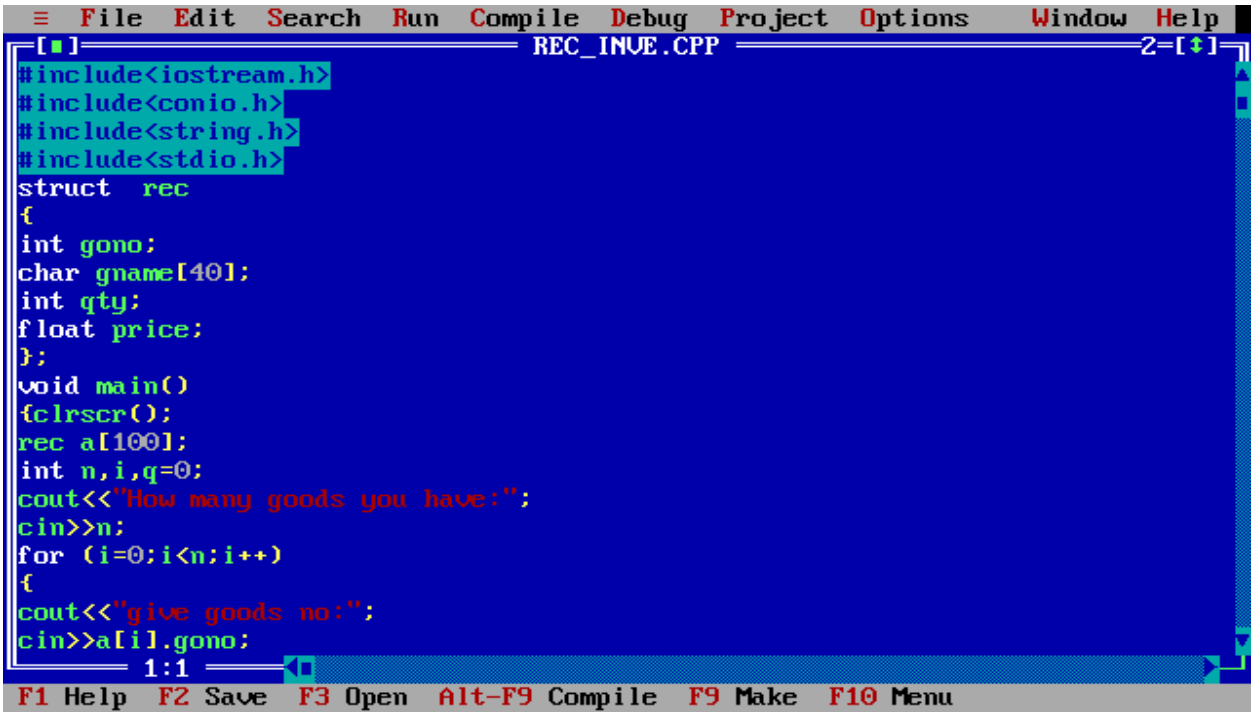
Name	Lesson1	Lesson2	Average

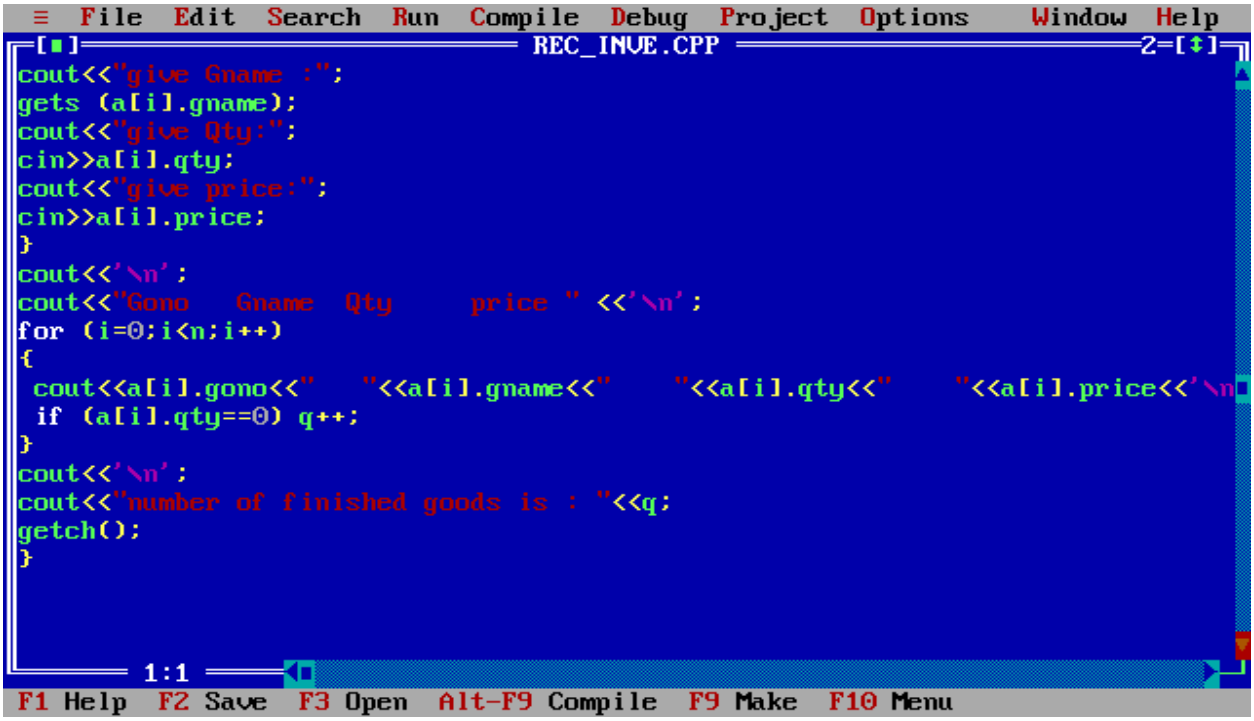




Ex) Write a program to save an information about inventory system like in the table below and find the number of finished goods

Gono	Goname	Qty	price





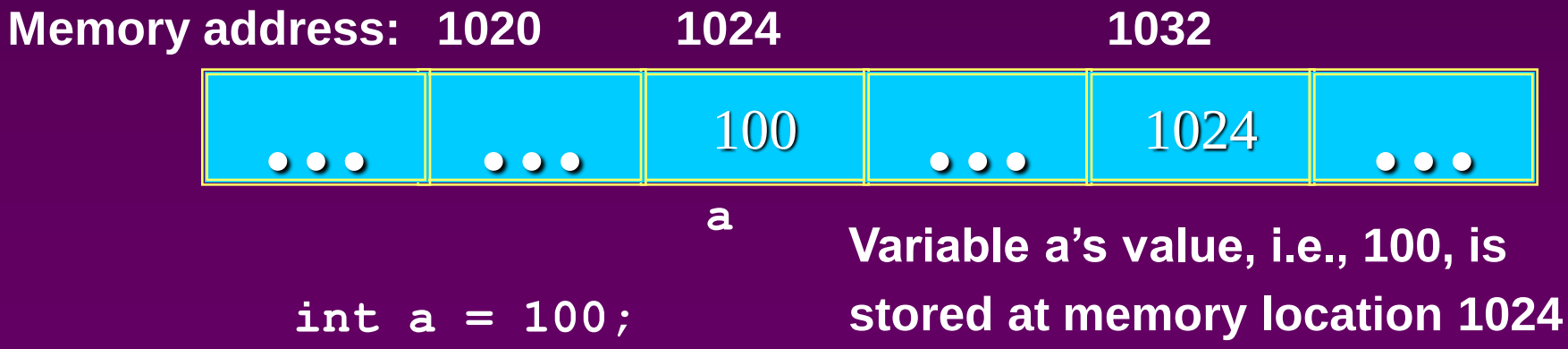
Pointers

Topics

- Pointers
  - Memory addresses
  - Declaration
  - Dereferencing a pointer
  - Pointers to pointer

Computer Memory

- Each variable is assigned a memory slot (the size depends on the data type) and the variable's data is stored there

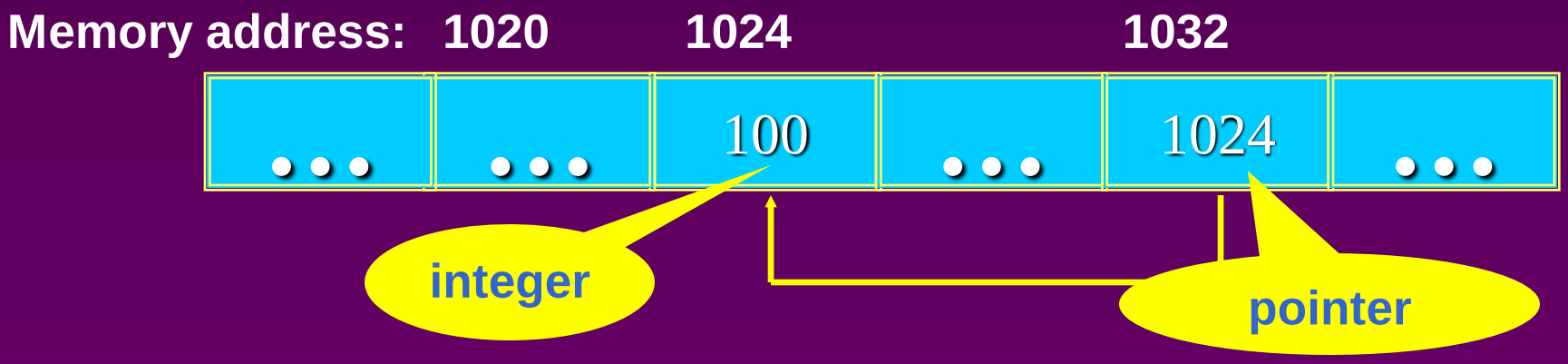



```
int a = 100;
```

Variable a's value, i.e., 100, is stored at memory location 1024

Pointers

- A pointer is a variable used to store the address of a memory cell.
- We can use the pointer to reference this memory cell



Pointer Types

- Pointer
  - C++ has pointer types for each type of object
    - Pointers to `int` objects
    - Pointers to `char` objects
    - Pointers to user-defined objects
      (e.g., `RationalNumber`)
  - Even pointers to pointers
    - Pointers to pointers to `int` objects

Pointer Variable

- Declaration of Pointer variables

```
type* pointer_name;
//or
type *pointer_name;
```

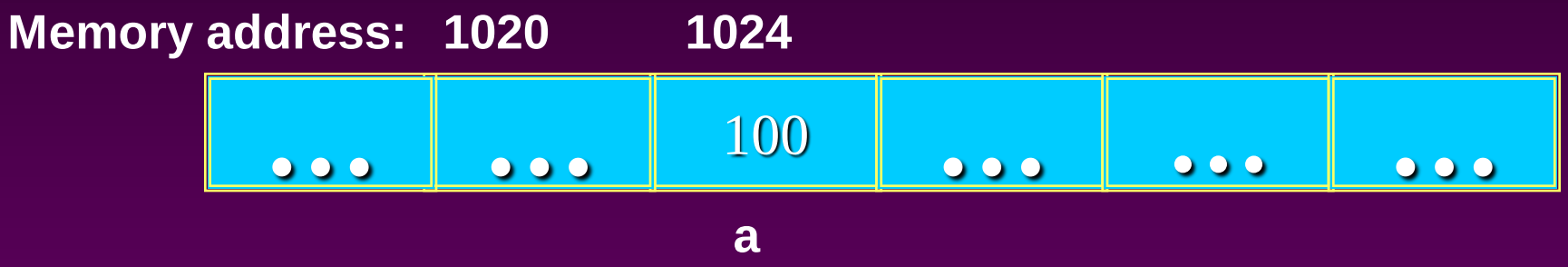
where *type* is the type of data pointed to (e.g. int, char, double)

Examples:

```
int *n;
RationalNumber *r;
int **p;    // pointer to pointer
```

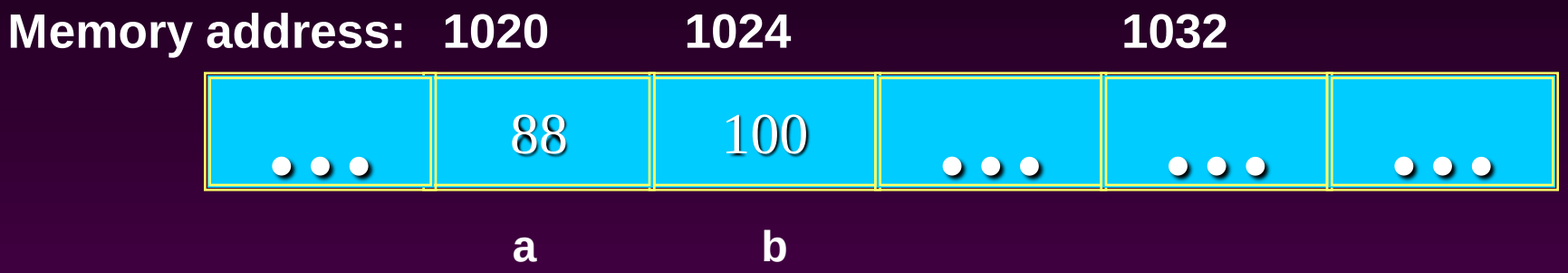
Address Operator &

- *The "address of" operator* (&) gives the memory address of the variable
  - **Usage:** `&variable_name`




```
int a = 100;
//get the value,
cout << a;     //prints 100
//get the memory address
cout << &a;    //prints 1024
```

Address Operator &



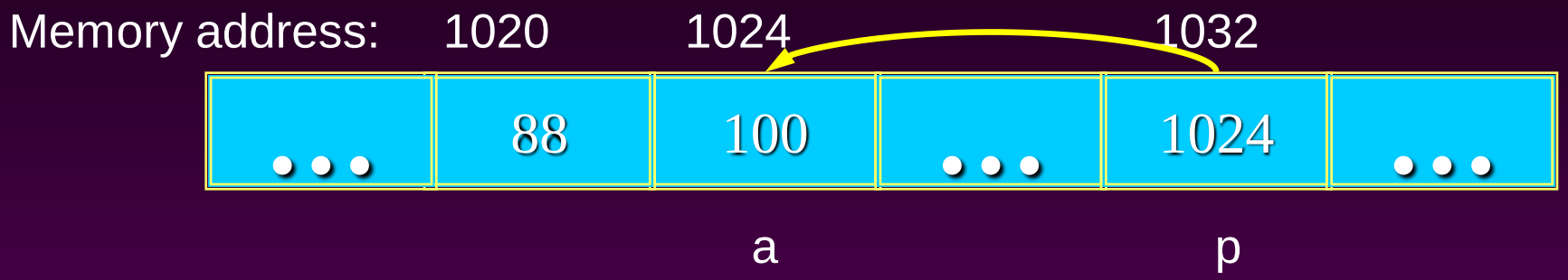

```
#include <iostream>
void main(){
   int a, b;
   a = 88;
   b = 100;
   cout << "The address of a is: " << &a << endl;
   cout << "The address of b is: " << &b << endl;
}
```

- Result is:

The address of a is: 1020

The address of b is: 1024

Pointer Variables




```
int a = 100;
int *p = &a;
cout << a << " " << &a <<endl;
cout << p << " " << &p <<endl;
```

- Result is:

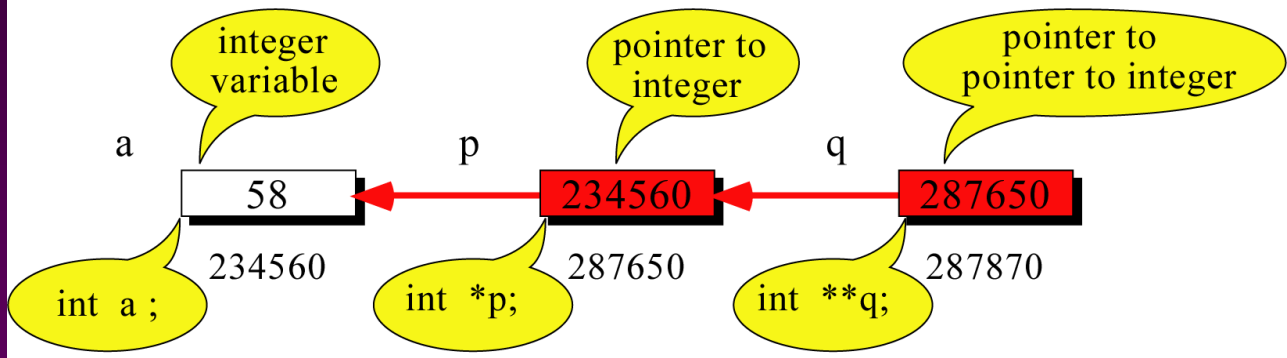
100 1024

1024 1032

- The value of pointer `p` is the address of variable `a`
- A pointer is also a variable, so it has its own memory address

Pointer to Pointer

```
// Local Declarations
int         a ;
int         *p ;
int         **q ;
```



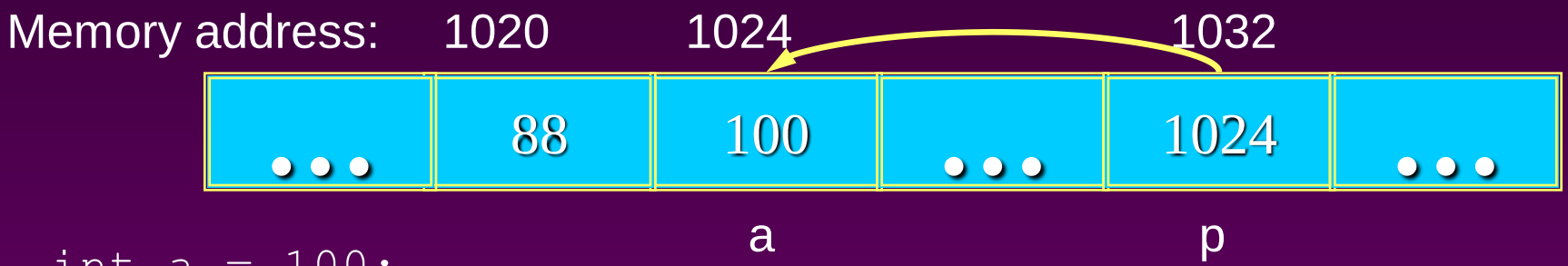
What is the output?

58 58 58

```
// Statements
a = 58 ;
p = &a ;
q = &p ;
cout <<       a << " ";
cout <<      *p << " ";
cout <<     **q << " ";
```

Dereferencing Operator *

- We can access to the value stored in the variable pointed to by using the dereferencing operator (*),




```
int a = 100;
int *p = &a;
cout << a << endl;
cout << &a << endl;
cout << p << " " << *p << endl;
cout << &p << endl;
```

- Result is:

100
1024
1024 100
1032

Don’t get confused

- Declaring a pointer means only that it is a pointer: `int *p;`
- Don’t be confused with the dereferencing operator, which is also written with an asterisk (*). They are simply two different tasks represented with the same sign

```
int a = 100, b = 88, c = 8;
int *p1 = &a, *p2, *p3 = &c;
p2 = &b;     // p2 points to b
p2 = p1;     // p2 points to a
b = *p3;     //assign c to b
*p2 = *p3;   //assign c to a
cout << a << b << c;
```

- **Result is: 888**

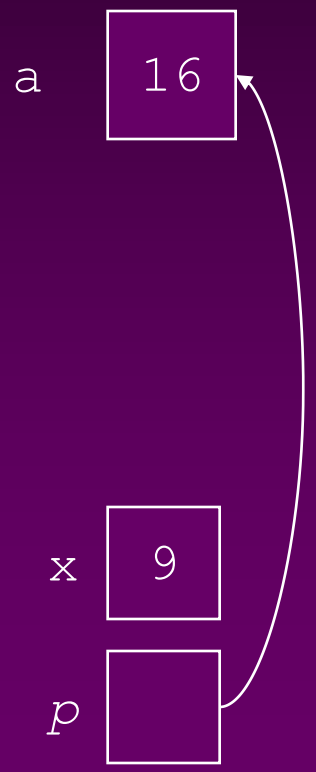
A Pointer Example

The code

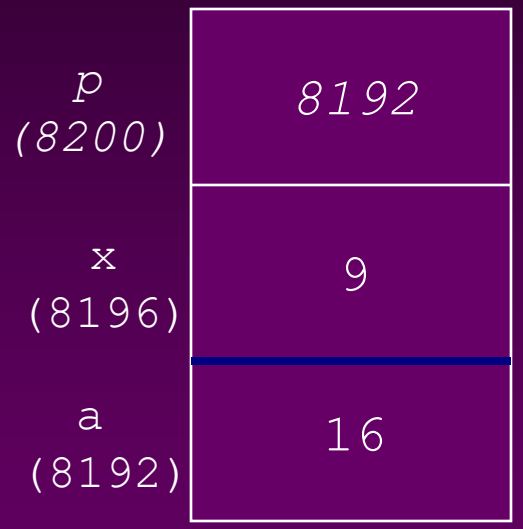
```
void ()
{
   int a = 16;
    int x = 9;
    int *p = &a;
    *p = 2 * x;
   }
```

a gets 18

Box diagram



Memory Layout



Another Pointer Example

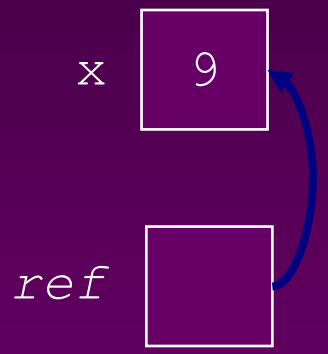
- **Let's figure out: value1==? / value2==? Also, p1=? p2=?**

```
#include <iostream>
void main (){
   int value1 = 5, value2 = 15;
   int *p1, *p2;
   p1 = &value1; // p1 = address of value1
   p2 = &value2; // p2 = address of value2
   *p1 = 10;     // value pointed to by p1=10
   *p2 = *p1;    // value pointed to by p2= value
                 // pointed to by p1
   p1 = p2;      // p1 = p2 (pointer value copied)
   *p1 = 20;     // value pointed to by p1 = 20
   cout << "value1==" << value1 << "/ value2==" <<
   value2;


}
```

Reference Variables

A reference is an additional name to an existing memory location

Pointer:



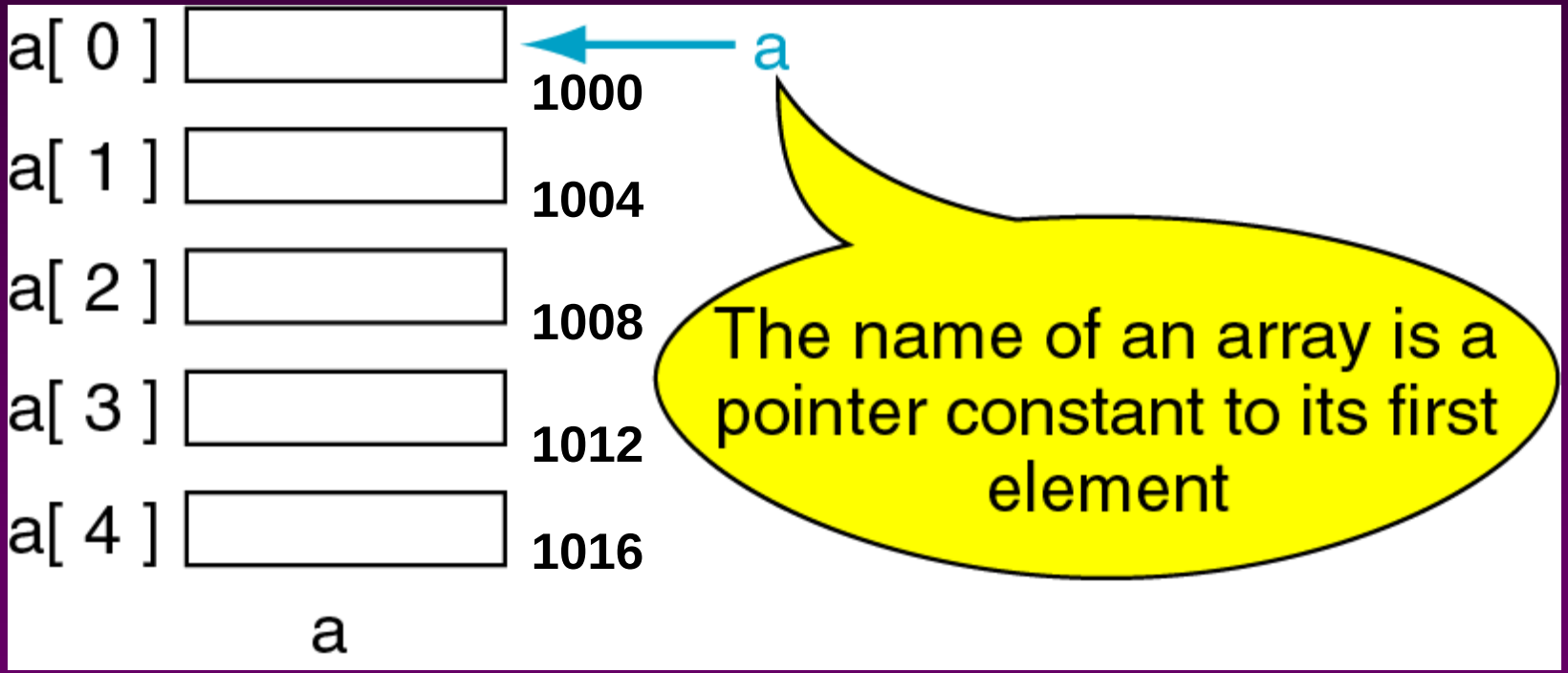

```
int x=9;
int *ref;
ref = &x;
```

Reference:


```
int x = 9;
int &ref = x;
```

Pointers and Arrays

- The name of an array points only to the first element not the whole array.



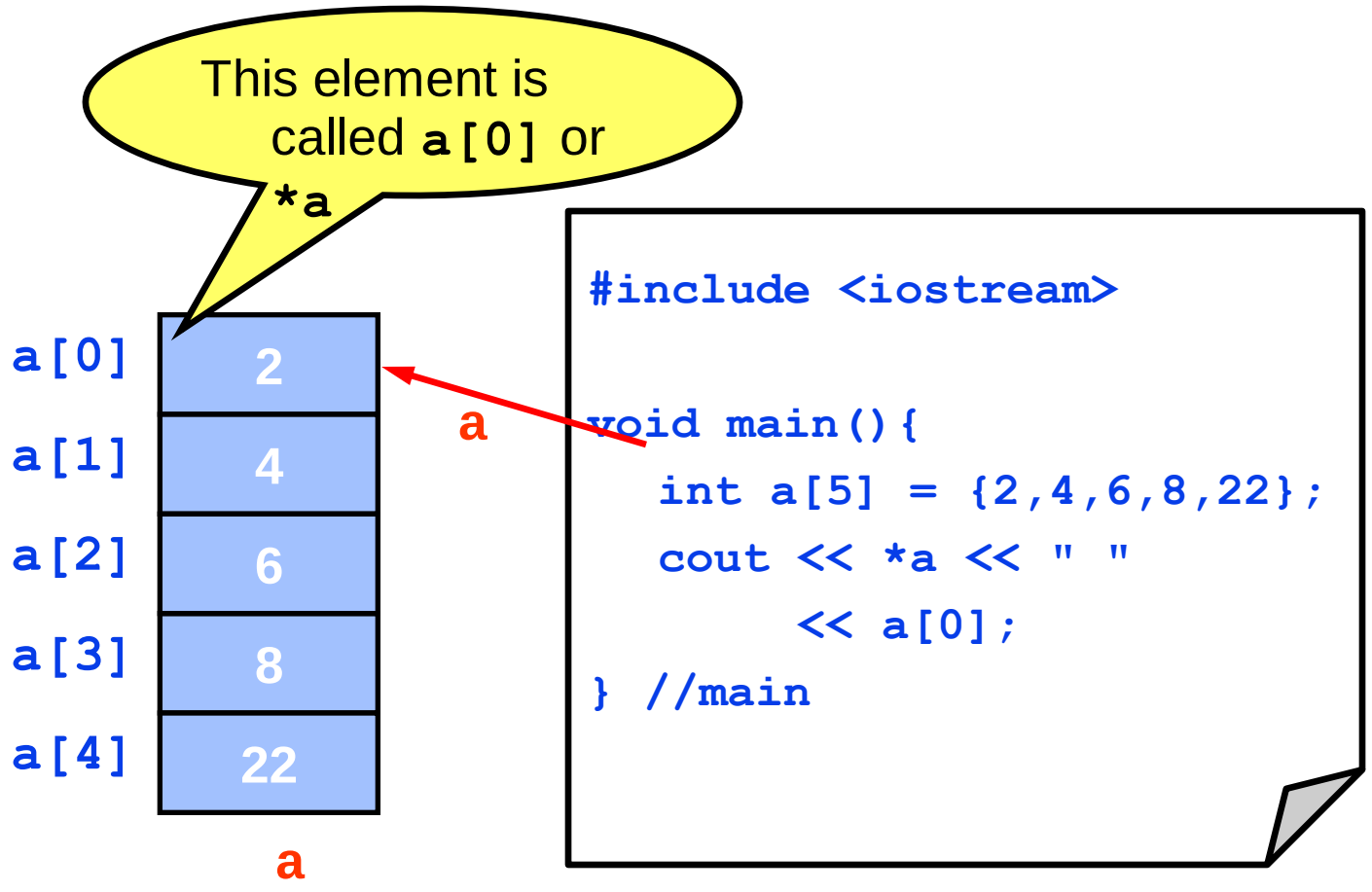
Array Name is a pointer constant

```
#include <iostream>

void main (){
    int a[5];
    cout << "Address of a[0]: " << &a[0] << endl
         << "Name as pointer: " << a << endl;
}
```

Result:

```
Address of a[0]: 0x0065FDE4
Name as pointer: 0x0065FDE4
```

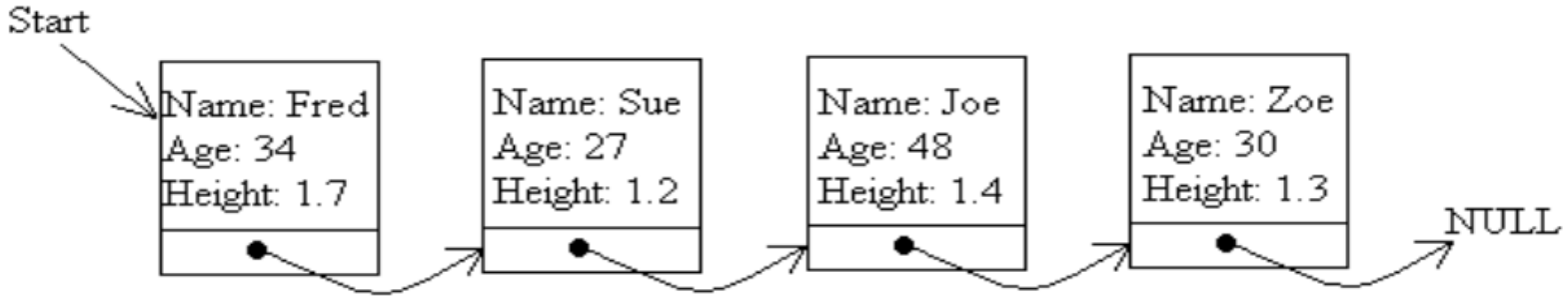
Dereferencing An Array Name



Linked list

What is a linked list?

A linked list is a data structure which is built from structures and pointers. It forms a chain of "nodes" with pointers representing the links of the chain and holding the entire thing together. A linked list can be represented by a diagram like this one:



Defining the data structure for a linked list:

The key part of a linked list is a structure, which holds the data for each node (the name, address , age or whatever for the items in the list), and, most importantly, a pointer to the next node.

```
struct Node
{
        char name[20];
        int age;
        float height;
        Node *next;
};
Node *start_ptr;
```

The use of operators ***New*** and ***Delete*** :

When we create a linked list we should to use the operator "new" to initialize and allocate a pointer to an empty node then we can fill its fields with data, so the more important field is that of type pointer which used to connect the node with the next node or with the final NULL value as in bellow .

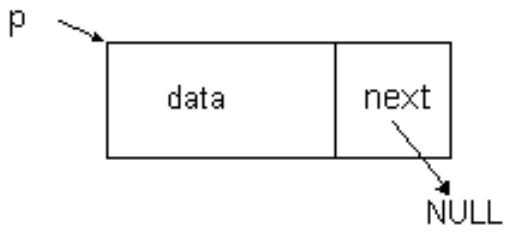
```
node *p;
p=new node;
```

And "delete" is used to erase any pointer to any node and written like in bellow:

```
delete p;
```

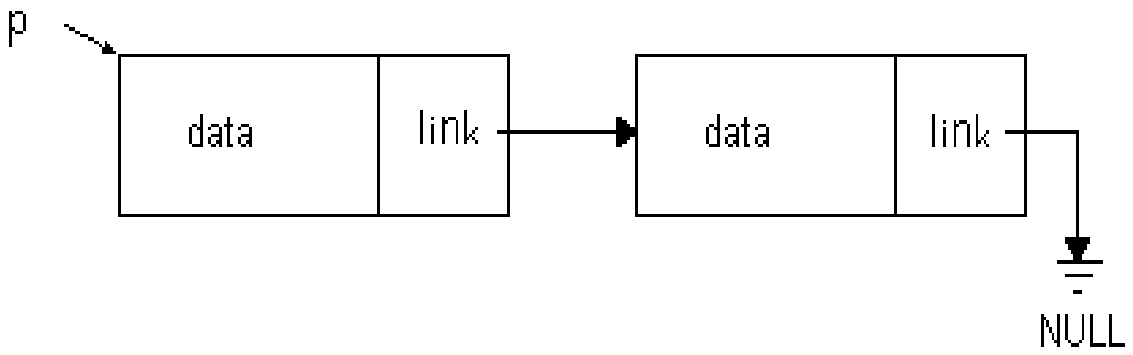
Ex)Write a program to create a linked list with one node

```
#include< iostream.h>
struct node
{
int data;
node* next;
};

void main()
{
   node *p;
   p=new node;
   cin>>p->data;
   p->next =NULL;
}
```



Ex)Write a program to create a linked list with two nodes

```
#include<iostream.h>
struct node
{
int data;
node* link;
};
void main()
{
   node *p,*q;

   p=new node();
   cin>>p->data;
   q=new node();
   cin>>q->data;
   q->link=NULL;
   p->link=q;
}
```



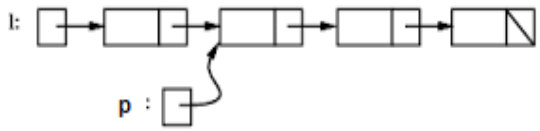
Ex)Write a program to create a linked list with n nodes

```
#include<iostream.h>
struct node
{
int data;
node* link;
};
void main()
{ int n;
   node *p,*q,*s;
   cin>>n;
   p=new node; s=p;
for (int i=1;i<n;i++)
{ cin>>p->data;
   q=new node;    p->link=q;   p=q;
}
cin>>p->data;
p->link=NULL;
}
```

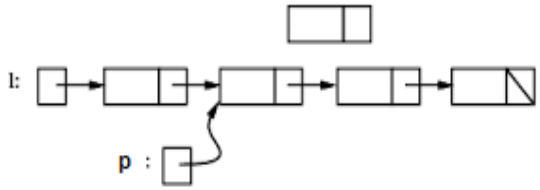
Adding new node

ADDING A NODE

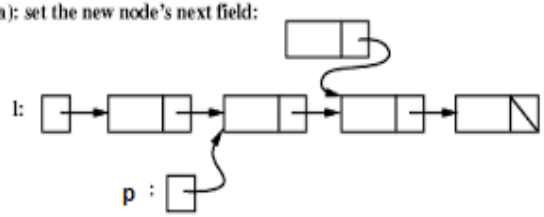
Here's the original list, with n pointing to one of its nodes:



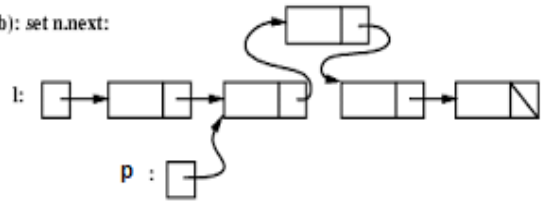
Step 1: create a new node

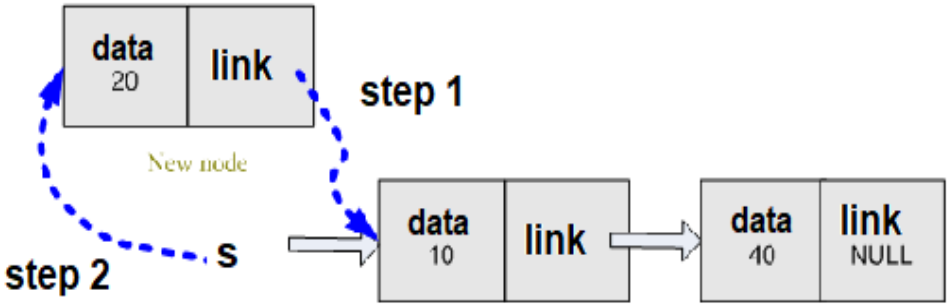


Step 2(a): set the new node's next field:



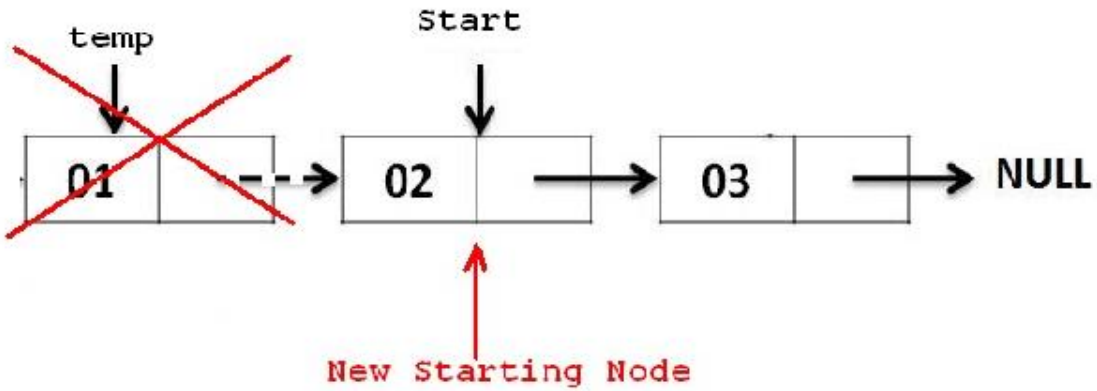
Step 2(b): set n.next:

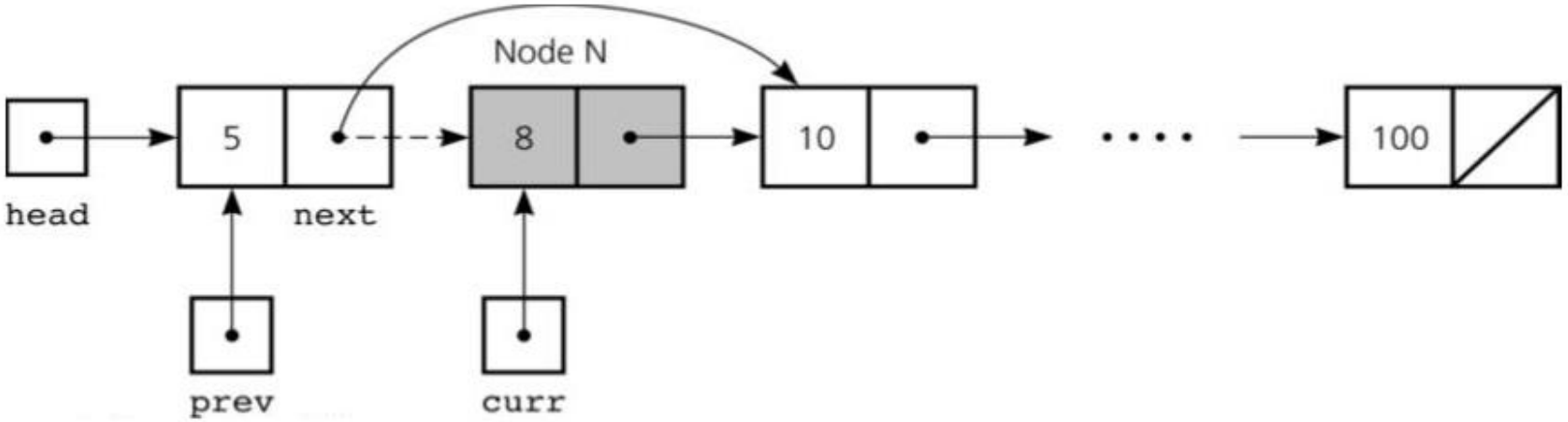




Linked list

Deleting a node





Functions

What is function?

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

Create a Function

C++ provides some pre-defined functions, such as `main()`, which is used to execute code. But you can also create your own functions to perform certain actions.

To create (often referred to as *declare*) a function, specify the name of the function, followed by parentheses **()**:

Syntax

```
Ordinary type myFunction() {
  // code to be executed
}
```

Example Explained

- `myFunction()` is the name of the function
- `Ordinary type` the value of the function which will be returned. You will learn more about return values later
- inside the function (the body), add code that defines what the function should do

Call a Function

Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are called.

To call a function, write the function's name followed by two parentheses `()` and a semicolon `;`

In the following example, `myFunction()` is used to print a text (the action), when it is called:

Example

Inside `main`, call `myFunction()`:

```
// Create a function
void myFunction() {
  cout << "I just got executed!";
}

void main() {
  myFunction(); // call the function
}
```

Function Declaration and Definition Notes:

A C++ function consist of two parts:

- **Declaration:** the function's name, return type, and parameters (if any)
- **Definition:** the body of the function (code to be executed)

```
void myFunction() { // declaration
  // the body of the function (definition)
}
```

Note: If a user-defined function, such as `myFunction()` is declared after the `main()` function, **an error will occur**. It is because C++ works from top to bottom; which means that if the function is not declared above `main()`, the program is unaware of it:

Example

```
void  main() {
  myFunction();
}
void myFunction() {
  cout << "I just got executed!";
}
// Error
```

However, it is possible to separate the declaration and the definition of the function - for code optimization.

You will often see C++ programs that have function declaration above `main()`, and function definition below `main()`. This will make the code better organized and easier to read:

Example

```
// Function declaration
void myFunction();

// The main method
void  main() {
  myFunction();  // call the function
}
// Function definition
void myFunction() {
  cout << "I just got executed!";
}
```

Another type of functions:

Syntax

Data type Function_name(Parameter list);

{

function body

return statement "when function type is not void"

}

Ex)

```
int sum( int x ,  int y);
{
   int s= x + y;
   return s;
}
```

Ex1) Write a program to find the value of y by using function where

$$y=\frac{x^n+h^m}{z^r}$$

Ex2) Write a program to find the value of Z by using function where

$$Z=\frac{x!+n!}{m!}$$

Ex3) Write a program to find the value of S by using function where

$$S=\frac{\sum_{i=1}^{7} i*\sum_{j=3}^{9} j}{\sum_{k=2}^{8} k}$$

```
#include<iostream.h>
#include<conio.h>

void print(char c, int& t)
{
  for(int i=1;i<=t;i++)
  cout<<i<<"-  "<<c<<'\n';
  t=t+i;
  cout<<"the value of t= "<<t<<'\n';

}

void main()
{  clrscr();
   int x;
   char ch;

   cout<<"Give the character: " ;
   cin>>ch;
   cout<<"Give the value of x: " ;
   cin>>x;
   cout<<"the value of x= "<<x<<'\n';
   print(ch,x);
   cout<<"the value of x= "<<x<<'\n';
   getch();
}
```

```
#include<iostream.h>
#include<conio.h>

void swap(int& x, int& y)
{
  int z;
  z=x;
  x=y;
  y=z;

}

void main()
{  clrscr();
   int x,n,h,m;


   cout<<"Give the value of x: " ;
   cin>>x;
   cout<<"Give the value of n: " ;
   cin>>n;
   swap(x,n);
   cout<<"After swaping x= "<<x<<"  and n= "<<n<<'\n'
;
   cout<<"Give the value of h: " ;
   cin>>h;
   cout<<"Give the value of m: " ;
   cin>>m;
   swap(h,m);
   cout<<"After swaping h= "<<h<<"  and m= "<<m<<'\n'
;

   getch();
}
```

```
#include<iostream.h>
#include<conio.h>

int sum(int a[10],int m)
{
  int f=0;
    for(int i=0;i<m;i++)
     f+=a[i];
 return (f);
}
void reading(int a[10],int &n)
{
  cout<<"how many element :";
   cin>>n;
   cout<<"Now give the elements :"<<'\n';
   for(int i=0;i<n;i++)
   { cout<<"x["<<i+1<<"]= ";
     cin>>a[i];
    }

}

void main()
{  clrscr();
   int x[10];
   int n;
   reading(x,n);
   cout<<"The value of summation=  "<<sum(x,n) ;

   getch();
}
```

Ex1) Write a program to find the value of y by using function where

$$y=\frac{x^n+h^m}{z^r}$$

Ex2) Write a program to find the value of Z by using function where

$$Z=\frac{x!+n!}{m!}$$

Ex3) Write a program to find the value of S by using function where

$$S=\frac{\sum_{i=1}^{7} i*\sum_{j=3}^{9} j}{\sum_{k=2}^{8} k}$$

Solution of example 1

```
#include<iostream.h>
#include<conio.h>

int power(int x, int n)
{
  int p=1;
  if (n>0)
    for(int i=1;i<=n;i++)
     p*=x;
```

```
 return (p);
}

void main()
{  clrscr();
   int x,n,h,m,z,r;
   float y;

   cout<<"Give the value of x: " ;
   cin>>x;
   cout<<"Give the value of n: " ;
   cin>>n;
   cout<<"Give the value of h: " ;
   cin>>h;
   cout<<"Give the value of m: " ;
   cin>>m;
   cout<<"Give the value of z: " ;
   cin>>z;
   cout<<"Give the value of r: " ;
   cin>>r;

   y=(power(x,n)+power(h,m))/(power(z,r)*1.0);
   cout<<"The value of Y=  "<<y ;

   getch();
}
```

Solution of example 2

```
#include<iostream.h>
#include<conio.h>

int fact(int x)
{
  int f=1;
  if (x>0)
    for(int i=1;i<=x;i++)
     f*=i;
```

```
 return (f);
}

void main()
{  clrscr();
   int x,n,m;
   float z;

   cout<<"Give the value of x: " ;
   cin>>x;
   cout<<"Give the value of n: " ;
   cin>>n;
   cout<<"Give the value of m: " ;
   cin>>m;

   z=(fact(x)+fact(n))/(fact(m)*1.0);
   cout<<"The value of Z=  "<<z ;

   getch();
}
```

Solution of example 3

```
#include<iostream.h>
#include<conio.h>

int sum(int x, int n)
{
  int p=0;
    for(int i=x;i<=n;i++)
     p+=i;
 return (p);
}

void main()
{  clrscr();
   float s;
   s=sum(1,7)*sum(3,9)/(sum(2,8)*1.0);
   cout<<"The value of S=  "<<s ;
```

```
    getch();
}
```

```
#include<iostream.h>
#include<conio.h>

int fact(int x)
{
  int f;
  if (x==0) return 1;
   else  f=x* fact(x-1);
 return (f);
}

void main()
{  clrscr();
   int x,n,m;
   float z;

   cout<<"Give the value of x: " ;
   cin>>x;
   cout<<"Give the value of n: " ;
   cin>>n;
   cout<<"Give the value of m: " ;
   cin>>m;

   z=(fact(x)+fact(n))/(fact(m)*1.0);
   cout<<"The value of Z=  "<<z ;

   getch();
}
```

```
#include<iostream.h>
#include<conio.h>

int power(int x, int n)
{
  int p;
  if (n==0) return 1;
   else
     p=x*power(x,n-1);
```

```
 return (p);
}

void main()
{  clrscr();
   int x,n,h,m,z,r;
   float y;

   cout<<"Give the value of x: " ;
   cin>>x;
   cout<<"Give the value of n: " ;
   cin>>n;
   cout<<"Give the value of h: " ;
   cin>>h;
   cout<<"Give the value of m: " ;
   cin>>m;
   cout<<"Give the value of z: " ;
   cin>>z;
   cout<<"Give the value of r: " ;
   cin>>r;

   y=(power(x,n)+power(h,m))/(power(z,r)*1.0);
   cout<<"The value of Y=  "<<y ;

   getch();
}
```

```
#include<iostream.h>
#include<conio.h>

class square
{
public:
void side(int a);
int area();
int perimeter();
private:
int s;

};
void square::side(int a)
{s=a;
}

int square::area()
{
return s*s;
}
int square::perimeter()
{
return(s*4);
}

void main ()
{clrscr();

square x,y;

x.side(5);
y.side(7);
cout<<"area of x= "<<x.area()<<'\n';
cout<<"perimeter of x= "<<x.perimeter()<<'\n';
cout<<"area of y= "<<y.area()<<'\n';
cout<<"perimeter of y= "<<y.perimeter();

getch();
}
```

```
#include<iostream.h>
#include<conio.h>

class shape
{
public:
void side(int a,int b);
int area_square();
int area_rectangle();
float area_circle();
int perimeter_sequare();
int perimeter_rectangle();
float circumference_circle();

private:
int s,r;

};

void shape::side(int a,int b)
{s=a;
 r=b;
}

int shape::area_square()
{
return r*r;
}
int shape::area_rectangle()
{
return s*r;
}
float shape::area_circle()
{  float hr;
hr=r/2;
return (hr*hr*3.14);
}
int shape::perimeter_sequare()
{
return(r*4);
```

```
}
int shape::perimeter_rectangle()
{
return((s+r)*2);
}
float shape::circumference_circle()
{

return(r*3.14);
}


void main ()
{clrscr();

shape x,y;

x.side(5,4);
y.side(7,3);

cout<<"area of sequare (x)= "<<x.area_square()<<'\n';
cout<<"area of rectangle (x)=
"<<x.area_rectangle()<<'\n';
cout<<"area of circle (x)= "<<x.area_circle()<<'\n';
cout<<"perimeter of sequare (x)=
"<<x.perimeter_sequare()<<'\n';
cout<<"perimeter of rectangle (x)=
"<<x.perimeter_rectangle()<<'\n';
cout<<"circumference of circle (x)=
"<<x.circumference_circle()<<'\n'<<'\n';

cout<<"area of sequare (y)= "<<y.area_square()<<'\n';
cout<<"area of rectangle (y)=
"<<y.area_rectangle()<<'\n';
cout<<"area of circle (y)= "<<y.area_circle()<<'\n';
cout<<"perimeter of rectangle (y)=
"<<y.perimeter_rectangle()<<'\n';
cout<<"perimeter of sequare (y)=
"<<y.perimeter_sequare()<<'\n';
```

```
cout<<"circumference of circle (y)=
"<<y.circumference_circle()<<'\n';

getch();

}
```

```
#include<iostream.h>
#include<conio.h>

class square
{
public:

square(int a);
int area();
int perimeter();
private:
int s;

};

square::square(int a)
{s=a;
}
int square::area()
{
return s*s;
}
int square::perimeter()
{
return(s*4);
}
void main ()
{clrscr();
square x(5),y(7),z(8);
cout<<"area of x= "<<x.area()<<'\n';
cout<<"perimeter of x= "<<x.perimeter()<<'\n';

cout<<"area of z= "<<z.area()<<'\n';
cout<<"perimeter of z= "<<z.perimeter()<<'\n';

cout<<"area of y= "<<y.area()<<'\n';
cout<<"perimeter of y= "<<y.perimeter();

getch();
}
```

```
#include<iostream.h>
#include<conio.h>

class square
{
public:

square(int a);
int area();

private:
int s;

};


square::square(int a)
{s=a;
}

int square::area()
{
return s*s;
}

void main ()
{clrscr();

square x(4),y(3),z(2);
float sh;
   cout<<sh;

   sh=x.area()- y.area()/2.0 -z.area();

   cout<<"The shadowed area is "<<sh;
getch();
}
```

Data Encapsulation and Abstraction

All C++ programs are composed of the following two fundamental elements:

- Program statements code: This is the part of a program that performs actions and they are called methods.
- Program data: The data is the information of the program which affected by the program methods.

Encapsulation is an Object Oriented Programming concept that binds together the data and methods that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of data hiding.

Data encapsulation is a mechanism of bundling the data, and the methods that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes. We already have studied that a class can contain *private* and *public* members. By default, all items defined in a class are *private*. For example:

```
class Box
{
private:
   double length; // Length of a
   box double breadth; // Breadth
   of a box double height; //
   Height of a box
public:
  double getVolume(void)
   {
     return length * breadth * height;
   }
};
```

The variables length, breadth, and height are private. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.

Any C++ program where you implement a class with public and private

members is an example of data encapsulation and data abstraction. Consider the following example:

Ex1: Data Encapsulation and Data Abstraction

```
#include <iostream>
using namespace std;

class Adder
{
private:
        // hidden data from outside
        world int total;

public:
        // constructor
        Adder(int i = 0)
        {
                total = i;
        }

        // interface to outside world
        void addNum(int number)
        {
                total += number;
        }

        // interface to outside world
        int getTotal()
        {
                return total;
        }
};

int main()
{
        Adder a;
        a.addNum(10);
        a.addNum(20);
        a.addNum(30);
        cout << "Total " << a.getTotal() << endl;

        system("pause"
        ); return 0;
}
```

Output:

Total 60

Above class adds numbers together, and returns the sum. The public members **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that is hidden from the outside world, but is needed for the

class to operate properly.

Abstract Class (Interfaces)

An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.

The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

A class is made abstract by declaring at least one of its functions as pure **virtual method**. A pure virtual function is specified by placing "= 0" in its declaration as follows:

```
class Box
{
private:
   double length; // Length of a box
   double breadth; // Breadth of a
   box double height; // Height of a
   box
public:
// pure virtual function
  virtual double getVolume() = 0;
};
```

The purpose of an abstract class is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an interface. Attempting to instantiate an object of an abstract class causes a compilation error.

Consider the following example where parent class provides an interface to the base class to implement a function called **getArea**():

Ex2: Abstract Classes

```
#include <iostream>
using namespace std;

// Base class
class Shape
{
public:
        // pure virtual function providing interface framework.
        virtual int getArea() = 0;
        void setWidth(int w)
        {
                width = w;
        }
        void setHeight(int h)
        {
                height = h;
        }
protected:
        int width;
        int height;
};
// Derived classes
class Rectangle : public Shape
{
public:
        int getArea()
        {
                return (width * height);
        }
};
```

```
class Triangle : public Shape
{
public:
        int getArea()
        {
                return (width * height) / 2;
        }
};

int main(void)
{
        Rectangle Rect;
        Triangle  Tri;

        Rect.setWidth(5);
        Rect.setHeight(7);
        // Print the area of the object.
        cout << "Total Rectangle area: " << Rect.getArea() << endl;

        Tri.setWidth(5);
        Tri.setHeight(7);
        // Print the area of the object.
        cout << "Total Triangle area: " << Tri.getArea() << endl;

        system("pause"
        ); return 0;
}
```

Output:

```
Total Rectangle area: 35
Total Triangle area: 17
```

You can see how an abstract class defined an interface in terms of getArea() and two other classes implemented same function but with different algorithm to calculate the area specific to the shape.

Inheritance in C++

The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object Oriented Programming.

Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.

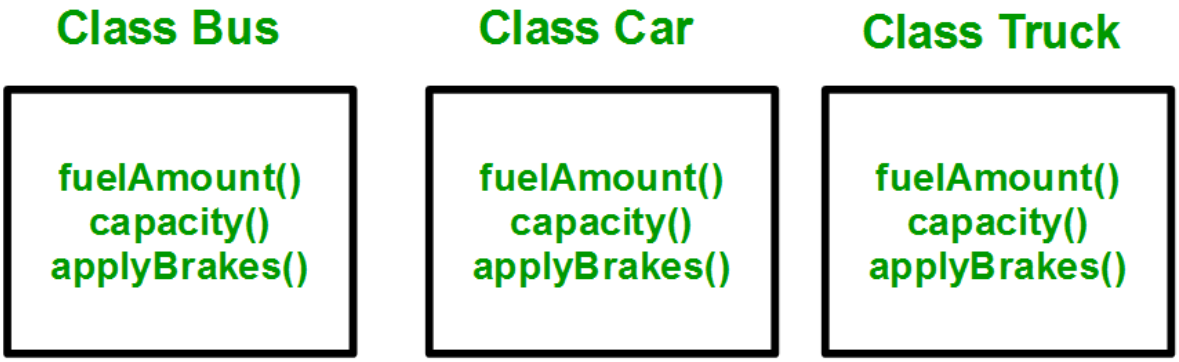
Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.

The article is divided into following subtopics:

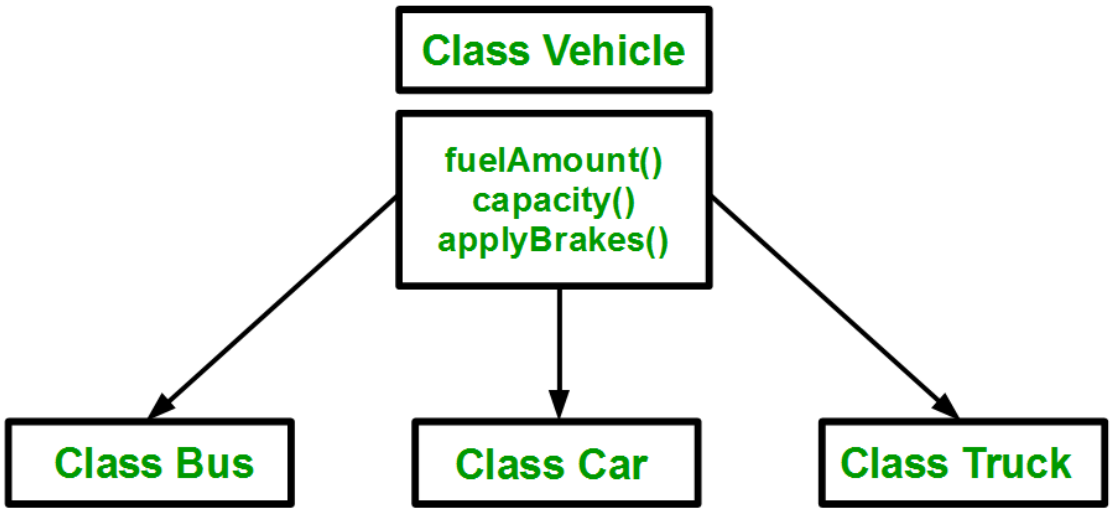
1. Why and when to use inheritance?
2. Modes of Inheritance
3. Types of Inheritance

Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below figure:



You can clearly see that above process results in duplication of same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:



Using inheritance, we have to write the functions only one time instead of three times as we have inherited rest of the three classes from base class(Vehicle).
Implementing inheritance in C++: For creating a sub-class which is inherited from the base class we have to follow the below syntax.
Syntax:

```
class subclass_name : access_mode base_class_name
{
  //body of subclass
};
```

Here, **subclass_name** is the name of the sub class, **access_mode** is the mode in which you want to inherit this sub class for example: public, private etc. and **base_class_name** is the name of the base class from which you want to inherit the sub class.

Note: A derived class doesn't inherit ***access*** to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

```
// C++ program to demonstrate implementation
// of Inheritance

#include <bits/stdc++.h>
using namespace std;

//Base class
class Parent
{
   public:
     int id_p;
};

// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
   public:
     int id_c;
```

```
};

//main function
int main()
  {

      Child obj1;

      // An object of class child has all data members
      // and member functions of class parent
      obj1.id_c = 7;
      obj1.id_p = 91;
      cout << "Child id is " <<  obj1.id_c << endl;
      cout << "Parent id is " <<  obj1.id_p << endl;

      return 0;
  }
```

Output:

```
Child id is 7

Parent id is 91
```

In the above program the 'Child' class is publicly inherited from the 'Parent' class so the public data members of the class 'Parent' will also be inherited by the class 'Child'.

Modes of Inheritance

1. **Public mode**: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
2. **Protected mode**: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.
3. **Private mode**: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

Note : The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C and D all contain the variables x, y and z in below example. It is just question of access.

```
// C++ Implementation to show that a derived class
// doesn't inherit access to private data members.
// However, it does inherit a full parent object
class A
{
public:
   int x;
protected:
   int y;
private:
   int z;
};
```

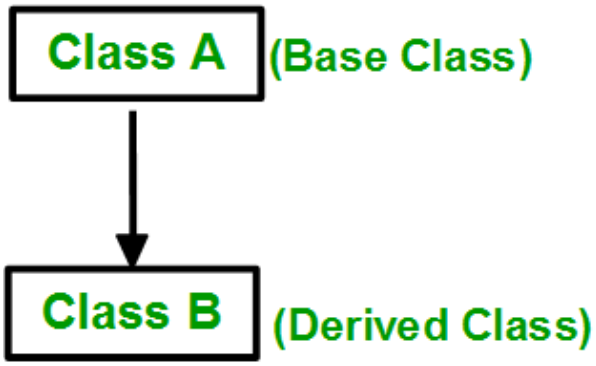
```
class B : public A
{
   // x is public
   // y is protected
   // z is not accessible from B
};

class C : protected A
{
   // x is protected
   // y is protected
   // z is not accessible from C
};

class D : private A    // 'private' is default for classes
{
   // x is private
   // y is private
   // z is not accessible from D
};
```

The below table summarizes the above three modes and shows the access specifier of the members of base class in the sub class when derived in public, protected and private modes:

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Types of Inheritance in C++

1. **Single Inheritance**: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.



Syntax:

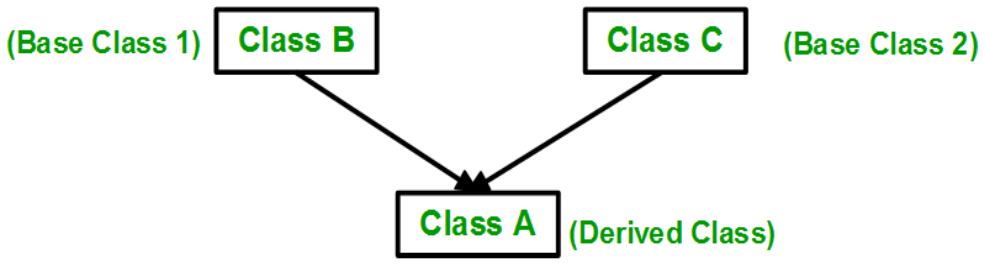
```
class subclass_name : access_mode base_class

{

//body of subclass

};
```

```
// C++ program to explain
// Single inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

// sub class derived from two base classes
class Car: public Vehicle{

};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Output:

```
This is a vehicle
```

2. **Multiple Inheritance:** Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base class**



Syntax:

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
  //body of subclass
};
```

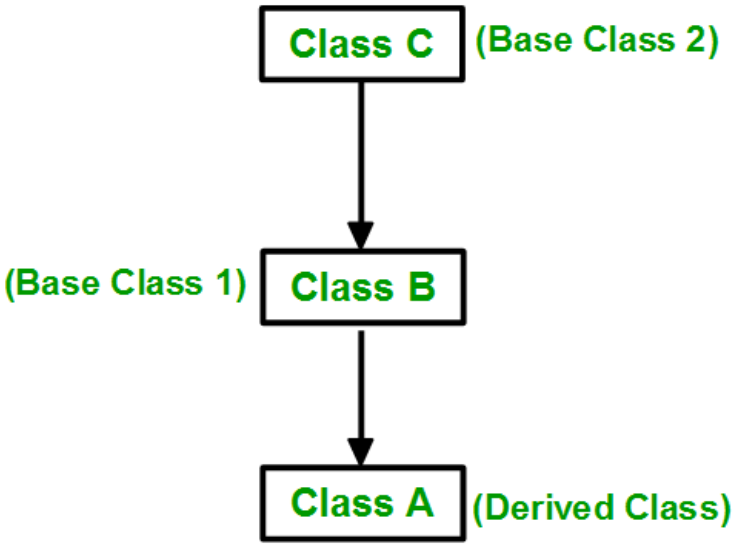
Here, the number of base classes will be separated by a comma (‘, ‘) and access mode for every base class must be specified.

```
// C++ program to explain
// multiple inheritance
#include <iostream>
using namespace std;

// first base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

// second base class
class FourWheeler {
  public:
    FourWheeler()
    {
      cout << "This is a 4 wheeler Vehicle" << endl;
    }
};

// sub class derived from two base classes
class Car: public Vehicle, public FourWheeler {

};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Output:

```
This is a Vehicle
This is a 4 wheeler Vehicle
```

3. **Multilevel Inheritance**: In this type of inheritance, a derived class is created from another derived class.



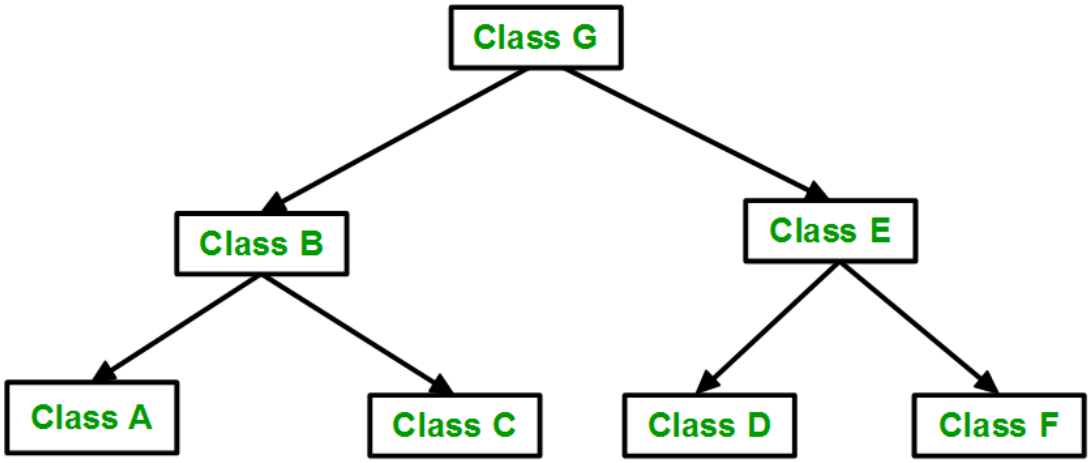

```
// C++ program to implement
// Multilevel Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};
class fourWheeler: public Vehicle
{  public:
    fourWheeler()
    {
      cout<<"Objects with 4 wheels are vehicles"<<endl;
    }
};
// sub class derived from two base classes
class Car: public fourWheeler{
   public:
     car()
     {
       cout<<"Car has 4 Wheels"<<endl;
     }
```

```
};

// main function
int main()
{
   //creating object of sub class will
   //invoke the constructor of base classes
   Car obj;
   return 0;
}
```

output:

```
This is a Vehicle

Objects with 4 wheels are vehicles

Car has 4 Wheels
```

4. **Hierarchical Inheritance**: In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.




```
// C++ program to implement
// Hierarchical Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};


// first sub class
```

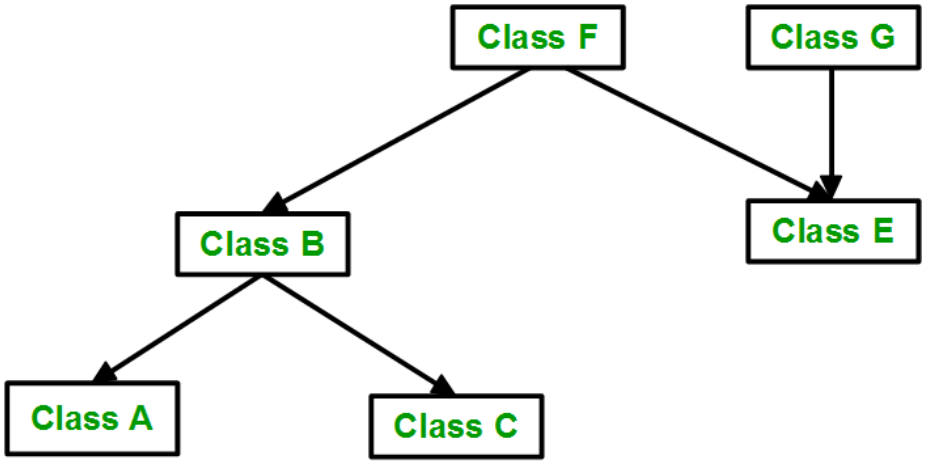
```
class Car: public Vehicle
{

};

// second sub class
class Bus: public Vehicle
{

};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base class
    Car obj1;
    Bus obj2;
    return 0;
}
```

Output:

```
This is a Vehicle

This is a Vehicle
```

5. **Hybrid (Virtual) Inheritance**: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.

Below image shows the combination of hierarchical and multiple inheritance:



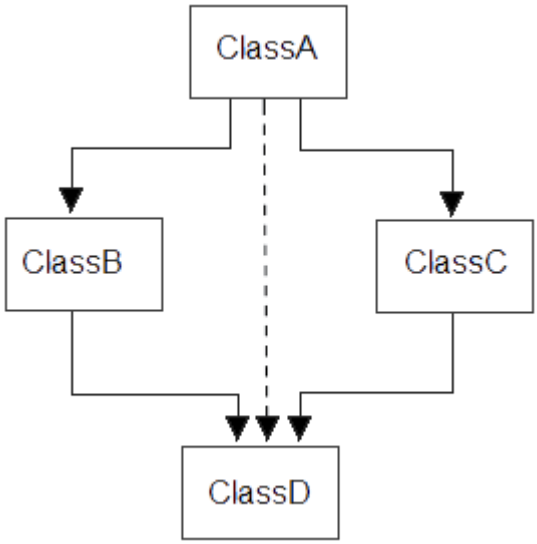

```
// C++ program for Hybrid Inheritance

#include <iostream>
using namespace std;

// base class
class Vehicle
{
```

```
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

//base class
class Fare
{
    public:
    Fare()
    {
        cout<<"Fare of Vehicle\n";
    }
};

// first sub class
class Car: public Vehicle
{

};

// second sub class
class Bus: public Vehicle, public Fare
{

};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base class
    Bus obj2;
    return 0;
}
```

Output:

```
This is a Vehicle

Fare of Vehicle
```

A special case of hybrid inheritance : Multipath inheritance:
A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. An ambiguity can arrise in this type of inheritance.



Consider the following program:

```
// C++ program demonstrating ambiguity in Multipath Inheritance

#include<iostream.h>
#include<conio.h>
class ClassA
    {
        public:
        int a;
    };

    class ClassB : public ClassA
    {
        public:
        int b;
    };
    class ClassC : public ClassA
    {
        public:
        int c;
    };

    class ClassD : public ClassB, public ClassC
    {
        public:
        int d;
    };

    void main()
    {

        ClassD obj;

        //obj.a = 10;               //Statement 1, Error
        //obj.a = 100;              //Statement 2, Error
```

```
        obj.ClassB::a = 10;       //Statement 3
        obj.ClassC::a = 100;      //Statement 4

        obj.b = 20;
        obj.c = 30;
        obj.d = 40;

        cout<< "\n A from ClassB  : "<< obj.ClassB::a;
        cout<< "\n A from ClassC  : "<< obj.ClassC::a;

        cout<< "\n B : "<< obj.b;
        cout<< "\n C : "<< obj.c;
        cout<< "\n D : "<< obj.d;

    }
```

Output:

```
A from ClassB  : 10
A from ClassC  : 100
B : 20
C : 30
D : 40
```

In the above example, both ClassB & ClassC inherit ClassA, they both have single copy of ClassA. However ClassD inherit both ClassB & ClassC, therefore ClassD have two copies of ClassA, one from ClassB and another from ClassC.
If we need to access the data member a of ClassA through the object of ClassD, we must specify the path from which a will be accessed, whether it is from ClassB or ClassC, bco'z compiler can't differentiate between two copies of ClassA in ClassD.

There are 2 ways to avoid this ambiguity:

1. **Use scope resolution operator**
2. **Use virtual base class**

Avoiding ambiguity using scope resolution operator:
Using scope resolution operator we can manually specify the path from which data member a will be accessed, as shown in statement 3 and 4, in the above example.
filter_none
edit
play_arrow
brightness_4

```
obj.ClassB::a = 10;       //Statement 3
obj.ClassC::a = 100;      //Statement 4
```

Note : Still, there are two copies of ClassA in ClassD.

Avoiding ambiguity using virtual base class:

```
include<iostream.h>
    #include<conio.h>

    class ClassA
    {
        public:
        int a;
    };

    class ClassB : virtual public ClassA
    {
        public:
        int b;
    };
    class ClassC : virtual public ClassA
    {
        public:
        int c;
    };

    class ClassD : public ClassB, public ClassC
    {
        public:
        int d;
    };

    void main()
    {

        ClassD obj;

        obj.a = 10;       //Statement 3
        obj.a = 100;      //Statement 4

        obj.b = 20;
        obj.c = 30;
        obj.d = 40;

        cout<< "\n A : "<< obj.a;
        cout<< "\n B : "<< obj.b;
        cout<< "\n C : "<< obj.c;
        cout<< "\n D : "<< obj.d;

    }
```

Output:

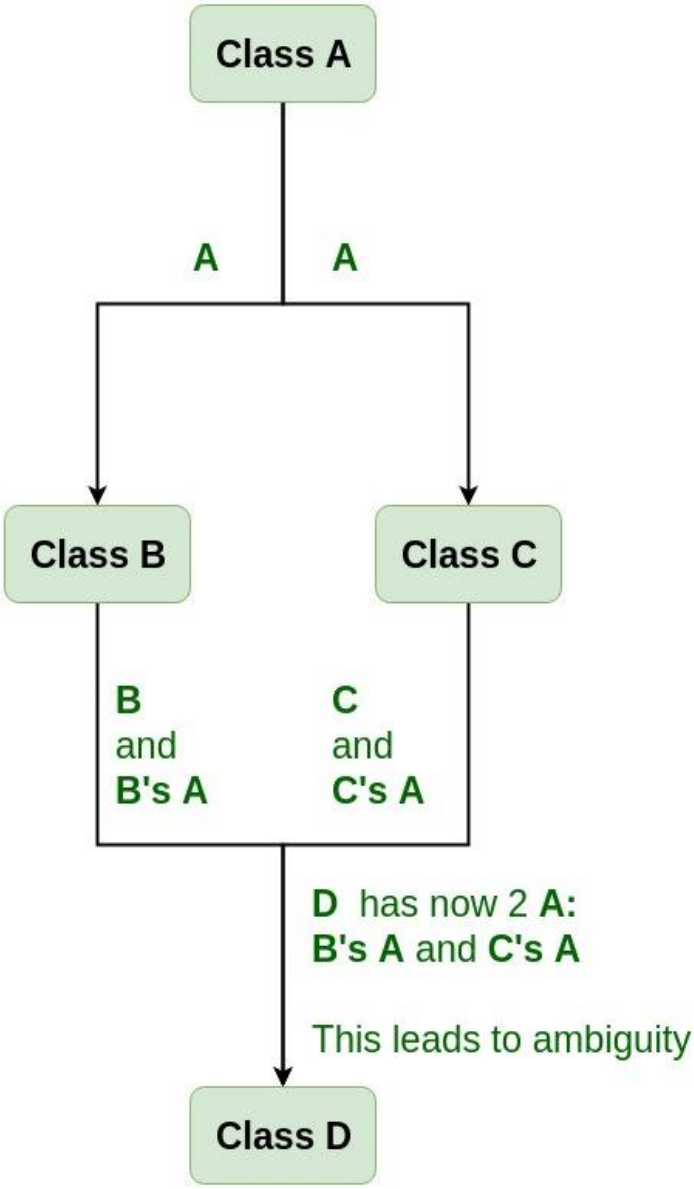
```
A : 100
B : 20
C : 30
D : 40
```

According to the above example, ClassD has only one copy of ClassA, therefore, statement 4 will overwrite the value of a, given at statement 3.

Virtual base class in C++

Virtual base classes are used in virtual inheritance in a way of preventing multiple "instances" of a given class appearing in an inheritance hierarchy when using multiple inheritances.

Need for Virtual Base Classes:
Consider the situation where we have one class **A** .This class is **A** is inherited by two other classes **B** and **C**. Both these class are inherited into another in a new class **D** as shown in figure below.



As we can see from the figure that data members/function of class **A** are inherited twice to class **D**. One through class **B** and second through class **C**. When any data / function member of class **A** is accessed by an object of class **D**, ambiguity arises as to which data/function

member would be called? One inherited through **B** or the other inherited through **C**. This confuses compiler and it displays error.

Example: To show the need of Virtual Base Class in C++

```
#include <iostream>
using namespace std;

class A {
public:
   void show()
   {
      cout << "Hello form A \n";
   }
};

class B : public A {
};

class C : public A {
};

class D : public B, public C {
};

int main()
{
   D object;
   object.show();
}
```

Compile Errors:

```
prog.cpp: In function 'int main()':
prog.cpp:29:9: error: request for member 'show' is ambiguous
  object.show();
         ^
prog.cpp:8:8: note: candidates are: void A::show()
   void show()
        ^
prog.cpp:8:8: note:                 void A::show()
```

How to resolve this issue?

To resolve this ambiguity when class **A** is inherited in both class **B** and class **C**, it is declared as **virtual base class** by placing a keyword **virtual** as :

Syntax for Virtual Base Classes:

Syntax 1:

```
class B : virtual public A
```

```
{
};
```

Syntax 2:

```
class C : public virtual A
{
};
```

Note: virtual can be written before or after the **public**. Now only one copy of data/function member will be copied to class **C** and class **B** and class **A** becomes the virtual base class. Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.

Example 1

```
#include <iostream>
using namespace std;

class A {
public:
   int a;
   A() // constructor
   {
      a = 10;
   }
};

class B : public virtual A {
};

class C : public virtual A {
};

class D : public B, public C {
};

int main()
{
   D object; // object creation of class d
   cout << "a = " << object.a << endl;

   return 0;
}
```

Output:

```
a = 10
```

Explanation : The class **A** has just one data member **a** which is **public**. This class is virtually inherited in class **B** and class **C**. Now class **B** and class **C** becomes virtual base class and no duplication of data member **a** is done.

Example 2:

```
#include <iostream>
using namespace std;
```

```
class A {
public:
   void show()
   {
      cout << "Hello from A \n";
   }
};

class B : public virtual A {
};

class C : public virtual A {
};

class D : public B, public C {
};

int main()
{
   D object;
   object.show();
}
```

Output:

```
Hello from A
```