VIRTUAL FUNCITONS

Pointers:

- Some C++ tasks are performed more easily with pointers, and other C++ tasks, such as dynamic memory allocation, cannot be performed without them.
- As you know every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory.

The Address-of Operator &:

- The & operator can find address occupied by a variable. If **var** is a variable then, **&var**gives the address of that variable.

Example 1: Address-of Operator

```
#include <iostream>
using namespace std;
int main() {
   int var1 = 3;
   int var2 = 24;
   int var3 = 17;
   cout<<&var1<<endl;
   cout<<&var2<<endl;
   cout<<&var3<<endl;
}
```

Pointers Variables:

- Consider a normal variable as in above example, these variables holds data.
- But pointer variables or simply pointers are the special types of variable that holds memory address instead of data.
- Pointer can be declared as int *p; OR, int* p;
- The statement above defines a pointer variable **p**.
- The pointer **p** holds the memory address. The asterisk is a dereference operator which means pointer to. Here pointer **p**is a pointer to **int**, that is, it is pointing an integer.
- A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it.
- The general form of a pointer variable declaration is:

  type *var-name;
- Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable.
- The asterisk you used to declare a pointer is the same asterisk that you use for multiplication.
- However, in this statement the asterisk is being used to designate a variable as a pointer.
- Following are the valid pointer declaration:

```
int    *ip;    // pointer to an integer
double *dp;    // pointer to a double
float  *fp;    // pointer to a float
char   *ch     // pointer to character
```

- The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address.
- The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

Using Pointers in C++:

- There are few important operations, which we will do with the pointers very frequently. **(a)** we define a pointer variables **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable.
- This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand.
- Following example makes use of these operations:

```
#include <iostream>
using namespace std;
int main ()
{
  int  var = 20;   // actual variable declaration.
  int  *ip;        // pointer variable
  ip = &var;       // store address of var in pointer variable
  cout << "Value of var variable: ";
  cout << var << endl;
  // print the address stored in ip pointer variable
  cout << "Address stored in ip variable: ";
  cout << ip << endl;
  // access the value at the address available in pointer
  cout << "Value of *ip variable: ";
  cout << *ip << endl;
  return 0;
}
```

- When the above code is compiled and executed, it produces result something as follows:

```
Value of var variable: 20
Address stored in ip variable: 0xbfc601ac
Value of *ip variable: 20
```

Note: In above statement **p** is a pointer variable that holds address not ***p**. The ***p** is an expression. The content(value) of the memory address pointer **p** holds is given by expression ***p**.

Example 2: C++ Pointers

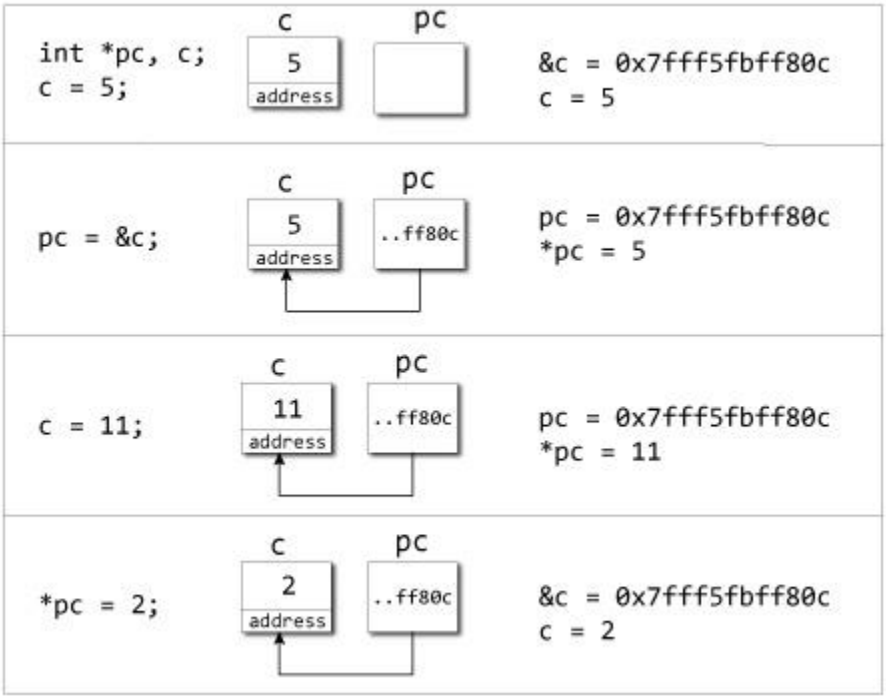
C++ Program to demonstrate the working of pointer.

```
#include <iostream>
using namespace std;
int main() {
```

```
    int *pc, c;
   c = 5;
    cout<< "Address of c (&c): " << &c << endl;
    cout<< "Value of c (c): " << c << endl << endl;
    pc = &c;    // Pointer pc holds the memory address of variable c
    cout<< "Address that pointer pc holds (pc): "<< pc << endl;
    cout<< "Content of the address pointer pc holds (*pc): " << *pc << endl << endl;
    c = 11;    // The content inside memory address &c is changed from 5 to 11.
    cout << "Address pointer pc holds (pc): " << pc << endl;
    cout << "Content of the address pointer pc holds (*pc): " << *pc << endl << endl;
    *pc = 2;
    cout<< "Address of c (&c): "<< &c <<endl;
    cout<<"Value of c (c): "<< c<<endl<< endl;
    return 0;
}
```

Output

```
Address of c (&c): 0x7fff5fbff80c
Value of c (c): 5
Address that pointer pc holds (pc): 0x7fff5fbff80c
Content of the address pointer pc holds (*pc): 5
Address pointer pc holds (pc): 0x7fff5fbff80c
Content of the address pointer pc holds (*pc): 11
Address of c (&c): 0x7fff5fbff80c
Value of c (c): 2
```



Explanation of program

- When **c = 5;** the value 5 is stored in the address of variable **c**.
- When **pc = &c;** the pointer **pc** holds the address of **c** and the expression ***pc**contains the value of that address which is 5 in this case.
- When **c = 11;** the address that pointer **pc** holds is unchanged. But the expression***pc** is changed because now the address **&c** (which is same as **pc**) contains 11.
- When ***pc = 2;** the content in the address **pc**(which is equal to **&c**) is changed from 11 to 2. Since the pointer **pc** and variable **c** has address, value of **c** is changed to 2.

Manipulating pointers :

- The value "pointed to" by a pointer can be "retrieved" or dereferenced by using the unary * operator;
- for example: int *p = ... int x = *p;
- The memory address of a variable is returned with the unary ampersand (&) operator; for example int *p = &x;
- Dereferenced pointer values can be used in normal expressions; for example: *pi += 5; or (*pi)++

Pointer arithmetic

- Pointer arithmetic can be used to adjust where a pointer points;
- for example, if pc points to the first element of an array, after executing pc+=3; then pc points to the fourth element
- A pointer can even be dereferenced using array notation; for example pc[2] represents the value of the array element which is two elements beyond the array element currently pointed to by pc
- In summary, for an array c, *(c+i)≡c[i] and c+i≡&c[i]
- A pointer is a variable, but an array name is not; therefore pc=c and pc++ are valid, but c=pc and c++ are not.

C++ Pointer to an Array

- An array name is a constant pointer to the first element of the array. Therefore, in the declaration:

```
double balance[50];
```

- **balance** is a pointer to &balance[0], which is the address of the first element of the array balance.
- Thus, the following program fragment assigns **p** the address of the first element of **balance**:

```
double *p;
double balance[10];
p = balance;
```

- It is legal to use array names as constant pointers, and vice versa.
- Therefore, *(balance + 4) is a legitimate way of accessing the data at balance[4].
- Once you store the address of first element in p, you can access array elements using *p, *(p+1), *(p+2) and so on.

- Below is the example to show all the concepts discussed above:

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

int main ()
{
  // an array with 5 elements.
  double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
  double *p;

  p = balance;

  // output each array element's value
  cout << "Array values using pointer " << endl;
  for ( int i = 0; i < 5; i++ )
  {
     cout << "*(p + " << i << ") : ";
     cout << *(p + i) << endl;
  }

  cout << "Array values using balance as address " << endl;
  for ( int i = 0; i < 5; i++ )
  {
     cout << "*(balance + " << i << ") : ";
     cout << *(balance + i) << endl;
  }

  return 0;
}
```

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

```
Array values using pointer
*(p + 0) : 1000
*(p + 1) : 2
*(p + 2) : 3.4
*(p + 3) : 17
*(p + 4) : 50
Array values using balance as address
*(balance + 0) : 1000
*(balance + 1) : 2
*(balance + 2) : 3.4
*(balance + 3) : 17
*(balance + 4) : 50
```

- In the above example, p is a pointer to double which means it can store address of a variable of double type.
- Once we have address in p, then ***p** will give us value available at the address stored in p, as we have shown in the above example.

C++ array of pointers

- Following example, which makes use of an array of 3 integers:

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main ()
{
  int  var[MAX] = {10, 100, 200};
  for (int i = 0; i < MAX; i++)
  {
    cout << "Value of var[" << i << "] = ";
    cout << var[i] << endl;
  }
  return 0;
}
```

When the above code is compiled and executed, it produces the following result:
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200

- There may be a situation, when we want to maintain an array, which can store pointers to an int or char or any other data type available.
- Following is the declaration of an array of pointers to an integer:
  int *ptr[MAX];
- This declares **ptr** as an array of MAX integer pointers.
- Thus, each element in ptr, now holds a pointer to an int value.
- Following example makes use of three integers which will be stored in an array of pointers as follows:

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main ()
{
  int  var[MAX] = {10, 100, 200};
  int *ptr[MAX];
  for (int i = 0; i < MAX; i++)
  {
    ptr[i] = &var[i]; // assign the address of integer.
  }
```

```
  for (int i = 0; i < MAX; i++)
  {
    cout << "Value of var[" << i << "] = ";
    cout << *ptr[i] << endl;
  }
  return 0;
}
```

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

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

You can also use an array of pointers to character to store a list of strings as follows:

```
#include <iostream>
using namespace std;
const int MAX = 4;
int main ()
{
  char *names[MAX] = {“FE”,”SE”,”TE”,”BE” };
  for (int i = 0; i < MAX; i++)
  {
    cout << "Value of names[" << i << "] = ";
    cout << names[i] << endl;
  }
  return 0;
}
```

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

```
Value of names[0] = FE
Value of names[1] = SE
Value of names[2] = TE
Value of names[3] = BE
```

Pointers to functions :

- A function pointer is a variable that stores the address of a function that can later be called through that function pointer.
- This is useful because functions encapsulate behavior.
- For instance, every time you need a particular behavior such as drawing a line, instead of writing out a bunch of code, all you need to do is call the function.
- But sometimes you would like to choose different behaviors at different times in essentially the same piece of code. Read on for concrete examples.

Function Pointer Syntax

- The syntax for declaring a function pointer might seem messy at first, but in most cases it's really quite straight-forward once you **understand** what's going on.
- Let's look at a simple example:

  void (*foo)(int);
- In this example, foo is a pointer to a function taking one argument, an integer, and that returns void.
- It's as if you're declaring a function called "*foo", which takes an int and returns void; now, if *foo is a function, then foo must be a pointer to a function.
- Similarly, a declaration like int *x can be read as *x is an int, so x must be a pointer to an int.
- The key to writing the declaration for a function pointer is that you're just writing out the declaration of a function but with (*func_name) where you'd normally just put func_name.

Reading Function Pointer Declarations

- Sometimes people get confused when more stars are thrown in:
  void *(*foo)(int *);
- Here, the key is to read inside-out; notice that the innermost element of the expression is *foo, and that otherwise it looks like a normal function declaration. *foo should refer to a function that returns a void * and takes an int *.
- Consequently, foo is a pointer to just such a function.

Initializing Function Pointers

- To initialize a function pointer, you must give it the address of a function in your program.
- The syntax is like any other variable:

```
#include <stdio.h>
void my_int_func(int x)
{
   printf( "%d\n", x );
}
int main()
{
   void (*foo)(int);
   /* the ampersand is actually optional */
   foo = &my_int_func;

   return 0;
}
```

Using a Function Pointer

- To call the function pointed to by a function pointer, you treat the function pointer as though it were the name of the function you wish to call.
- The act of calling it performs the dereference; there's no need to do it yourself:

```
#include <stdio.h>
void my_int_func(int x)
{
   printf( "%d\n", x );
}
int main()
{
   void (*foo)(int);
   foo = &my_int_func;
   /* call my_int_func (note that you do not need to write (*foo)(2) ) */
   foo( 2 );
   /* but if you want to, you may */
   (*foo)( 2 );
   return 0;
}
```

- Note that function pointer syntax is flexible; it can either look like most other uses of pointers, with & and *, or you may omit that part of syntax.
- This is similar to how arrays are treated, where a bare array decays to a pointer, but you may also prefix the array with & to request its address.

C++ Pointer to Pointer (Multiple Indirection)

- A pointer to a pointer is a form of multiple indirection or a chain of pointers.
- Normally, a pointer contains the address of a variable.
- When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.

- A variable that is a pointer to a pointer must be declared as such.
- This is done by placing an additional asterisk in front of its name.
- For example, following is the declaration to declare a pointer to a pointer of type int:
  int **var;
- When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example:

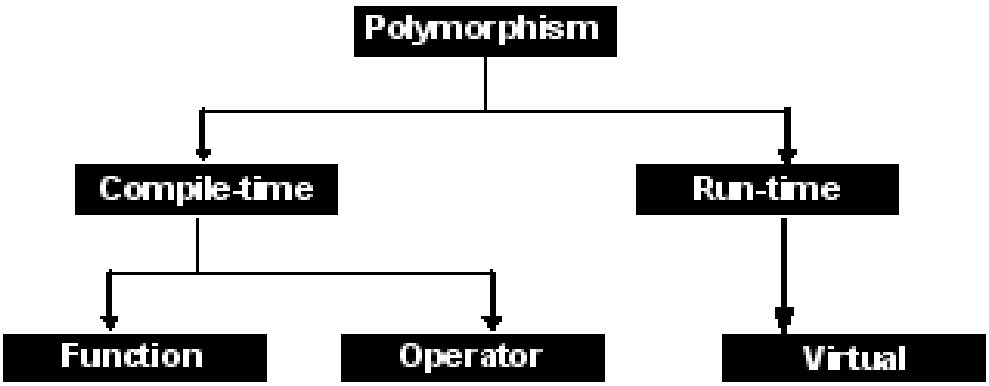
```
#include <iostream>
using namespace std;
int main ()
{
  int  var;
```

```
  int  *ptr;
  int  **pptr;
  var = 3000;
  // take the address of var
  ptr = &var;
  // take the address of ptr using address of operator &
  pptr = &ptr;
  // take the value using pptr
  cout << "Value of var :" << var << endl;
  cout << "Value available at *ptr :" << *ptr << endl;
  cout << "Value available at **pptr :" << **pptr << endl;
  return 0;
}
```

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

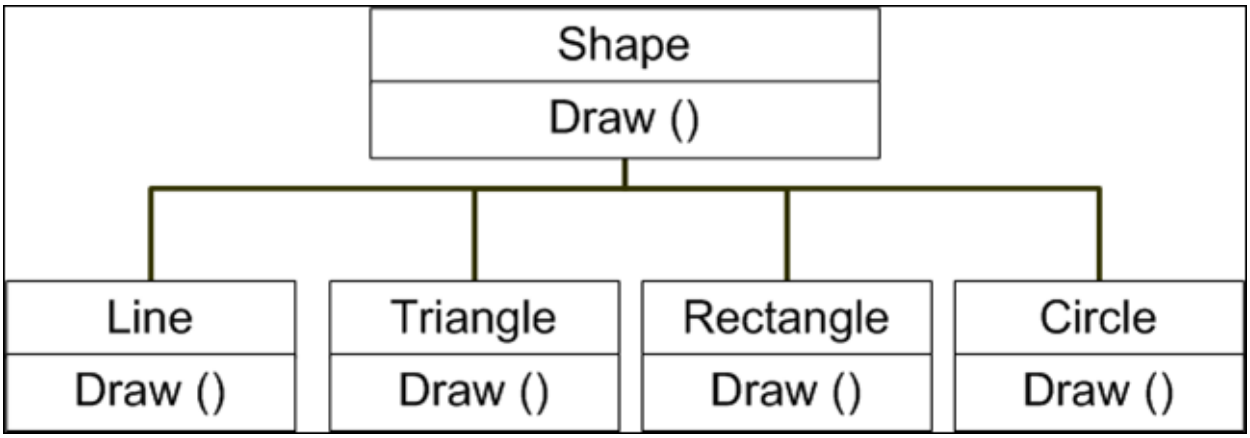
```
Value of var :3000
Value available at *ptr :3000
Value available at **pptr :3000
```

Polymorphism



- Polymorphism is the ability of an object to take on many forms.
- The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
- For example, given a base class *shape,* polymorphism enables the programmer to define different *area* methods for any number of derived classes, such as circles, rectangles and triangles.
- It has two distinct aspects:
- At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays.
- When this occurs, the object's declared type is no longer identical to its run-time type.

- Base classes may define and implement virtual methods, and derived classes can override them, which means they provide their own definition and implementation.
- At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes that override of the virtual method.
- Thus in your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.



Pointers to Objects:

- A variable that holds an address value is called a pointer variable or simply pointer.
- C++ allows you to have pointers to objects. The pointers pointing to objects are referred to as Object Pointers.

C++ Declaration and Use of Object Pointers

- Just like other pointers, the object pointers are declared by placing in front of a object pointer's name.
- It takes the following general form :

  class-name * object-pointer ;

  where class-name is the name of an already defined class and object-pointer is the pointer to an object of this class type.
- For example, to declare optr as an object pointer of Sample class type, we shall write

  Sample *optr ;

where Sample is already defined class.

- When accessing members of a class using an object pointer, the arrow operator (->) is used instead of dot operator.
- The following program illustrates how to access an object given a pointer to it.
- This C++ program illustrates the use of object pointer

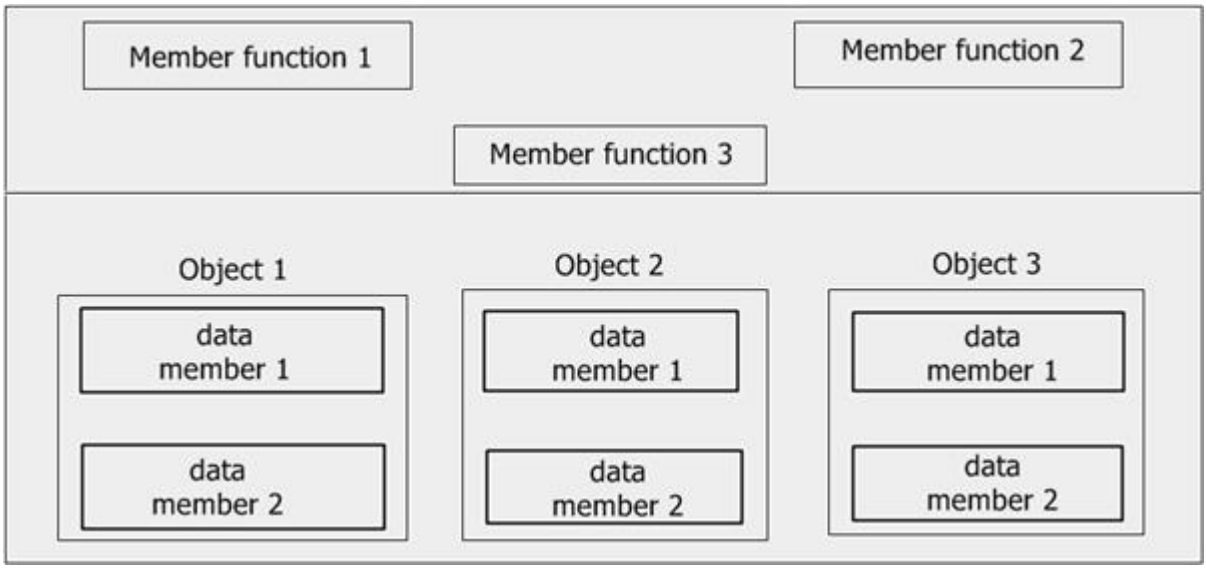
/* C++ Pointers and Objects. Declaration and Use of Pointers. This program demonstrates the use of pointers in C++ */

#include<iostream.h>

```
#include<conio.h>
class Time
{
	short int hh, mm, ss;
	public:
		Time()
		{
			hh = mm = ss = 0;
		}
		void getdata(int i, int j, int k)
		{
			hh = i;
			mm = j;
			ss = k;
		}
		void prndata(void)
		{
			cout<<"\nTime is "<<hh<<":"<<mm<<":"<<ss<<"\n";
		}
};
void main()
{
	clrscr();
	Time T1, *tptr;
	cout<<"Initializing data members using the object, with values 12, 22, 11\n";
	T1.getdata(12,22,11);
	cout<<"Printing members using the object ";
	T1.prndata();
	tptr = &T1;
	cout<<"Printing members using the object pointer ";
	tptr->prndata();
	cout<<"\nInitializing data members using the object pointer, with values 15, 10, 16\n";
	tptr->getdata(15, 10, 16);
	cout<<"printing members using the object ";
	T1.prndata();
	cout<<"Printing members using the object pointer ";
	tptr->prndata();
	getch();
}
```

C++ this Pointer

- As soon as you define a class, the member functions are created and placed in the memory space only once.
- That is, only one copy of member functions is maintained that is shared by all the objects of the class.
- Only space for data members is allocated separately for each object



- This has an associated problem.
- If only one instance of a member function exists, how does it come to know which object's data member is to be manipulated?
- For example, if member function 3 is capable of changing the value of data member2 and we want to change the value of data member2 of object1.
- How could the member function3 come to know which object's data member2 is to be changed ?
- The answer to this problem is this pointer.
- When a member function is called, it is automatically passed an implicit (in-built) argument that is a pointer to the object that invoked the function.
- This pointer is called this.
- That is if object1 is invoking member function3, then an implicit argument is passed to member function3 that points to object1 i.e., this pointer now points to object1.
- The this pointer can be thought of analogous to the ATM card.
- For instance, in a bank there are many accounts.
- The account holders can withdraw amount or view their bank-statements through Automatic-Teller-Machines.
- Now, these ATMs can withdraw from any account in the bank, but which account are they supposed to work upon ?
- This is resolved by the ATM card, which gives the identification of user and his accounts, from where the amount is withdrawn.
- Similarly, the this pointer is the ATM cards for objects, which identifies the currently-calling object.
- The this pointer stores the address of currently-calling object.

Pointers to Derived Classes

- If a base class and derived class has same function and if you write code to access that function using pointer of base class then, the function in the base class is executed even if, the object of derived class is referenced with that pointer variable.
- One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class.
- *Polymorphism* is the art of taking advantage of this simple but powerful and versatile feature.
- This can be demonstrated by an example.

```
#include <iostream>
using namespace std;
class B
{
   public:
     void display()
       { cout<<"Content of base class.\n"; }
};

class D : public B
{
   public:
     void display()
       { cout<<"Content of derived class.\n"; }
};

int main()
{
   B *b;
   D d;
   b->display();
   b = &d;    /* Address of object d in pointer variable */
   b->display();
   return 0;
}
```

Note: An object(either normal or pointer) of derived class is type compatible with pointer to base class. So, b = &d; is allowed in above program.

Output

```
Content of base class.
Content of base class.
```

- In above program, even if the object of derived class d is put in pointer to base class,display( ) of the base class is executed(member function of the class that matches the type of pointer).

The example about the rectangle and triangle classes can be written using pointers taking this feature into account:

```
// pointers to base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
};

class Rectangle: public Polygon {
  public:
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    int area()
      { return width*height/2; }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

- Function main declares two pointers to Polygon (named ppoly1 and ppoly2).
- These are assigned the addresses ofrect and trgl, respectively, which are objects of type Rectangle and Triangle.
- Such assignments are valid, since both Rectangle and Triangle are classes derived from Polygon.
- Dereferencing ppoly1 and ppoly2 (with *ppoly1 and *ppoly2) is valid and allows us to access the members of their pointed objects.

- For example, the following two statements would be equivalent in the previous example:

```
ppoly1->set_values (4,5);
rect.set_values (4,5);
```

- But because the type of ppoly1 and ppoly2 is pointer to Polygon (and not pointer to Rectangle nor pointer toTriangle), only the members inherited from Polygon can be accessed, and not those of the derived classesRectangle and Triangle.
- That is why the program above accesses the area members of both objects using rectand trgl directly, instead of the pointers; the pointers to the base class cannot access the area members.
- Member area could have been accessed with the pointers to Polygon if area were a member of Polygon instead of a member of its derived classes, but the problem is that Rectangle and Triangle implement different versions ofarea, therefore there is not a single common version that could be implemented in the base class.

```
#include <iostream>
using namespace std;
class BaseClass {
  int x;
public:
  void setx(int i) {
     x = i;
  }
  int getx() {
     return x;
  }
};

class DerivedClass : public BaseClass {
  int y;
public:
  void sety(int i) {
     y = i;
  }
  int gety() {
     return y;
  }
};

int main()
{
  BaseClass *p;                    // pointer to BaseClass type
  BaseClass baseObject;              // object of BaseClass
  DerivedClass derivedObject;          // object of DerivedClass
```

```
  p = &baseObject;                    // use p to access BaseClass object
  p->setx(10);                        // access BaseClass object
  cout << "Base object x: " << p->getx() << '\n';


  p = &derivedObject;                 // point to DerivedClass object
  p->setx(99);                        // access DerivedClass object


  derivedObject.sety(88);             // can't use p to set y, so do it directly
  cout << "Derived object x: " << p->getx() << '\n';
  cout << "Derived object y: " << derivedObject.gety() << '\n';

  return 0;
}
```

- If there are member functions with same name in base class and derived class, virtual functions gives programmer capability to call member function of different class by a same function call depending upon different context.
- This feature in C++ programming is known as polymorphism which is one of the important feature of OOP.

Dynamic memory

- In the programs seen earlier, all memory needs were determined before program execution by defining the variables needed. But there may be cases where the memory needs of a program can only be determined during runtime.
- For example, when the memory needed depends on user input.
- On these cases, programs need to dynamically allocate memory, for which the C++ language integrates the operators new and delete.

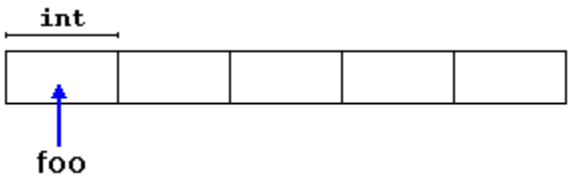
Operators new and new[]

- Dynamic memory is allocated using operator new.
- new is followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets [].
- It returns a pointer to the beginning of the new block of memory allocated. Its syntax is:

  pointer = new type
  pointer = new type [number_of_elements]

- The first expression is used to allocate memory to contain one single element of type type.
- The second one is used to allocate a block (an array) of elements of type type, where number_of_elements is an integer value representing the amount of these.

- For example:
  ```
  int * foo;
  foo = new int [5];
  ```
- In this case, the system dynamically allocates space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to foo (a pointer).
- Therefore, foo now points to a valid block of memory with space for five elements of type int.



- Here, foo is a pointer, and thus, the first element pointed to by foo can be accessed either with the expressionfoo[0] or the expression *foo (both are equivalent).
- The second element can be accessed either with foo[1] or *(foo+1), and so on...
- There is a substantial difference between declaring a normal array and allocating dynamic memory for a block of memory using new.
- The most important difference is that the size of a regular array needs to be a *constant expression*, and thus its size has to be determined at the moment of designing the program, before it is run, whereas the dynamic memory allocation performed by new allows to assign memory during runtime using any variable value as size.
- The dynamic memory requested by our program is allocated by the system from the memory heap. However, computer memory is a limited resource, and it can be exhausted.
- Therefore, there are no guarantees that all requests to allocate memory using operator new are going to be granted by the system.

Operators delete and delete[]

- In most cases, memory allocated dynamically is only needed during specific periods of time within a program; once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory.
- This is the purpose of operator delete, whose syntax is:
  ```
  delete pointer;
  delete[] pointer;
  ```
- The first statement releases the memory of a single element allocated using new, and the second one releases the memory allocated for arrays of elements using new and a size in brackets ([]).
- The value passed as argument to delete shall be either a pointer to a memory block previously allocated with new, or a *null pointer* (in the case of a *null pointer*, delete produces no effect).

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

```
int main ()
{
  int i,n;
  int * p;
  cout << "How many numbers would you like to type? ";
  cin >> i;
  p= new (nothrow) int[i];
  if (p == nullptr)
    cout << "Error: memory could not be allocated";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Enter number: ";
      cin >> p[n];
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
      cout << p[n] << ", ";
    delete[] p;
  }
  return 0;
}
```

- Notice how the value within brackets in the new statement is a variable value entered by the user (i), not a constant expression:
- There always exists the possibility that the user introduces a value for i so big that the system cannot allocate enough memory for it.
- For example, when I tried to give a value of 1 billion to the "How many numbers" question, my system could not allocate that much memory for the program, and I got the text message we prepared for this case (Error: memory could not be allocated).
- It is considered good practice for programs to always be able to handle failures to allocate memory, either by checking the pointer value (if nothrow) or by catching the proper exception.

STATIC FUNCTION

- Static Keyword: Static is a keyword in C++ used to give special characteristics to an element.
- Static elements are allocated storage only once in a program lifetime in static storage area. And they have a scope till the program lifetime.
- Static Keyword can be used with following:
    - Static variable in functions

- Static Class Objects
- Static member Variable in class
- Static Methods in class

Static variables inside Functions

- Static variables when used inside function are initialized only once, and then they hold there value even through function calls.
- These static variables are stored on static storage area , not in stack.

```
void counter()
{
 static int count=0;
 cout << count++;
}
int main(0
{
 for(int i=0;i<5;i++)
 {
   counter();
 }
}
```

Output :
0 1 2 3 4

Let's se the same program's output **without using static** variable.

```
void counter()
{
 int count=0;
 cout << count++;
}
int main(0
{
 for(int i=0;i<5;i++)
 {
   counter();
 }
}
```

Output :
0 0 0 0 0

- If we do not use static keyword, the variable count, is reinitialized everytime when counter() function is called, and gets destroyed each time when counter() functions ends.

- But, if we make it static, once initialized count will have a scope till the end of main() function and it will carry its value through function calls too.
- If you don't initialize a static variable, they are by default initialized to zero.

Static class Objects

- Static keyword works in the same way for class objects too.
- Objects declared static are allocated storage in static storage area, and have scope till the end of program.
- Static objects are also initialized using constructors like other normal objects.
- Assignment to zero, on using static keyword is only for primitive datatypes, not for user defined datatypes.

```
class Abc
{
 int i;
 public:
 Abc()
 {
  i=0;
  cout << "constructor";
 }
 ~Abc()
 {
   cout << "destructor";
 }
};

void f()
{
 static Abc obj;
}

int main()
{
 int x=0;
 if(x==0)
 {
  f();
 }
 cout << "END";
}
```

Output :

constructor END destructor

You must be thinking, why was destructor not called upon the end of the scope of if condition. This is because object was static, which has scope till the program lifetime, hence destructor for this object was called when main() exits.

Static data member in class

- Static data members of class are those members which are shared by all the objects.
- Static data member has a single piece of storage, and is not available as separate copy with each object, like other non-static data members.
- Static member variables (data members) are not initialied using constructor, because these are not dependent on object initialization.
- Also, it must be initialized explicitly, always outside the class. If not initialized, Linker will give error.

```
class X
{
 static int i;
 public:
 X(){};
};

int X::i=1;

int main()
{
 X obj;
 cout << obj.i;   // prints value of i
}
```

- Once the definition for static data member is made, user cannot redefine it.
- Though, arithmetic operations can be performed on it.

Static Member Functions

- These functions work for the class as whole rather than for a particular object of a class.
- It can be called using an object and the direct member access . operator.
- But, its more typical to call a static member function by itself, using class name and scope resolution :: operator.

Example :

```
class X
{
 public:
 static void f(){};
};

int main()
{
```

```
 X::f();   // calling member function directly with class name
}
```

- These functions cannot access ordinary data members and member functions, but only static data members and static member functions.

Points to Remember about Static functions:

1) static member functions do not have this pointer. For example following program fails in compilation with error "`this' is unavailable for static member functions "

```
#include<iostream>
class Test {
  static Test * fun() {
    return this; // compiler error
  }
};

int main()
{
  getchar();
  return 0;
}
```

2) A static member function cannot be virtual

3) Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration. For example, following program fails in compilation with error "'void Test::fun()' and `static void Test::fun()' cannot be overloaded "

```
#include<iostream>
class Test {
  static void fun() {}
  void fun() {} // compiler error
};

int main()
{
  getchar();
  return 0;
}
```

4) A static member function can not be declared const, volatile, *or* const volatile. For example, following program fails in compilation with error "static member function `static void Test::fun()' cannot have `const' method qualifier "

```
#include<iostream>
class Test {
  static void fun() const { // compiler error
    return;
  }
};
int main()
{
  getchar();
  return 0;
}
```

Virtual Function

- A virtual function is a member function that is declared within a base class and redefined by a derived class.
- To create virtual function, precede the function's declaration in the base class with the keyword virtual.
- When a class containing virtual function is inherited, the derived class redefines the virtual function to suit its own needs.
- Base class pointer can point to derived class object.
- In this case, using base class pointer if we call some function which is in both classes, then base class function is invoked.
- But if we want to invoke derived class function using base class pointer, it can be achieved by defining the function as virtual in base class, this is how virtual functions support runtime polymorphism.
- Consider the following program code :

```
class A
{
  int a;
  public:
    A()
    {
      a = 1;
    }
    virtual void show()
    {
      cout <<a;
    }
};
class B: public A
{
  int b;
  public:
    B()
    {
      b = 2;
```

```
    }
    virtual void show()
    {
      cout <<b;
    }
};
int main()
{
  A *pA;
  B oB;
  pA = &oB;
  pA→show();
  return 0;
}
```

Output is 2 since pA points to object of B and show() is virtual in base class A.

- Polymorphism is also achieved in C++ using virtual functions.
- If a function with same name exists in base as well as parent class, then the pointer to the base class would call the functions associated only with the base class.
- However, if the function is made virtual and the base pointer is initialized with the address of the derived class, then the function in the child class would be called.
- Virtual function is the member function of a class that can be overriden in its derived class.
- It is declared with virtual keyword.
- Virtual function call is resolved at run-time (dynamic binding) whereas the non-virtual member functions are resolved at compile time (static binding).
- Describe the virtual function and virtual function table.

A virtual function in C++ is :
- A simple member function of a class which is declared with “virtual” keyword
- It usually performs different functionality in its derived classes.
- The resolving of the function call is done at run-time.

- Virtual Function is a function in base class, which is overrided in the derived class, and which tells the compiler to perform **Late Binding** on this function.
- Virtual Keyword is used to make a member function of the base class Virtual.

Late Binding
- In Late Binding function call is resolved at runtime.
- Hence, now compiler determines the type of object at runtime, and then binds the function call.
- Late Binding is also called **Dynamic** Binding or **Runtime** Binding.

Problem without Virtual Keyword

```
class Base
{
 public:
 void show()
 {
  cout << "Base class";
 }
};
class Derived:public Base
{
 public:
 void show()
 {
  cout << "Derived Class";
 }
}
int main()
{
 Base* b;       //Base class pointer
 Derived d;     //Derived class object
 b = &d;
 b->show();     //Early Binding Ocuurs
}
```

Output : Base class

When we use Base class's pointer to hold Derived class's object, base class pointer or reference will always call the base version of the function

Using Virtual Keyword

- We can make base class's methods virtual by using **virtual** keyword while declaring them.
- Virtual keyword will lead to Late Binding of that method.

```
class Base
{
 public:
 virtual void show()
 {
  cout << "Base class";
 }
};
class Derived:public Base
{
```

```
 public:
 void show()
 {
  cout << "Derived Class";
 }
}
int main()
{
 Base* b;       //Base class pointer
 Derived d;     //Derived class object
 b = &d;
 b->show();     //Late Binding Ocuurs
}
```

Output : Derived class

- On using Virtual keyword with Base class's function,
- Late Binding takes place and the derived version of function will be called, because base class pointer pointes to Derived class object.

Using Virtual Keyword and Accessing Private Method of Derived class

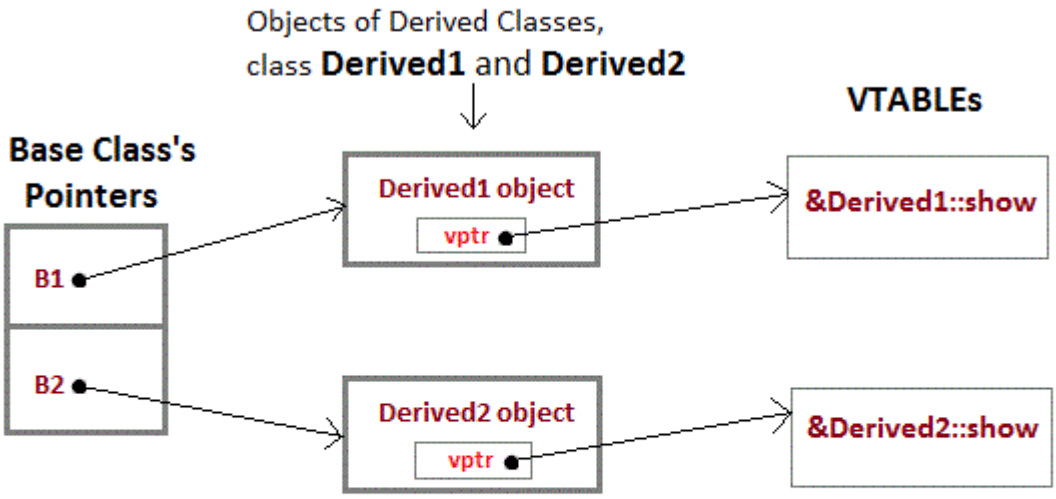
- We can call **private** function of derived class from the base class pointer with the help of virtual keyword.
- Compiler checks for access specifier only at compile time.
- So at run time when late binding occurs it does not check whether we are calling the private function or public function.

```
#include
using namespace std;
class A
{
   public:
   virtual void show()
   {
      cout << "Base class\n";
   }
};
class B: public A
{
private:
   virtual void show()
   {
      cout << "Derived class\n";
   }
};
int main()
```

```
{
   A *a;
   B b;
   a = &b;
   a -> show();
}
```

Output : Derived class

Mechanism of Late Binding



vptr, is the vpointer, which points to the Virtual Function for that object.

VTABLE, is the table containing address of Virtual Functions of each class.

- To accomplish late binding, Compiler creates **VTABLEs**, for each class with virtual function.
- The address of virtual functions is inserted into these tables.
- Whenever an object of such class is created the compiler secretly inserts a pointer called **vpointer**, pointing to VTABLE for that object.
- Hence when function is called, compiler is able to resovle the call by binding the correct function using the vpointer.

Important Points to Remember

- Only the Base class Method's declaration needs the **Virtual** Keyword, not the definition.
- If a function is declared as **virtual** in the base class, it will be virtual in all its derived classes.

- The address of the virtual Function is placed in the **VTABLE** and the copiler uses **VPTR**(vpointer) to point to the Virtual Function.

C++ Pure Virtual Function:
- It is a virtual function that does not have any implementation part.
- It has only declaration part.
- It is declared by assigning the '0' in the declaration part.

Example:
```
class A
{
  public:
     virtual void pureVirtual() = 0;   // Pure Virtual Function
       // There is no function body
};
```

- This function is also known as, pure specifier.
- The "= 0" notation indicates that the virtual function is a pure virtual function and it has no function body or definition.

Why don't we have virtual constructors?

- A virtual call is a mechanism to get work done given partial information.
- In particular, "virtual" allows us to call a function knowing only any interfaces and not the exact type of the object. To create an object you need complete information.
- In particular, you need to know the exact type of what you want to create.
- Consequently, a "call to a constructor" cannot be virtual.
- Virtual functions basically provide polymorphic behavior.
- That is, when you work with an object whose dynamic type is different than the static (compile time) type with which it is referred to, it provides behavior that is appropriate for the *actual* type of object instead of the static type of the object.
- Now try to apply that sort of behavior to a constructor. When you construct an object the static type is always the same as the actual object type since:
- To construct an object, a constructor needs the exact type of the object it is to create [...] Furthermore [...]you cannot have a pointer to a constructor

Virtual Destructor

- Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behavior.
- To correct this situation, the base class should be defined with a virtual destructor.
- For example, following program results in undefined behavior.
- Although the output of following program may be different on different compilers, when compiled using Dev-CPP, it prints following.

```
Constructing base
Constructing derived
Destructing base
```

```
// A program without virtual destructor causing undefined behavior
#include<iostream>
 using namespace std;
 class base {
  public:
    base()
    { cout<<"Constructing base \n"; }
    ~base()
    { cout<<"Destructing base \n"; }
};
 class derived: public base {
  public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};
 int main(void)
{
  derived *d = new derived();
  base *b = d;
  delete b;
```

```
  getchar();
  return 0;
}
```

- Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called.
- For example, following program prints:

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

```
// A program with virtual destructor
#include<iostream>
 using namespace std;
 class base {
  public:
    base()
    { cout<<"Constructing base \n"; }
    virtual ~base()
    { cout<<"Destructing base \n"; }
};
 class derived: public base {
  public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
```

```
};

 int main(void)

{

  derived *d = new derived();

  base *b = d;

  delete b;

  getchar();

  return 0;

}
```

C++ friend Function and friend Classes

One of the important concept of OOP is data hiding, i.e., a nonmember function cannot access an object's private or protected data.

But, sometimes this restriction may force programmer to write long and complex codes.

So, there is mechanism built in C++ programming to access private or protected data from non-member function which is friend function and friend class.

friend Function in C++

If a function is defined as a friend function then, the private and protected data of class can be accessed from that function.

The complier knows a given function is a friend function by its keyword friend.

The declaration of friend function should be made inside the body of class (can be anywhere inside class either in private or public section) starting with keyword friend.

```
class class_name
{
   ..... .... ........
   friend return_type function_name(argument/s);
   ..... .... ........
}
```

Now, you can define friend function of that name and that function can access the private and protected data of that function.

No keywords in used in function definition of friend function.

```
class class_name
{
    ......  ....   ........
    friend return_type function_name(argument/s);
    ......  ....   ........
}
...... ..... ......
return_type function_name(argument/s)
{
..... ....... ....
/* Private and protected data of the above class can be accessed from
this function because, this function is a friend function of above class*/
..... ....... ....
}
```

Example to Demonstrate working of friend Function

```
/* C++ program to demonstrate the working of friend function.*/
#include <iostream>
using namespace std;
class Distance
{
   private:
      int meter;
   public:
      Distance(): meter(0){ }
      friend int func(Distance);  //friend function
};
int func(Distance d)           //function definition
{
   d.meter=5;        //accessing private data from non-member function
   return d.meter;
}
int main()
{
   Distance D;
   cout<<"Distace: "<<func(D);
   return 0;
}
```

Output
Distance: 5

Here, friend function func() is declared inside Distance class.
So, the private data can be accessed from this function.
Though this example gives you what idea about the concept of friend function, this program doesn't give you idea about when friend function is helpful.

Suppose, you need to operate on objects of two different class then, friend function can be very helpful.
You can operate on two objects of different class without using friend function but, you program will be long, complex and hard to understand.
Example to operate on Objects of two Different class using friend Function

```
#include <iostream>
using namespace std;
class B;     // forward declaration
class A {
   private:
     int data;
   public:
     A(): data(12){ }
     friend int func(A , B);   //friend function Declaration
};
class B {
   private:
      int data;
   public:
      B(): data(1){ }
      friend int func(A , B);  //friend function Declaration
};
int func(A d1,B d2)
/*Function func() is the friend function of both classes A and B. So, the private data of both class can be accessed from this function.*/
{
  return (d1.data+d2.data);
}
int main()
   {
      A a;
      B b;
      cout<<"Data: "<<func(a,b);
      return 0;
   }
```

In this program, classes A and B has declared func() as a friend function.
Thus, this function can access private data of both class.
In this program, two objects of two different class A and B are passed as an argument to friend function.
Thus, this function can access private and protected data of both class. Here, func() function adds private data of two objects and returns it to main function.
To work this program properly, a forward declaration of a class should be made as in above example(forward declaration of class B is made).
It is because class B is referenced from class A using code: friend int func(A , B);.

So, class A should be declared before class B to work properly.

friend Class in C++ Programming

Similarly like, friend function, A class can be made a friend of another class using keyword friend.

For example:

```
..... ..... .....
class A{
  friend class B;      // class B is a friend class
  ..... ..... .....
}
class B{
  ..... ..... .....
}
```

When a class is made a friend class, all the member functions of that class becomes friend function.

In this program, all member functions of class B will be friend function of class A.

Thus, any member function of class B can access the private and protected data of class A.

If B is declared friend class of A then, all member functions of class B can access private data and protected data of class A but, member functions of class A cannot private and protected data of class B.

Remember, friendship relation in C++ is granted not taken.