Introduction to C++

- Method Overriding
- Difference between method overloading & overriding
- Upcasting
- Dynamic Memory Allocation
    - new operator
    - delete operator

Method Overriding

- If a subclass has the same method as declared in the parent class, it is known as Method Overriding.
- When a method in a sub class has same name and type signature as a method in its super class, then the method is known as overridden method.
- The key benefit of overriding is the ability to define method that's specific to a particular subclass type.

Advantages of Method Overriding:

- Method overriding is used to provide specific implementation of a method that is already provided by its super class.

Rules for Method Overriding:

- Method in sub class must have same name as in the parent class.
- Method in sub class must have same parameter as in the parent class.
- Must be IS-A relationship (inheritance).

Method Overriding

Example:

```
#include<iostream>
using namespace std;
class Bank
{
		public:
		int getInterestRate(){return 0;}
};
class Canara:public Bank
{
		public:
		int getInterestRate(){return 5;}
};
class Axis:public Bank
{
		public:
		int getInterestRate(){return 7;}
};
```

Method Overriding

```
int main()
{
        Bank b;
        Canara c;
        Axis a;
        cout<<"Bank rate of Interest:"<<b.getInterestRate()<<endl;
        cout<<"Canara bank rate of Interest:"<<c.getInterestRate()<<endl;
        cout<<"Axis bank rate of Interest:"<<a.getInterestRate()<<endl;
}
```

Method Overriding

Accessing the Overridden Function in Base Class From Derived Class:

To access the overridden function of base class from derived class, scope resolution operator :: is used.

For example: If we want to access overriden get_data() function of base class from derived class then, the following statement is used in derived class get_data() function.

```
A::get_data;   // Calling get_data() of class A.
```

Method Overriding

Example of Accessing the Overridden Function in Base Class From Derived Class:

```
#include<iostream>
using namespace std;
class Super
{           public:
            void display()
            {           cout << "Super function‟<<endl;       }
};
class Sub : public Super
{           public:
            void display()
            {           Super::display();
                        cout << "Sub function‟<<endl;
            }
};
int main()
{           Sub s;
            s.display();
}
```

Output:

```
Super function
Sub function
```

Difference between Method overloading & method overriding

Method Overloading

- It is used to increase the readability of the program.
- Parameters must be different, in case of method overloading
- Method overloading can't be performed by changing return type of the method only.
- Method Overloading does not require more than one class for overloading.

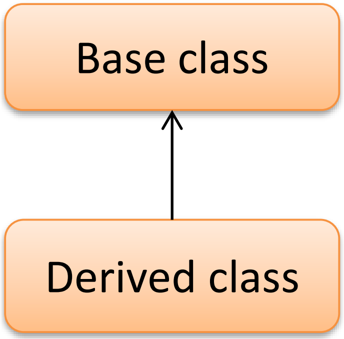
Method Overriding

- It is used to provide the specific implementation of the method that is already provided by its super class.
- Parameters must be same,in case of method overriding,
- Return type must be same in method overriding.
- Method Overriding requires at least two classes for overriding.

Upcasting

Upcasting:

Upcasting is using the Super class's reference or pointer to refer to a Sub class's object. Or we can say that, the act of converting a Sub class's reference or pointer into its Super class's reference or pointer is called Upcasting.



Upcasting

Example:

```
#include<iostream>
using namespace std;
class Super
{           public:
            void funBase() { cout << "Super function“<<endl; }
};
class Sub : public Super
{           public:
            void funBase() { cout << "Sub function“<<endl; }
};
int main()
{ Super* ptr;      // Super class pointer
 Sub obj;
 ptr = &obj;
 ptr->funBase();
 Super &ref=obj;   // Super class's reference
 ref.funBase();
}
```

Output:

```
Super function
Super function
```

Dynamic memory allocation

Dynamic Memory Allocation:

- Memory allocated "on the fly" during run time.
- dynamically allocated space usually placed in a program segment known as the heap or the free store.
- Exact amount of space or number of items does not have to be known by the compiler in advance.
- For dynamic memory allocation, pointers are crucial.

Dynamic allocation requires two steps:

- Creating the dynamic space.
- Storing its address in a pointer (so that the space can be accessed).

To dynamically allocate memory in C++, we use the **new** operator.

De-allocation:

- Deallocation is the "clean-up" of space being used for variables or other data storage.
- Compile time variables are automatically deallocated based on their known extent (this is the same as scope for "automatic" variables).
- It is the programmer's job to deallocate dynamically created space.
- To de-allocate dynamic memory, we use the **delete** operator.

Dynamic memory allocation

Allocating space with new:

We can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

Syntax :

data_type pointer_var=new data_type;

Example:

int *a=new int(10);

If the call to the new operator is successful,it returns a pointer to the space that is allocated.otherwise it returns the address zero if the space could not be found or if some kind of error is detected.

Dynamic memory allocation

Deallocating space with delete:

The delete operator is used to destroy the variable space which has been created by using the new operator dynamically.

Syntax:

delete pointer;

Dynamic memory allocation

Example:

```
#include<iostream>
using namespace std;
int main()
{
   int *ptr_i=new int(25);
   float *ptr_f=new float();
   cout<<"enter float no.";
   cin>>*ptr_f;
   cout<<"integer is "<<*ptr_i<<endl;
   cout<<"float is "<<*ptr_f;
   delete ptr_i;
   delete ptr_f;
}
```

Dynamic memory allocation

Example of dynamic memory allocation for Array:

```
#include<iostream>
using namespace std;
int main()
{ int x;
  cout<<"enter no. of elements"<<endl;
  cin>>x;
  int *ptr=new int[x];
  for(int i=0;i<x;i++)
  {
          cout<<"enter element";
          cin>>ptr[i];
  }
  cout<<"elements are:"<<endl;
  for(int i=0;i<x;i++)
  {
          cout<<ptr[i]<<endl;
  }
  delete[] ptr;
}
```

Dynamic Memory allocation

Example of Dynamic memory allocation for objects:

```
#include <iostream>
using namespace std;
class Box
{ public:
Box()
{   cout << "Constructor called!" <<endl;   }
~Box()
{   cout << "Destructor called!" <<endl;    }
};
int main( )
{
Box* myBoxArray = new Box[2];
delete [] myBoxArray;         // Delete array of objects
}
```

Output:

```
Constructor called!
Constructor called!
Destructor called!
Destructor called!
```

Discussions