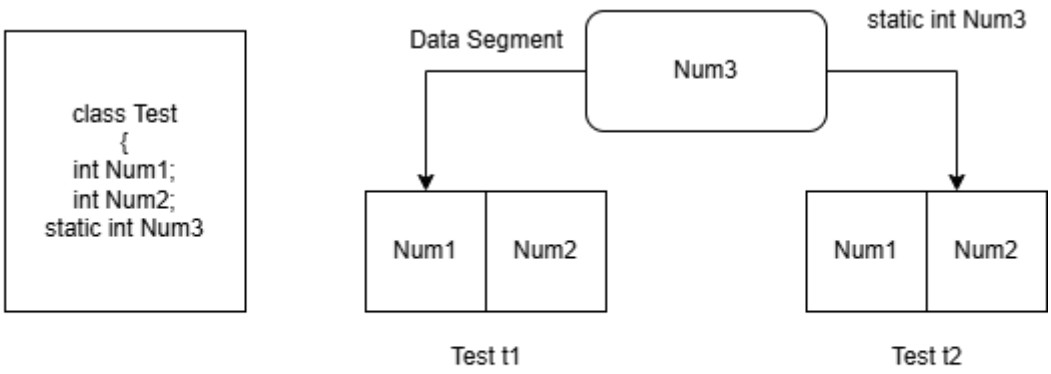
C++ Notes Day-8 Date: 13 June 2025

Revision

- Friend Function and Friend Class

Static Data Member



this-> it store address of current object/instance

- If we want to share value of any data member in all the objects of same class then we should declare data member static.
- Static data member get space during program loading once per class on data segment.
- if we create object of the class then only non static data member get space inside it. Hence size of object depends on size of all the data members declared inside class.
- Data member of the class which get space inside object is called as instance variable. In other words non static member is also called as instance variable.
- Instance variable get space once per object.
- To access instance variable either we should use object or pointer/reference to that object.
- Data member of the class which do not get space inside object is called as class level variable. In other words static member is also called as class level variable.
- Class level variable get space once per class.
- To access class level variable we should class name and :: operator.
- Example 1:

```
class A{
 int n1;
 int n2;
 static int count;
};
int main( void ){
 A a1,a2,a3;
 return 0
}
```

- Example 2:

```
class B{
 int n3;
 int n4;
 static int count;
};
int main( void ){
 B b1,b2,b3;
 return 0
}
```

- Example 3:

```
class C{
 int n5;
 int n6;
 static int count;
};
int main( void ){
 C t1,t2,t3;
 return 0
}
```

- if we want to declare data member static then we must provide global definition for it. Otherwise linker will generate error.

```
 #include<iostream>
 using namespace std;
 class Test{
 public:
 int num1; //Instance variable
 int num2; //Instance variable
 static int num3; //Class level variable
 public:
 Test( void ){
 this->num1 = 0;
 this->num2 = 0;
 }
 Test( int num1, int num2 ){
 this->num1 = num1;
 this->num2 = num2;
 }
 void printRecord( void ){
 cout << "Num1 : " << this->num1 << endl;
 cout << "Num2 : " << this->num2 << endl;
 cout << "Num3 : " << Test::num3 << endl;
 }
```

```
};
int Test::num3 = 500; //Global definition
int main( void ){
Test t1;
t1.printRecord( );
return 0;
}
```

- We can declare static data member constant. Consider following code:

```
#include<iostream>
using namespace std;
class Test{
public:
int num1; //Instance variable
int num2; //Instance variable
const static int num3; //Class level variable
public:
Test( void ){
this->num1 = 0;
this->num2 = 0;
}
Test( int num1, int num2 ){
this->num1 = num1;
this->num2 = num2;
}
void printRecord( void ){
cout << "Num1 : " << this->num1 << endl;
cout << "Num2 : " << this->num2 << endl;
cout << "Num3 : " << Test::num3 << endl;
}
};
const int Test::num3 = 500; //Global definition
int main( void ){
Test t1;
t1.printRecord( );
return 0;
}
```

Static Member Function

- To access non static members of the class, we should define non static member function inside class.
- Non static member functions are designed to call on object. Hence it is also called as instace method.
- Since non static member functions / instance methods are designed to call on object/instance, it gets this pointer. Since non static member function get this pointer, we can access static as well as non static members inside non static member function.
- To access static member of the class, we should define static member function inside class.
- Static member functions are designed to call on class name. Hence it is also called as class level method.

- Since static member functions / class level methods are designed to call on class name, it doesn't get this pointer. Since static member function doesn't get this pointer, we can access only static members inside static member function.
- Static member function do not get this pointer but we can create object inside static member function.
- Using object, we can access non static members inside static member function.
- Example 1:

```
#include<iostream>
 using namespace std;
 class Test{
 private:
 int num1; //Non static data member / Instance variable
 int num2; //Non static data member / Instance variable
 static int num3; //Static data member / Class level variable
 public:
 Test( void ){
 this->num1 = 0;
 this->num2 = 0;
 }
 void setNum1( int num1 ){
 this->num1 = num1;
 }
 void setNum2( int num2 ){
 this->num2 = num2;
 }
 static void setNum3( int num3 ){
 Test::num3 = num3;
 }
 void printRecord( void ){
 cout << "Num1 : " << this->num1 << endl;
 cout << "Num2 : " << this->num2 << endl;
 cout << "Num3 : " << Test::num3 << endl;
 }
 };
 int Test::num3 = 0; //Global definition
 int main( void ){
 Test t1;
 t1.setNum1( 10 );
 t1.setNum2( 20 );
 Test::setNum3( 30 );
 t1.printRecord( );
 return 0;
 }
 int main( void ){
 Test t1;
 //t1.num1 = 10; //error: 'num1' is a private member of 'Test'
 //t1.num2 = 20; //error: 'num2' is a private member of 'Test'
 //Test::num3 = 30;//error: 'num3' is a private member of 'Test'
 t1.printRecord( );
 return 0;
 }
```

- Example 2:

```
class Test{
 private:
 int num1;
 static int num2;
 public:
 Test( void ) : num1( 10 ){
 }
 static void print( void ){
 Test t;
 cout << "Num1 : " << t.num1 << endl;
 cout << "Num2 : " << Test::num2 << endl;
 }
};
int Test::num2 = 20; //Global definition
int main( void ){
 Test::print( );
 return 0;
}
```

- Why static member function do not get this pointer?
  - If we call non static member function on object then non static member function get this pointer.
  - Static member function is designed to call on class name.
  - Since static member function is not designed to call on object, it does not get this pointer.
- Can we declare static member function constant?
  - If we dont want to modify state on only current object inside member function then we should declare that member function constant. In other words, constant member functions are designed to call on object.
  - Static member function is designed to call on class name.
  - Since static member function is not designed to call on object, we can not declare static member function constant. Conclusion
  - In C++, we can declare static data member constant but we can not declare static member function constant.
  - If there is need to use this pointer inside member function then it should be non static otherwise it should be static.
- Example:

```
 #include<iostream>
 using namespace std;
 class Math{
 public:
 static const double PI;
 public:
 static double pow( double base, int index ){
 double result = 1;
 for( int count = 1; count <= index ; ++ count )
```

```
result = base * result;
return result;
}
};
const double Math::PI = 3.14;
int main( void ){
double result = Math::pow( 2.0, 3 );
cout << "Result : "<<result <<endl;
return 0;
}
```

- We can not declare below functions static:
  - constructor
  - destructor
  - constant member function
  - volatile member function
  - virtual member function
  - main function (other global functions can be static)
- How will you write code to count number of instances created from class?

```
#include<iostream>
using namespace std;
class InstanceCounter{
private:
static int count;
public:
InstanceCounter( ){
InstanceCounter::count = InstanceCounter::count + 1;
}
static int getCount( void ){
return InstanceCounter::count;
}
~InstanceCounter( ){
InstanceCounter::count = InstanceCounter::count - 1;
}
};
int InstanceCounter::count = 0;
int main( void ){
InstanceCounter t1, t2, t3;
cout << "Instance Counter : " << InstanceCounter::getCount( ) <<endl;
return 0;
}
```

- Anonymous class: A class defined without name is known as Anonymous class.

```
#include<iostream>
using namespace std;
class{ //Anonymous class
```

```
public:
 void showRecord( void ){
 cout << "void showRecord( void )" << endl;
 }
 static void displayRecord( void ){
 cout << "static void displayRecord( void )" << endl;
 }
}t1;
int main( void ){
 t1.showRecord( );
 t1.displayRecord( );
 return 0;
}
```

Operator Overloading in C++

- Token
  - Token is a basic unit of a program.
  - Classification of tokens:
    - Identifier
    - Keywords
    - Constant
    - Operator
    - Separator / punctuators
  - Classfication of Operators
    - Unary Operators
      - An operator which requires only one operand(e.g sizeof(a)) is called as unary operator.
      - Example: sizeof, typeid, ++, --, !(Logical NOT), ~, +, -, * etc
    - Binary Operators
      - An operator which requires two operands(e.g a + b) is called as binary operator.
      - Arithmetic operators +, -, *, /, %
      - Relational operators <, <= >, >=, ==, !=
      - Logical operators &&(Logical AND), || (Logical OR),
      - Bitwise operators &(Bitwise AND), | (Bitwies OR), ^(Bitwise XOR), <<, >>
      - Assignment operators =, short hand operators(+=, -=, *= etc): X+=Y=>X=X+Y
    - Ternary Operators
      - An operator which requires three operands is called as ternary operator.
      - Conditional operator(? :)
- Example 1: We can use operators with variables of fundamental data types

```
int main( void ){
 int num1 = 10;
 int num2 = 20;
 int result = num1 + num2; //OK
 return 0;
}
```

- Example 2: In C, we cannot use operator with objects of user defined type.

```
struct Point{
 int xPos;
 int yPos;
};
int main( void ){
 struct Point pt1 = { 10, 20 }; //OK
 struct Point pt2 = { 30, 40 }; //OK
 struct Point pt3; //OK
 pt3.xPos = pt1.xPos + pt2.xPos; //OK
 pt3.yPos = pt1.yPos + pt2.yPos; //OK
 return 0;
}
int main1( void ){
 struct Point pt1 = { 10, 20 }; //OK
 struct Point pt2 = { 30, 40 }; //OK
 struct Point pt3; //OK
 pt3 = pt1 + pt2; //Not OK
 return 0;
}
```

- Consider code in C++ programming language:
- Example 1: In C++, we can use operator with the variables of fundamental types.

```
int main( void ){
 int num1 = 10;
 int num2 = 20;
 int result = num1 + num2; //OK
 return 0;
}
```

- Example 2: In C++, we cannot use operator with objects of user defined type directly.

```
 #include<iostream>
 using namespace std;
 class Test{
 private:
 int Num1;
 int Num2;
 public:
 Test( void ){
 this->Num1 = 0;
 this->Num2 = 0;
 }
 Test( int Num1, int Num2 ){
 this->Num1 = Num1;
```

```
this->Num2 = Num2;
}
void printRecord( void ){
cout << "Num1 Number : " << this->Num1 <<endl;
cout << "Num2 Number : " << this->Num2 <<endl;
}
};
int main( void ){
Test t1( 10, 20 );
Test t2( 30, 40 );
Test t3;
t3 = t1 + t2; //error: invalid operands to binary expression
('Test' and 'Test')
t2.printRecord( );
return 0;
}
```

- If we want to use operator with the objects of user defined type(structure, class etc.) then we should overload operator.
- To overload operator, we should define operator function.
- operator is keyword in C++ which is used to define operator function.
- We can define operator function using 2 ways:
    - Member function
    - Non member function
- By defining operator function, we are increasing capability of exsiting operators. This process of givining extension to the meaning of the operator is called as operator overloading.
- Example: Addition (+) Operator Overloading using member function

```
#include<iostream>
using namespace std;
class Test{
private:
int Num1;
int Num2;
public:
Test( void ){
this->Num1 = 0;
this->Num2 = 0;
}
Test( int Num1, int Num2 ){
this->Num1 = Num1;
this->Num2 = Num2;
}
//Test other = t2
//Test *const this = &t1
Test operator+( Test other ){
Test result;
result.Num1 = this->Num1 + other.Num1;
result.Num2 = this->Num2 + other.Num2;
return result;
```

```
}
void printRecord( void ){
cout << "Num1 Number : " << this->Num1 <<endl;
cout << "Num2 Number : " << this->Num2 <<endl;
}
};
int main( void ){
Test t1( 10, 20 );
Test t2( 30, 40 );
Test t3;
t3 = t1 + t2; // t3 = t1.operator+( t2 )
t3.printRecord( );
return 0;
}
```

- Example: Addition (+) Operator Overloading using non member function

```
#include<iostream>
using namespace std;
class Test{
private:
int Num1;
int Num2;
public:
Test( void ){
this->Num1 = 0;
this->Num2 = 0;
}
Test( int Num1, int Num2 ){
this->Num1 = Num1;
this->Num2 = Num2;
}
void printRecord( void ){
cout << "Num1 Number : " << this->Num1 <<endl;
cout << "Num2 Number : " << this->Num2 <<endl;
}
friend Test operator+( Test t1, Test t2 );
};
Test operator+( Test t1, Test t2 ){
Test result;
result.Num1 = t1.Num1 + t2.Num1;
result.Num2 = t1.Num2 + t2.Num2;
return result;
}
int main( void ){
Test t1( 10, 20 );
Test t2( 30, 40 );
Test t3;
t3 = t1 + t2; // t3 = operator+( t1, t2 )
t3.printRecord( );
return 0;
}
```

- Using operator overloading we can not create user defined operators rather we can increase capability of existing operators.
- Limitations of operator overloading
    - We can not overload below operators using member function as well as non member function
        - dot(.) / meber selection operator
        - .* (pointer to member selection operator)
        - sizeof operator
        - :😦 scope resolution operator)
        - Conditional (? :) / Ternary operator
        - typeid operator
        - static_cast operator
        - dyanmic_cast operator
        - const_cast operator
        - reinterpret_cast operator
    - We can not overload below operators using non member function but we can overload it using member function
        - Assignment operator(=)
        - Index / subscript operator
        - Call / Function call operator[()]
        - Arrow(->) operator
- using operator overloading, we can change meaning of the operator.

Arithmatic Operator Overloading

- Example: Subtraction (-) Operator Overloading using member function

```
#include<iostream>
 using namespace std;
 class Test{
 private:
 int Num1;
 int Num2;
 public:
 Test( void ){
 this->Num1 = 0;
 this->Num2 = 0;
 }
 Test( int Num1, int Num2 ){
 this->Num1 = Num1;
 this->Num2 = Num2;
 }
 //Test other = t2
 //Test *const this = &t1
 Test operator-( Test other ){
 Test result;
 result.Num1 = this->Num1 - other.Num1;
 result.Num2 = this->Num2 - other.Num2;
```

```
return result;
}
void printRecord( void ){
cout << "Num1 Number : " << this->Num1 <<endl;
cout << "Num2 Number : " << this->Num2 <<endl;
}
};
int main( void ){
Test t1( 10, 20 );
Test t2( 30, 40 );
Test t3;
t3 = t1 - t2; // t3 = t1.operator-( t2 )
t3.printRecord( );
return 0;
}
```

- Example: Subtraction (-) Operator Overloading using non member function

```
#include<iostream>
using namespace std;
class Test{
private:
int Num1;
int Num2;
public:
Test( void ){
this->Num1 = 0;
this->Num2 = 0;
}
Test( int Num1, int Num2 ){
this->Num1 = Num1;
this->Num2 = Num2;
}
void printRecord( void ){
cout << "Num1 Number : " << this->Num1 <<endl;
cout << "Num2 Number : " << this->Num2 <<endl;
}
friend Test operator-( Test t1, Test t2 );
};
Test operator-( Test t1, Test t2 ){
Test result;
result.Num1 = t1.Num1 - t2.Num1;
result.Num2 = t1.Num2 - t2.Num2;
return result;
}
int main( void ){
Test t1( 10, 20 );
Test t2( 30, 40 );
Test t3;
t3 = t1 - t2; // t3 = operator-( t1, t2 )
t3.printRecord( );
```

```
return 0;
}
```

- Calling Syntax: Arithmetic operator overloading
- Example 1:

```
Test t1( 10, 20 );
Test t2( 40, 30 );
Test t3;
t3 = t1 + t2; //t3 = t1.operator+( t2 ); //Using member function
```

- Example 2:

```
Test t1( 10, 20 );
Test t2( 40, 30 );
Test t3;
t3 = t1 + t2; //t3 = operator+( t1, t2 ); //Using non member function
```

- Example 3:

```
Test t1( 10, 20 );
Test t2( 40, 30 );
Test t3;
t3 = t1 - t2; //t3 = t1.operator-( t2 ); //Using member function
```

- Example 4:

```
Test t1( 10, 20 );
Test t2( 40, 30 );
Test t3;
t3 = t1 - t2; //t3 = operator-( t1, t2 ); //Using non member function
```

- Example 5:

```
Test t1( 10, 20 );
Test t2( 40, 30 );
Test t3;
t3 = t1 * t2; //t3 = t1.operator*( t2 ); //Using member function
```

- Example 6:

```
Test t1( 10, 20 );
Test t2( 40, 30 );
Test t3;
t3 = t1 * t2; //t3 = operator*( t1, t2 ); //Using non member function
```

- Example 7:

```
Test t1( 10, 20 );
Test t2( 40, 30 );
Test t3;
t3 = t1 / t2; //t3 = t1.operator/( t2 ); //Using member function
```

- Example 8:

```
Test t1( 10, 20 );
Test t2( 40, 30 );
Test t3;
t3 = t1 / t2; //t3 = operator/( t1, t2 ); //Using non member function
```

- Example and Calling Syntax: Relational Operator overloading
- Example-1 (Overloading of == operator)

```
#include <iostream>
using namespace std;
class Test
{
public:
    int Num1;
    int Num2;
    Test()
    {

    }
    Test(int Num1, int Num2)
    {
        this->Num1=Num1;
        this->Num2=Num2;
    }
    void ShowData()
    {
        cout<<"Num1:    "<<this->Num1<<" Num2:  "<<this->Num2<<endl;
    }
};

bool operator==(Test obj1, Test obj2)
    {
```

```
        if(obj1.Num1==obj2.Num1&&obj1.Num2==obj2.Num2)
        {
            return true;
        }
        return false;
    }
int main()
{
    Test t1(10,20);
    Test t2(10,20);

    bool status=t1==t2; //OK

    cout<<status;
    return 0;
}
```

- Example 1:

```
Test t1( 10, 20 );
Test t2( 10, 20 );
bool status = t1 == t2; //status = t1.operator==( t2 ); //Using member function
```

- Example 2:

```
Test t1( 10, 20 );
Test t2( 10, 20 );
bool status = t1 == t2; //status = operator==( t1, t2 ); //Using non member
function
```

- Example 3:

```
Test t1( 10, 20 );
Test t2( 10, 20 );
bool status = t1 != t2; //status = t1.operator!=( t2 ); //Using member function
```

- Example 4:

```
Test t1( 10, 20 );
Test t2( 10, 20 );
bool status = t1 != t2; //status = operator!=( t1, t2 ); //Using non member
function
```

- Example 5:

```
Test t1( 10, 20 );
Test t2( 10, 20 );
bool status = t1 < t2; //status = t1.operator<( t2 ); //Using member function
```

- Example 6:

```
Test t1( 10, 20 );
Test t2( 10, 20 );
bool status = t1 < t2; //status = operator<( t1, t2 ); //Using non member function
```

- Example 7:

```
Test t1( 10, 20 );
Test t2( 10, 20 );
bool status = t1 > t2; //status = t1.operator>( t2 ); //Using member function
```

- Example 8:

```
Test t1( 10, 20 );
Test t2( 10, 20 );
bool status = t1 > t2; //status = operator>( t1, t2 ); //Using non member function
```

- Calling Syntax: Unary operator overloading
- Example 1:

```
Test t1( 10, 20 );
Test t2 = ++ t1; //t2 = t1.operator++( ); //Using member function
```

- Example 2:

```
Test t1( 10, 20 );
Test t2 = ++ t1; //t2 = operator++( t1 ); //Using non member function
```

- Example 3:

```
Test t1( 10, 20 );
Test t2 = t1 ++; //t2 = t1.operator++( 0 ); //Using member function
```

- Example 4:

```
Test t1( 10, 20 );
Test t2 = t1 ++; //t2 = operator++( t1, 0 ); //Using non member function
```

- Extraction operator overloading
    - cin: character input, which represents keyboard.
    - extraction operator(>>) is designed to use with cin.
    - If we want to accept state the object from keybaord using cin then we should overload extraction operator.
- Consider call using member function:

```
Test t1;
cin >> t1; //cin.operator>>( t1 );
```

- Here to accept record for t1, we should define operator>>() function inside istream class, which is not recommended. So we will not overload it using member function.
- Consider call using non member function:

```
Test t1;
cin >> t1; //operator>>( cin, t1 );
```

- Here to accept record for t1, we should define operator>>() function global, which is possible for us. Hence we will overload operator >> using non member function.
- Consider another example:

```
Test t1;
Test t2;
cin >> t1 >> t2; //operator>>( operator>>( cin, t1 ), t2 );
```

- General Syntax:

```
class ClassName{
friend istream& operator>>( istream &cin, ClassName &other );
};
istream& operator>>( istream &cin, ClassName &other ){
//TODO: accept record using other reference variable
return cin;
}
```

- Insertion operator overloading

    - cout: character output, which represents monitor.
    - insertion operator(<<) is designed to use with cout.
    - If we want to print state the object on monitor using cout then we should overload insertion operator.
- Consider call using member function:

```
Test t1(10,20);
cout << t1; //cout.operator<<( t1 );
```

- Here to print record of t1, we should define operator<<() function inside ostream class, which is not recommended. So we will not overload it using member function.
- Consider call using non member function:

```
Test t1(10,20);
cout << t1; //operator<<( cout, t1 );
```

- Here to print record of t1, we should define operator<<() function global, which is possible for us. Hence we will overload operator << using non member function.
- Consider another example:

```
Test t1( 10, 20 );
Test t2( 30, 40 );
cout << t1 << t2; //operator<<( operator<<( cout, t1 ), t2 );
```

- General Syntax:

```
class ClassName{
friend ostream& operator<<( ostream &cout, ClassName &other );
};
ostream& operator<<( ostream &cout, ClassName &other ){
//TODO: print record using other reference variable
return cout;
}
```

```
#include <iostream>
using namespace std;
class Test
{
private:
    int Num1;
    int Num2;
public:
```

```
    Test()
    {

    }
    Test(int Num1, int Num2)
    {
        this->Num1=Num1;
        this->Num2=Num2;
    }

    void ShowData()
    {
        cout<<"Num1:    "<<this->Num1<<" Num2:  "<<this->Num2<<endl;
    }


    friend ostream& operator<<(ostream& cout, Test &ref);
};

ostream& operator<<(ostream& cout, Test &ref)
{
    cout<<"Num1:    "<<ref.Num1<<endl;
    cout<<"Num2:    "<<ref.Num2<<endl;

    return cout;
}

int main()
{
    Test t1(10,20);
    cout<<t1;
    return 0;
}
```

- Kinldy do it by yourself: Index operator overloading, Function call operator overloading
- Assignment operator overloading
- Example 1:

```
int num1 = 10; //Initialization
int num2 = num1; //Initialization
```

- Process of storing value during declaration of variable is called as initialization.
- Example 2:

```
Test t1( 10, 20 ); //on t2 parameterized constructor will call
Test t2 = t1; //on t2 copy constructor will call.
Test t3( t1 ); //on t3 copy constructor will call.
```

- If we "initialize" object from another object of same class the copy constructor gets called.
- Syntax:

```
class ClassName{
public:
ClassName( const ClassName &other ){
//TODO: Shallow / Deep Copy
}
}
```

- Example 3:

```
int num1 = 10;
int num2;
num2 = num1; //Assignment
```

- Process of storing value after declaration of variable is called as assignment.
- Example 4:

```
Test t1( 10, 20 ); //on t2 parameterized constructor will call
Test t2; //on t2 parameterless constructor will call.
t2 = t1; //t2.operator=( t1 );
```

- If we assign object to another object of same class the assignment operator function gets called.
- If we do not define assignment operator function for the class then compiler generates one assignment operator function for the class by default, it is called as default assignment operator function.
- Default copy constructor and default assignment operator function by default creates shallow copy.
- Example 5:use "-fno-elide-constructors" compiler option

```
Test t1( 10, 20 ); //on t2 parameterized constructor will call
Test t2; //on t2 parameterless constructor will call.
Test t3; //on t3 parameterless constructor will call.
t3 = t2 = t1; //t2.operator=( t2.operator=( t1 ), t2 );
```

- Syntax:

```
class ClassName{
public:
ClassName& operator=( const ClassName &other ){
//TODO: Shallow / Deep Copy
return (*this);
}
}
```

- We get following functions for any class by default:
    - Constructor
    - Destructor
    - Copy constructor
    - Assignment operator function
- Example:

```
#include <iostream>
using namespace std;
#include<iostream>
using namespace std;
class Test{
    private:
     int Num1;
     int Num2;
    public:
     Test( void ){
     this->Num1 = 0;
     this->Num2 = 0;
     }
     Test(int Num1, int Num2)
     {
         cout<<"Parameter Cons"<<endl;
         this->Num1=Num1;
         this->Num2=Num2;
     }
     Test(Test &other)
     {
         cout<<"Test(Test &other)"<<endl;
         this->Num1=other.Num1;
         this->Num2=other.Num2;
     }
     friend ostream& operator<<( ostream &cout, Test &other ){
     cout << "Num1 Number : " << other.Num1 << endl;
     cout << "Num2 Number : " << other.Num2 << endl;
     return cout;
     }
     Test& operator=(Test &other)
     {
         cout<<"Operator=(Test &other)"<<endl;
         this->Num1=other.Num1;
         this->Num2=other.Num2;
         return (*this);
     }
};
int main( void ){
     Test t2;
     Test t1(100,200);
     Test t3;
     t3=t2=t1;              //t3.operator(&t3,t2.operator=(&t2,t1));
```

```
    cout<<t2<<t3;
    return 0;
}
```

- Index / Subscript operator overloading
    - Array
        - Definition: It is linear / sequential data structure / collection in which we can store multiple elements of same type in continuous memory location.
        - Types:
            - Single dimensional array
            - Multi dimensional array
        - To access elements of array, we should use integer index. Array index always begins with 0.
        - We can create array statically as well as dynamically.

```
int arr[ 3 ]; //Static memory allocation
int *ptr = new int[ 3 ]; //Dynamic memory allocation
```

- Advnatage of Array over linked
    - We can access elements of array randomly.
- Limitations of Array
    - It requires continous memory
    - We can not resize array
    - Element insertion and deletion is time consuming task
    - Using assignment operator we can not copy elements of array into another array.
- We can overcome limitations of array using 2 ways:
    - Use LinkedList instead of Array.
    - Encapsulate(declare variable as a data meber/member function) array inside class. Create object of the class and consider that object as a array.
    - If we want to consider object as a array then we should overload subscript / index operator.
    - If we want to use subscript operator with object at R.H.S of assignment operator then expression should return value.
- Example: Encapsulated Array Class

```
#include <iostream>
using namespace std;
class Array
{
private:
    int Size;
    int *ptr;
public:
    Array()
    {
        this->Size=0;
        this->ptr=nullptr;
    }
```

```
    Array(int Size)
    {
        this->Size=Size;
        this->ptr=new int[this->Size];
    }
    friend istream& operator>>(istream &cin, Array &other)
    {
        for(int i=0;i<other.Size;i++)
        {
            cout<<"Enter Element:   ";
            cin>>other.ptr[i];
        }
        cout<<endl;
        return cin;
    }
    friend ostream& operator<<(ostream &cout, Array &other)
    {
        for(int i=0;i<other.Size;i++)
        {
            cout<<other.ptr[i]<<"\t";
        }
        cout<<endl;
        return cout;
    }
    Array& operator=(Array &other)
    {
        this->Size=other.Size;
        this->~Array();         //Calling destructor explicitly
        this->ptr=new int[other.Size];      //Creating Deep Copy
        for(int i=0;i<this->Size;i++)
        {
            this->ptr[i]=other.ptr[i];
        }
        return (*this);
    }
    int& operator[](int Index)      //Overloading Array Subscript Operator
    {
        return this->ptr[Index];
    }
    ~Array()
    {
        if(ptr!=nullptr)
        delete[] ptr;
        ptr=nullptr;
    }
};
int main()
{
    Array a1(3);
    cin>>a1;
    a1[2]=40;
    int element=a1[2];      //a1.operator[](&a1,2);
    cout<<"Element: "<<element<<endl;
    return 0;
```

```
}
int main1()
{
    Array a1(3);
    Array a2(2);
    cin>>a1;
    a2=a1;
    cout<<a1;
    cout<<a2;
    return 0;
}
```

- Example:

```
int main( void ){
Array a1( 3 );
cin >> a1; //operator>>( cin, a1 )
int element = a1[ 2 ];
//int element = a1.operator[ ]( 2 );
cout << "Element : " << element << endl;
cout << a1; //operator<<( cour, a1 )
return 0;
}
```

- If we want to use subscript operator with object at L.H.S of assignment operator then expression should not return value. Rather it should return pointer/reference of the memory location.
- Example:

```
int main( void ){
 Array a1( 3 );
 cin >> a1; //operator>>( cin, a1 )
 a1[ 2 ] = 300;
 //a1.operator[ ]( 2 ) = 300;
 cout << a1; //operator<<( cour, a1 )
 return 0;
}
```

- Call operator / function call operator overloading
    - If we want to consider object as a function then we should overload call/function call operator.
    - If we consider object as a function then such object is called as function object / functor.
- Example:

```
class Test{
private:
 int Num1;
 int Num2;
```

```
public:
 Test( void ) : Num1( 0 ), Num2( 0 ){
 }
 void operator()( int Num1, int Num2 ) {
 this->Num1 = Num1;
 this->Num2 = Num2;
 }
 friend ostream& operator<<( ostream &cout, const Test &other ){
 cout << "Num1 Number : "<< other.Num1 <<endl;
 cout << "Num2 Number : "<< other.Num2 <<endl;
 return cout;
 }
};
int main( void ){
 Test t1;
 t1( 10, 20 ); //t1. operator()( 10, 20 );
 cout << t1; //operator<<( cout, t1 );
 return 0;
}
```

Virtual Function Table & Virtual Function Pointer

Revision Virtual function and Concept of Early Binding and Late Binding

- In case of upcasting, if we want to call member funtion, depending on type of object rather than type of pointer then we should declare member function in base class virtual.
- If class contains at least one virtual function then such class is called as polymorphic class.
- If signature of base class member function and derived class member function is same and if function in base class is virtual then derived class member function will be considered as virtual.
- Process of redefining virtual member function of base, class inside derived class, with same signature is called as function overriding and virtual function redefined in derived class is called as overrided function.
- For function overriding:
    - Function must be exist in base class and derived class
    - Signature of functions (including return type) must be same.
    - Function in base class must be virtual
- Definition:
    - In case of upcasting, A member function, which gets called depending on type of object rather that type of pointer is called as virtual funtion.
    - In case of upcasting, A member function of derived class which is deisigned to call using pointer/reference of base class is called as virtual function.
    - It means that virtual functions are not designed to call on object / class rather it is designed to call on base class pointer or base class reference.
- Can we declare static member function virtual?
    - Virtual member function is designed to call on base class pointer / reference.
    - Static member function is designed to call on class name.
    - Since static member function is not designed to call on base class pointer / referece, we can not declare static member function virtual.

    - Since we can not declare static member function virtual, we can not override it inside derived class.
- What is runtime polymorphism:
    - Process of calling member function derived class on pointer / reference of base class is called as runtime polymorphism.

```
class Demo1
{
    public:
    virtual void Show()
    {
        cout<<"Am Show of Demo1"<<endl;
    }
};
class Demo2
{
    public:
    virtual void Show()
    {
        cout<<"Am Show of Demo2"<<endl;
    }
};
class Demo3
{
    public:
    virtual void Show()
    {
        cout<<"Am Show of Demo3"<<end;
    }
};
int main()
{
    Demo1 *ptr=new Demo2();     //Object of Demo2 to the pointer of Demo1
    ptr->Show();                //Show of Demo2 will be called, Runtime
Polymorphism
    ptr=new Demo3();            //Object of Demo3 to the pointer of Demo1
    ptr->Show();                //Show of Demo3 will be called, Runtime
Polymorphism
    return 0;
}
```

Early Binding and Late Binding

- If call to the function gets resolved at compile time then it is called as early binding. In other words, if binding between function and object gets resolved at compile time then it is called as early binding.
- If call to the function gets resolved at runtime then it is called as late binding. In other words, if binding between function and object gets resolved at run time then it is called as late binding.
- If we call virtual or non virtual function on object then it is considered as early binding. This call always gets resolved at compile time.

- If we call non virtual function on pointer/reference then it is considered as early binding. This call always gets resolved at compile time.
- If we call virtual function on pointer/reference then it is considered as late binding. This call always gets resolved at run time. Consider below code:
- Example:

```
class A{
private:
int num1;
int num2;
public:
A( void ){
this->num1 = 10;
this->num2 = 20;
}
virtual void f1( void ){
cout << "A::f1" << endl;
}
virtual void f2( void ){
cout << "A::f2" << endl;
}
virtual void f3( void ){
cout << "A::f3" << endl;
}
void f4( void ){
cout << "A::f4" << endl;
}
void f5( void ){
cout << "A::f5" << endl;
}
};
class B : public A{
private:
int num3;
public:
B( void ){
this->num3 = 30;
}
virtual void f1( void ){
cout << "B::f1" << endl;
}
void f2( void ){
cout << "B::f2" << endl;
}
void f4( void ){
cout << "B::f4" << endl;
}
virtual void f5( void ){
cout << "B::f5" << endl;
}
virtual void f6( void ){
cout << "B::f6" << endl;
```

```
}
};
int main( void ){
A a; //OK
a.f1( ); //OK: A::f1 -> Early Binding
a.f2( ); //OK: A::f2 -> Early Binding
a.f3( ); //OK: A::f3 -> Early Binding
a.f4( ); //OK: A::f4 -> Early Binding
a.f5( ); //OK: A::f5 -> Early Binding
a.f6( ); //Not OK: f6 is not a member of class A
return 0;
}
```

- If we call any member function on object then it is considered as early binding

```
int main( void ){
A *ptr = new A( ); //OK
ptr->f1( ); //OK: A::f1 -> Late Binding
ptr->f2( ); //OK: A::f2 -> Late Binding
ptr->f3( ); //OK: A::f3 -> Late Binding
ptr->f4( ); //OK: A::f4 -> Early Binding
ptr->f5( ); //OK: A::f5 -> Early Binding
ptr->f6( ); //Not OK: f6 is not a member of class A
return 0;
}
```

- If we call virtual function on pointer then it is considered as late binding.
- If we call non virtual function on pointer then it is considered as early binding.

```
int main( void ){
A *ptr = new B( ); //OK: Upcasting
ptr->f1( ); //OK: B::f1 -> Late Binding
ptr->f2( ); //OK: B::f2 -> Late Binding
ptr->f3( ); //OK: A::f3 -> Late Binding
ptr->f4( ); //OK: A::f4 -> Early Binding
ptr->f5( ); //OK: A::f5 -> Early Binding
ptr->f6( ); //Not OK: f6 is not a member of class A
return 0;
}
```

```
int main( void ){
B *ptr = new B( );
ptr->f1( ); //OK: B::f1 -> Late Binding
ptr->f2( ); //OK: B::f2 -> Late Binding
ptr->f3( ); //OK: A::f3 -> Late Binding
ptr->f4( ); //OK: B::f4 -> Early Binding
ptr->f5( ); //OK: B::f5 -> Late Binding
```

```
ptr->f6( ); //OK: B::f6 -> Late Binding
return 0;
}
```

```
int main( void ){
B *ptr = new A( ); //NOT OK
ptr->f1( );
ptr->f2( );
ptr->f3( );
ptr->f4( );
ptr->f5( );
ptr->f6( );
return 0;
}
```

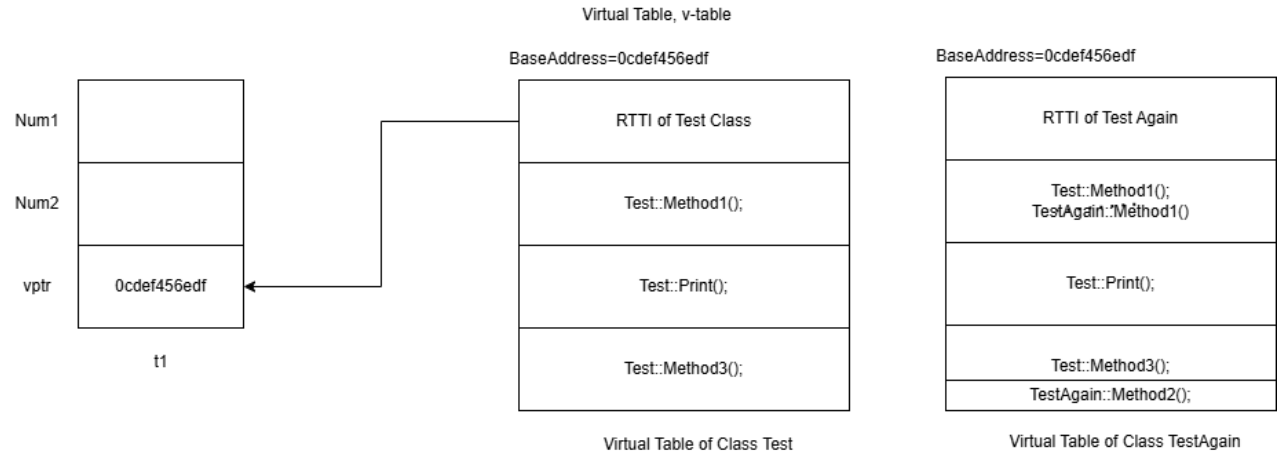
- Members of Base class inherit into Derived class. Hence we can consider Derived class object as Base class object.
- Also Base class pointer can contain address of derived class object.
- Members of Derived class do not inherit into Base class. Hence we can not consider Base class object as Derived class object.
- Also Derived class pointer can not contain address of Base class object.

```
int main( void ){
B b; //OK
b.f1( ); //OK: B::f1 -> Early Binding
b.f2( ); //OK: B::f2 -> Early Binding
b.f3( ); //OK: A::f3 -> Early Binding
b.f4( ); //OK: B::f4 -> Early Binding
b.f5( ); //OK: B::f5 -> Early Binding
b.f6( ); //OK: B::f6 -> Early Binding
return 0;
}
```

Virtual Function Table and Virtual Function Pointer.

- In case of upcasting, using base class pointer, if we want to call member function of derived class then we should declare function in base class virtual.
- If class contains at least one virtual function then compiler implicitly generate one table to store address of that virtual function(s). Such table(which can be array/strucrure depending on compiler vendor) is called as virtual function table / vf-table/v-table.
- In short, a table which contains address of virtual function is called as v-table.
- At the time of creation of V-Table for derived class, compiler simply copy Base class V-Table and make necessary changes.
- Compiler generate v-table per class. It gets generated at compile time.

- To store address of virtual function table, compiler implicitly declare pointer as a data meber inside class. Such pointer is called as virtual function pointer/vf-pointer/v-ptr.
- In short, a pointer which contain address of virtual function table is called as v-ptr.
- Consider V-Table and V-Ptr for the class Test and TestAgain.



- Compiler generates V-Table and V-Ptr at compile time.
- New definition of size of object:
  - sum of all the non static data members declared in base and derived class + (2/4/8) bytes depending on the compiler.
- Address of V-Table come into V-Ptr inside constructor. It means that V-ptr gets initialized after calling constructor.
- Can we declare constructor virtual? Why?
  - In C++, we can not declare constructor virtual
  - Reason 1:
    - Virtual functions are designed to call on base class pointer/reference only.
    - In C++, We can not call constructor on object, pointer or reference explicitly.
    - Since constructor is not designed to call on pointer or reference explicitly, we can not declare constructor virtual.
  - Reason 2:
    - To call any virtual function, compiler need to access value of vptr then it can do indexing into V-Table.
    - But V-Ptr gets initialized after calling constructor. Hence we can not declare constructor virtual.
- Can we declare destructor virtual? Why?
  - We can not declare constructor virtual but we can declare destructor virtual.
  - In Case of upcasting, constructor of Base class and Derived class gets call properly. But when we use delete operator on Base class pointer then only Base class destructor gets called.
  - To get call to destructor of Derived class first, we need to declare destructor in Base class virtual.
- Example:

```
class Base{
private:
 int *ptr;
public:
 Base( void ){
 this->ptr = new int[ 3 ];
 }
```

```
 virtual ~Base( void ){
 delete[] this->ptr;
 }
};
class Derived : public Base{
private:
 int *ptr;
public:
 Derived( void ){
 this->ptr = new int[ 5 ];
 }
 ~Derived( void ){
 delete[] this->ptr;
 }
};
int main( void ){
 Base *ptr = new Derived( );
 //TODO
 delete ptr;
 return 0;
}
```

- What is the difference between Function Overloading and Function Overriding?
    - We achieve compile time polymorphism using function overloading and run time polymorphism using function overriding.
    - In case of function overloading functions must be exist in same scope but in case function overriding functions must be exist in base class and derived class.
    - In case of function overloading signature of functions must be different but In case of function overriding signature of function must be same.
    - Return type is not considered in function overloading but return type is considered in function overriding.
    - Function overloading is based on mangled name whereas function overriding is based on V-Table and V-ptr
    - Function overloading do not require any keyword but for function overriding, function in base class must be virtual.
- How to get size of object in C++:

```
int main( void ){
 Test t1;
 size_t size = sizeof( t1 );
 cout<<size<<endl;
 return 0;
}
```

- size_t is a alias for unsigned long.
- How to get type of object in C++:

```
```

```
#include<iostream>
#include<typeinfo>
using namespace std;
int main( void ){
int number;
const type_info &type = typeid( number );
string typeName = type.name( );
cout << "Type Name : " << typeName << endl;
return 0;
}
```

- typeid is a operator which return reference of constant object of type_info class.
- type_info class is declared in std namespace and it is available int typeinfo header file.
- Consider declaration of type_info class

```
namespace std {
 class type_info{
 public:
 const char* name() const noexcept;
 bool operator==(const type_info& rhs) const noexcept;
 bool operator!=(const type_info& rhs) const noexcept;
 virtual ~type_info();
 private:
 type_info(const type_info& rhs);
 type_info& operator=(const type_info& rhs);
 };
}
```

- Process of getting type of object at runtime is called as Runtime Type Identification / Information.
- In case of upcasting, if we want to find out true type of object then we should use RTTI.
- Example:

```
#include<iostream>
#include<typeinfo>
using namespace std;
class Base{
int num1;
public:
Base( void ){
this->num1 = 10;
}
void print( void ){
cout << "Num1 : " << this->num1 << endl;
}
};
class Derived : public Base{
int num2;
public:
```

```
 Derived( void ){
 this->num2 = 20;
 }
 void print( void ){
 Base::print( );
 cout << "Num2 : " << this->num2 << endl;
 }
 };
 int main( void ){
 Base *ptrBase = new Derived( ); //Upcasting
 cout << typeid( ptrBase ).name( ) << endl; //P4Base
 cout << typeid( *ptrBase ).name( ) << endl; //4Base
 return 0;
 }
 int main4( void ){
 Derived *ptrDerived = new Derived( );
 cout << typeid( ptrDerived ).name( ) << endl; //P7Derived
 cout << typeid( *ptrDerived ).name( ) << endl; //7Derived
 return 0;
 }
 int main3( void ){
 Derived derived;
 cout << typeid( derived ).name( ) << endl; //7Derived
 return 0;
 }
 int main2( void ){
 Base *ptrBase = new Base( );
 cout << typeid( ptrBase ).name( ) << endl; //P4Base
 cout << typeid( *ptrBase ).name( ) << endl; //4Base
 return 0;
 }
 int main1( void ){
 Base base;
 cout << typeid( base ).name( ) << endl; //4Base
 return 0;
 }
```

- In case of upcasting, using RTTI, to get true type of object, Base class must be polymorphic.

```
class Base{
 int num1;
public:
 Base( void ){
 this->num1 = 10;
 }
 virtual void print( void ){
 cout << "Num1 : " << this->num1 << endl;
 }
};
class Derived : public Base{
 int num2;
public:
```

```
 Derived( void ){
 this->num2 = 20;
 }
 void print( void ){
 Base::print( );
 cout << "Num2 : " << this->num2 << endl;
 }
};
int main( void ){
 Base *ptrBase = new Derived( ); //Upcasting
 cout << typeid( ptrBase ).name( ) << endl; //P4Base
 cout << typeid( *ptrBase ).name( ) << endl; //7Derived
 return 0;
}
```

- Using Null pointer, if we try to find out true type of object then typeid operator throws bad_typeid exception.

```
int main( void ){
 try{
 Base *ptrBase = NULL; //Upcasting
 cout << typeid( ptrBase ).name( ) << endl; //P4Base
 cout << typeid( *ptrBase ).name( ) << endl; //7Derived
 }catch( bad_typeid &ex ){
 cout << ex.what() << endl;
 }
 return 0;
}
```

Advanced Type Casting Operators

- static_cast
- dynamic_cast
- const_cast
- reinterpret_cast
- reinterpret_cast operator
    - We can access private data members of the class inside non member function using:
        - Member functions e.g. getter and setter
        - Friend function
        - Pointer
- If we want to convert pointer of any type into pointer of any other type then we should use reinterpret_cast operator
- Example:

```
#include<iostream>
using namespace std;
class Test{
```

```
private:
 int Num1;
 int Num2;
public:
 Test( void ){
 this->Num1 = 10;
 this->Num2 = 20;
 }
 friend ostream& operator<<( ostream &cout, Test &other ){
 cout << "Num1 Number : " << other.Num1 << endl;
 cout << "Num2 Number : " << other.Num2 << endl;
 return cout;
 }
};
int main( void ){
 Test t1;
 cout << t1 << endl;
 //int *ptr = (int*)(&t1); //C-Style
 int *ptr = reinterpret_cast<int*>( &t1 ); //C++ Style
 *ptr = 50;
 ptr = ptr + 1;
 *ptr = 60;
 cout << t1 << endl;
 return 0;
}
```

- const_cast operator
    - If we want to convert pointer to constant object into pointer to non constant object or reference to constant object into reference to non constant object then we should use const_cast operator.
- Example:

```
 #include<iostream>
 using namespace std;
 class Test{
 int number;
 public:
 //Test *const this
 Test( void ){
 this->number = 10;
 }
 //Test *const this
 void showRecord( void ){
 cout << "Number : "<<this->number << endl;
 }
 //const Test *const this
 void displayRecord( void ) const{
 //Test *const ptr = ( Test *const)this; //C-Style
 Test *const ptr = const_cast<Test *const>( this ); //C++ Style
 ptr->showRecord( );
 }
 };
```

```
int main( void ){
const Test t;
t.displayRecord( );
return 0;
}
```

- static_cast operator
    - If we want to do type conversion between compatible types then we should use static_cast operator.
- Example:

```
int main( void ){
 double num1 = 10.5;
 //int num2 = ( int )num1; //C-Style
 int num2 = static_cast< int >( num1 ) ; //C++ -Style
 cout << "Num2 : " << num2 << endl;
 return 0;
}
```

- In case of non polymorphic type, if we want to do downcasting the we should use static_cast operator.
- static_cast operator do not check whether type consversion is valid or invalid. it only checks inheritance relationship at compile time.
- dynamic_cast operator
    - In case of polymorphic type, if we want to do downcasting the we should use dynamic_cast operator.
    - If we want to check whether, type conversion is valid or invalid then we should use dynamic_cast operator.
    - dynamic_cast operator checks inheritance relationship at runtime.
    - In case of pointer, if dynamic_cast operator fail to do conversion then it returns NULL.
    - In case of reference, if dynamic_cast operator fail to do conversion then it throws bad_cast excetion.
- Example:

```
int main( void ){
 Base *ptrBase = new Derived( ); //OK: Upcasting
 ptrBase->setNum1( 10 ); //OK
 ptrBase->setNum2( 20 ); //OK
 Derived *ptrDerived = dynamic_cast< Derived*>( ptrBase );
//Downcasting: C++ -Style
 if( ptrDerived != NULL ){
 ptrDerived->setNum3( 30 );
 ptrDerived->Base::print( );
 ptrDerived->Derived::print( );
 delete ptrBase;
 }
 return 0;
}
```

- Self Study: Enum Demo