Agenda

- Operator overloading
- Conversion Function
- Singleton class
- Factory Design Pattern
- noexcept
- nullptr
- Smart Pointer

Operator Overloading

- operator is token in C/C++.
- It is used to generate expression.
- operator is keyword in C++.
- Types of operator:
    1. Unary operator
    2. Binary Operator
    3. Ternary operator
- Unary Operator:
    - If operator require only one operand then it is called unary operator.
    - example : Unary(+,-,*) , &, !, ~, ++, --, sizeof, typeid etc.
- Binary Operator:
    - If operator require two operands then it is called binary operator.
    - Example:
        1. Arithmetic operator
        2. Relational operator
        3. Logical operator
        4. Bitwise operator
        5. Assignment operator
- Ternary operator:
    - If operator require three operands then it is called ternary operator.
    - Example:
        - Conditional operator(? :)
- In C/C++, we can use operator with objects of fundamental type directly.(No need to write extra code).

```
int num1 = 10; //Initialization
int num2 = 20; //Initialization
int num3 = num1 + num2; //OK
```

- In C++, also we can not use operator with objects of user defined type directly.
- If we want to use operator with objects of user defined type then we should overload operator.

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

- If we want to use operator with objects of user defined type then we should overload operator.
- To overload operator, we should define operator function.
- We can define operator function using 2 ways
    1. Using member function
    2. Using non member function.
- By defining operator function, it is possible to use operator with objects of user defined type. This process of giving extension to the meaning of operator is called operator overloading.
- Using operator overloading we can not define user defined operators rather we can increase capability of existing operators.

Limitations of operator overloading

- We can not overloading following operator using member as well as non member function:

    1. dot/member selection operator(.)
    2. Pointer to member selection operator(.*)
    3. Scope resolution operator(::)
    4. Ternary/conditional operator(? :)
    5. sizeof() operator
    6. typeid() operator
    7. static_cast operator
    8. dynamic_cast operator
    9. const_cast operator
    10. reinterpret_cast operator

- We can not overload following operators using non member function:

    1. Assignment operator(=)
    2. Subscript / Index operator([])
    3. Function Call operator[()]
    4. Arrow / Dereferencing operator(->)

- Using operator overloading, we can change meaning of operator.

- Using operator overloading, we can not change number of parameters passed to the operator function.

Operator overloading using member function(operator function must be member function)

- If we want to overload, binary operator using member function then operator function should take only one parameter.
- Using operator overloading, we can not change, precedance and associativity of the operator.
- If we want to overload unary operator using member function then operator function should not take any parameter.

```
c3 = c1 + c2; //c3 = c1.operator+(c2);

c4 = c1 + c2 + c3; //c4 = c1.operator+( c2 ).operator+( c3 );
```

Operator overloading using non member function(operator function must be global function)

- If we want to overload binary operator using non member function then operator function should take two parameters.
- If we want to overload unary operator using non member function then operator function should take only one parameters.

```
c3 = c1 + c2; //c3 = operator+(c1,c2);

c4 = c1 + c2 + c3; //c4 = operator+(operator+(c1,c2),c3);

c2 = ++ c1; //c2=operator++( c1 );
```

Overloading Insertion Operator(<<)

-cout is an external object of ostream class which is declared in std namespace.

- ostream class is typdef of basic_ostream class.
- If we want print state of object on console(monitor) then we should use cout object and insertion operator(<<).
- Copy constructor of ostream class is private hence we can not copy of cout object inside our program
- If we want to avoid copy then we should use reference.
- If we want to print state of object(of structure/class) on console then we should overload insertion operator.

```
//ostream out = cout; // NOT OK
ostream &out = cout; //OK
```

```
1. cout<<c1; //cout.operator<<( c1 );
2. cout<<c1; //operator<<(cout, c1 );
```

- According to first statement, to print state of c1 on console, we should define operator<<( ) function inside ostream class. But ostream class is library defined class hence we should not modify its implementation.
- According to second statement, to print state of c1 on console, we should define operator<<() function globally. Which possible for us. Hence we should overload operator<<( ) using non member function.

```
class ClassName
{
  friend ostream& operator<<( ostream &cout, ClassName &other );
};

ostream& operator<<( ostream &cout, ClassName &other )
{
  //TODO : print state of object using other
  return cout;
}
```

Overloading Extraction Operator(>>)

- cin stands for character input. It represents keyboard.
- cin is external object of istream class which is declared in std namespace.
- istream class is typedef of basic_istream class.
- If we want to accept data/state of the variable/object from console/keyboard then we should use cin object and extraction operator.
- Copy constructor of istream class is private hence, we can not create copy of cin object in out program.
- To avoid copy, we should use reference.

```
istream in = cin; // NOT OK
istream &in = cin; // OK
```

- If we want to accept state of object (of structure/class) from console(keyboard) then we should overload extraction operator.

```
1. cin>>c1; //cin.operator>>( c1 )

2. cin>>c1;//operator>>( cin, c1 );
```

- According to first statement, to accept state of c1 from console, we should define operator>>( ) function inside istream class. But istream class is library defined class hence we should not modify its implementation.

- According to second statement, to accept state of c1 from console, we should define operator>>() function globally. Which possible for us. Hence we should overload operator>>() using non member function.

```
class ClassName
{
  friend istream& operator>>( istream &cin, ClassName &other );
};
istream& operator>>( istream &cin, ClassName &other )
{
  //TODO : accept state of object using other
  return cin;
}
```

Overloading Call / Function Call operator:

- If we want to consider any object as a function then we should overload function call operator.

```
class Complex
{
private:
    int real;
    int imag;

public:
    Complex(int real = 0, int imag = 0)
    {
        this->real = real;
        this->imag = imag;
    }
    void operator()(int real, int imag)
    {
        this->real = real;
        this->imag = imag;
    }
    void printRecord(void)
    {
        cout << "Real Number :" << this->real << endl;
        cout << "Imag Number :" << this->imag << endl;
    }
};
int main(void)
{
    Complex c1;
    c1(10, 20); // c1.operator()( 10, 20 );
    c1.printRecord();
    return 0;
}
```

- If we use any object as a function then such object is called function object or functor.
- In above code, c1 is function object.

Index/Subscript Operator Overloading

- If we want to overcome limitations of array then we should encapsulate array inside class and we should perform operations on object by considering it array.
- If we want to consider object as a array then we should overload sub script/index operator.

```
//Array *const this = &a1
int& operator[]( int index )throw( ArrayIndexOutOfBoundsException )
{
  if( index >= 0 && index < SIZE )
    return this->arr[ index ];
  throw ArrayIndexOutOfBoundsException("ArrayIndexOutOfBoundsException");
}

//If we use subscript operator with object at RHS of assignment operator then
expression must return value from array.

Array a1;
cin>>a1; //operator>>( cin, a1 );
cout<<a1; //opeator<<( cout, a1 );
int element = a1[ 2 ]; //int element = a1.operator[]( 1 );

// If we want to use sub script operator with object at LHS of assignment operator
then expression should not return a value rather it should return either address /
reference of memory location.

Array a1;
cin>>a1; //operator>>( cin, a1 );
a1[ 1 ] = 200; //a1.operator[]( 1 ) = 200;
cout<<a1; //opeator<<( cout, a1 );
```

Overloading assignment operator.

- If we initialize newly created object from existing object of same class then copy constructor gets called.
- If we assign, object to the another object then assignment operator function gets called.

```
Complex c1(10,20);
Complex c2 = c1; //On c2 copy ctor will call

Complex c1(10,20);
Complex c2;
c2 = c1; //c2.operator=( c1 )
```

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

- If we do not define assignment operator function inside class then compiler generates default assignment operator function for the class. By default it creates shallow Copy.
- During assignment, if there is need to create deep copy then we should overload assignment operator function.

Overloading dereferencing/arrow operator.

```
class Array
{
};

// Static memory allocation for the object
Array a1;

//Dynamic memory allocation for the object
Array *ptr = new Array( );
delete ptr;
ptr = NULL;
```

- If we create object of a class dynamically then implicitly ctor gets called and if try to deallocate that memory then dtor gets called.
- Using object, if we want to access members of the class then we should use dot/member selection operator.
- In other words, using dot operator, if we want to access members of the class then left hand operand must be object of a class.
- Using pointer, if we want to access members of the class then we should use arrow or dereferencing operator.
- In other words, using arrow operator, if we want to access members of the class then left side operand must be pointer of a class.
- if we use any object as a pointer then such object is called smart pointer.

```
class AutoPtr
{
private:
    Array *ptr;
```

```
public:
    AutoPtr(Array *ptr)
    {
        this->ptr = ptr;
    }
    Array *operator->()
    {
        return this->ptr;
    }
    ~AutoPtr()
    {
        delete this->ptr;
    }
};
int main(void)
{
    AutoPtr obj(new Array(3));
    obj->acceptRecord(); // obj.operator ->()->acceptRecord();
    obj->printRecord(); // obj.operator ->()->printRecord();
    return 0;
}
```

Conversion Function

It is a member function of a class which is used to convert state of object of fundamental type into user defined type or vice versa. Following are conversion functions in C++

1. Single Parameter Constructor

```
int main( void )
{
int number = 10;
Complex c1 = number; //Complex c1( number );
c1.printRecord();
return 0;
}
```

- In above code, single parameter constructor is responsible for converting state of number into c1 object. Hence single parameter constructor is called conversion function.

2. Assignment operator function

```
int main( void )
{
int number = 10;
Complex c1;
c1 = number;//c1 = Complex( number );
//c1.operator=( Complex( number ) );
c1.printRecord();
```

```
return 0;
}
```

- In above code, assignment operator function is responsible for converting state of number into c1 object hence it is considered as converion function.
- If we want to put restriction on automatic instantiation then we should declare single parameter constructor explicit.
- "explicit" is a keyword in C++.
- We can use it with any constructor but it is designed to use with single parameter constructor.

3. Type conversion operator function.

```
int main( void )
{
Complex c1(10,20);
int real = c1; //real = c1.operator int( )
cout<<"Real Number : "<<real<<endl;
return 0;
}
```

- In above code, type conversion operator function is responsible for converting state of c1 into integer variable(real). Hence it is considered as conversion function.

Operator overloading using member function vs non member function:

- During operator overloading, if left side operand need not to be l-value then we should overload operator using non member function.
- We should overload following operators using non member function:
    1. Arithmetic Operators
    2. Relational Operators
    3. Logical Operators
- During operator overloading, if left side operand need to be l-value then we should overload operator using member function.
- We should overload following operators using member function:
    1. =, [], (), ->
    2. Short hand operators
    3. Unary Operators(++, --)

Singleton class

- It is a design pattern
- Design patterns are a standard solution to well-known problem
- It enables to use a single object of the class through out the application

Factory Design Pattern

- It is a creational design that offers a way to produce objects in a baseclass while letting derived classes change the kind of objects that are created.
- It is useful when there is a need to create multiple objects of the same type, but the type of the objects is not known until runtime.
- It is implemented using a factory function, which is a method that returns an object of the specified type.

noexcept operator

- The noexcept operator performs a compile-time check that returns true if an expression is declared to not throw any exceptions.
- It can be used within a function template's noexcept specifier to declare that the function will throw exceptions for some types but not others.

```
void may_throw(); // it will throw exception
void no_throw() noexcept; // it will not throw exception

int main()
{
cout<<"may_throw() is noexcept(" << noexcept(may_throw())<<")"<<endl;
cout<<"no_throw() is noexcept(" << noexcept(no_throw())<<")"<<endl;
}

// output
// may_throw() is noexcept(false)
// no_throw() is noexcept(true)
```


Smart Pointer

- Smart pointers are objects that behave like pointers but provide automatic memory management.
- Smart pointers enable automatic, exception-safe, object lifetime management.
- They help prevent memory leaks and manage the lifetime of dynamically allocated objects.
- Smart pointers are part of the C++ Standard Library and are implemented as template classes.
- The main smart pointers in C++ are:

1. std::unique_ptr

- `std::unique_ptr` is a smart pointer that owns a dynamically allocated object and ensures that the object is deleted when the `unique_ptr` goes out of scope or is reset.
- It is unique in that it cannot be copied or shared. It can only be moved.
- It is lightweight and efficient because it does not incur the overhead of reference counting.
- The object is disposed of, using the associated deleter when either of the following happens:
    1. the managing unique_ptr object is destroyed.
    2. the managing unique_ptr object is assigned another pointer via operator= or reset().

2. std::shared_ptr

- `std::shared_ptr` is a smart pointer that retains shared ownership of an object through a pointer.

- Several shared_ptr objects may own the same object.
- The object is destroyed and its memory deallocated when either of the following happens:
    1. the last remaining shared_ptr owning the object is destroyed;
    2. the last remaining shared_ptr owning the object is assigned another pointer via operator= or reset().
- It manages the ownership of a dynamically allocated object using reference counting.

3. std::weak_ptr

- `std::weak_ptr` is a smart pointer that provides a non-owning reference to an object managed by `std::shared_ptr`.
- It does not participate in reference counting and does not keep the object alive.
- It is used to break cyclic dependencies between `std::shared_ptr` objects.
- It must be converted to `std::shared_ptr` in order to access the referenced object.
- It models temporary ownership when an object needs to be accessed only if it exists, and it may be deleted at any time by someone else

nullptr

- A nullptr is a keyword introduced in C++11 to represent a null pointer.
- It provides a type-safe and clearer alternative to using NULL or 0 for null pointer constants.
- It's recommended to use nullptr in modern C++ code.
- nullptr helps avoid ambiguities in function overloading and template specialization scenarios.

```
void f1(int *n)
{
    cout << "Function with int* " << endl;
}
void f1(int n)
{
    cout << "Function with int " << endl;
}

int main()
{
    int *ptr1 = 0;
    int *ptr2 = NULL;
    int *ptr3 = nullptr;
    cout << "ptr1 = " << ptr1 << endl;
    cout << "ptr2 = " << ptr2 << endl;
    cout << "ptr3 = " << ptr3 << endl;

    f1(0); // fun with int
    // f1(NULL); // ambugity
    f1(nullptr); // fun with int*

    return 0;
}
```