Agenda

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

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;
}
```

SUNBEAM INFOTECH

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

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();
```

SUNBEAM INFOTECH

```
    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( )
```

SUNBEAM INFOTECH

```
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.

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.

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.

SUNBEAM INFOTECH

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;
}
```

SUNBEAM INFOTECH