Lecture 4
POLYMORPHISM & POINTERS

September 13, 2021
Monday

OVERLOADING

Giving the same name to more than one method, or operators for more than one operations.

- Methods
  - Member Functions
  - Constructor
- Operators

FUNCTION OVERLOADING

- Parameters must differ either by type Or by number.
- Methods are within the same scope. Doesn’t require inheritance.
- Less flexible, but faster execution.

FUNCTION OVERLOADING

```
class Node {
    private:
        int num1, num2;
    public:
        Node (int a, int b ) : num1(a), num2(b) {      }

        void add ( ) { cout<<num1 + num2;     }

        int add ( ) { return num1 + num2; }

        int add (int i) { return num1 + num2 + i;  }

        double add (double j) {return num1 + num2 + i ; }
};
```

All functions are overloaded?

CONSTRUCTOR OVERLOADING

Design an OOP based solution for students, faculty and staff of employees. Highlighting how you solution is achieving. Just 5 ~ 10 Minutes be very brief.

- Inheritance
- Encapsulation
- Abstraction
- Polymorphism
  - Constructor Overloading
  - Function Overloading.
- Allowing a Deep Copy for Constructor

OPERATOR OVERLOADING

We can overload the operators for user defined types like objects.

- Provides additional usage for the same operator.
- Some operators which C++ does not allow to be overloaded
  - Scope Operator (::)
  - sizeof
  - member selector (.), member pointer selector (*)
  - Ternary Operator (? :)

OPERATOR OVERLOADING

```
class className {
    ... .. ...
    public
      returnType operator symbol (arguments) {
          ... .. ...
      }
    ... .. ...
};
```

Can we use Friend Function for Operator Overloading outside the class?

- Operator is keyword.
- Symbol is the operator you want to overload e.g., +
- Defined inside the class or struct for the objects of that class to use the overloaded operator.

OPERATOR OVERLOADING

```
class ComplexNumber {
    private:
        int real, imaginary;
    public:
        ComplexNumber ( ) { real = 0, imaginary = 0 }
        ComplexNumber (int r, int i) : real ( r ), imaginary (i) { }

        void DisplayValue ( ) {    cout<<real<<" + " <<imaginary<<"i"<<endl;
};

int main ( ) {
    ComplexNumber firstNumber(10, 14), secondNumber(20, 31), thirdNumber;
    thirdNumber = firstNumber + secondNumber;
    return 0;
}
```

Will this Work?

OPERATOR OVERLOADING

```
class ComplexNumber {
    public:

        ComplexNumber operator + (const Complex& number){
            Complex obj;
            obj.real = real + number.real;
            obj.imaginary = imaginary + number.imaginary;
            return obj;
        }

};
```

ASSIGNMENT OPERATOR OVERLOADING

```
class  Node {
        char* name;
        int age;

        Node ( char* n = "", int a ) {
                name = strdup (n);
                age = a;
        }

        Node (const Node& n){
                name = strdup (n.name);
                age = n.age;
        }
};
```

```
int main ( ) {
        Node node1 ("Akram", 20), node2
("Anwar", 32), node3;
        node3 = node1;

        strcpy(node1.name, "Aslam");
        node1.age = 33;

        strcpy(node3.name, "Amjad");
        node3.age = 34;


}
```

ASSIGNMENT OPERATOR OVERLOADING

```
class  Node {
        Node ( char* n = “”, int a ) {
                name = strdup (n);
                age = a;
        }

Node operator = (const Node& n) {
        if (this != &n) {        // no assignment to itself;
                if (name != 0)
                        free(name);
                name = strdup(n.name);
                age = n.age;
        }
        return *this; }
};
```

Each object can access its own address through the pointer *this.*

*this is the object to itself.

OVERRIDING

Giving the same method a **different implementation** in derived class than the base class function, with Virtual Functions

- Parameters must be same by type and number.
- Methods are in different scopes. Occurs with inheritance.
- More flexible, but slower execution.

OVERRIDING

```
class Base {
    public:
        virtual void display ( ) {
            cout<<"I am a Base display function"<<end;
        }
};

class Derived : Base {
    public:
        void display ( ) {
            cout<<"I am a Derived display function"<<endl;
        }
}
```

OVERRIDING | ISSUES

- Functions with incorrect names.
- Functions with different return types.
- Functions with different parameters.
- No Virtual Function declared in the Base Class.

OVERRIDING | OVERRIDE

- C++ 11 provided **override** identifier.
- Avoids the chances of error and forces them on compile time.

```
class Base {
    public:
        virtual void display ( ) {
            cout<<"I am a Base display function"<<end;
        }
};

class Derived : Base {
    public:
        void display ( ) override {
            cout<<"I am a Derived display function";
        }
}
```

MEMBER SELECTION OPERATOR

- The dot "." operator.
  - Can be used with an object to directly access "Direct Member Selection".
    - Data members
    - Member Functions
- We can define a pointer of class type which can points to the objects of the class.
- We can access the data members & member functions through pointers too.
  - Known as "Indirect Member Selection"
  - First we need to dereference the pointer then use the dot operator.
  - Or we can use the more preferred approach
    - arrow (->) to access the member without the need of dereferencing.

MEMBER SELECTION OPERATORS

```
class  Node {
        char* name;
        int age;

        Node ( char* n = "", int a ) {
                name = strdup (n);
                age = a;
        }

        Node (const Node& n){
                name = strdup (n.name);
                age = n.age;
        }
};
```

```
int main ( ) {
        Node node1 ("Akram", 20),*nodePtr;
        Node node2 ("Aslam", 18);
        nodePtr = &node2;

        strcpy(node1.name, "Anwar);
        strcpy( (*nodePtr).name, "Amjad");

        strcpy(nodePtr->name, "Akbar");
}
```

Why we use (*nodePtr) instead of *nodePtr

POINTERS TO MEMBERS

- Since data member and member function of class also reside in memory and have an address.
- We can use pointers to point to data members and member functions as well.

```
datatype class_name :: *pointer_name;

pointer_name = &class_name :: datamember_name;

datatype class_name::*pointer_name = &class_name::datamember_name;

return_type (class_name::*ptr_name) (argument_type) = &class_name::function_name;
```

POINTERS TO MEMBERS

- Since data member and member function of class also reside in memory and have an address.
- We can use pointers to point to data members and member functions as well.

```
datatype class_name :: *pointer_name;

pointer_name = &class_name :: datamember_name;

datatype class_name::*pointer_name = &class_name::datamember_name ;
```

```
class SampleClass {
public:
   void display() { cout << "Hello World"; }
   int age;
};
```

```
int main() {
   void (SampleClass::*pDisplay)() =
&SampleClass::display;
   int MyClass::*pAge = &SampleClass::age;

   SampleClass obj;
   Sample *pobj = new SampleClass;

   obj.*pAge = 10;
   pobj->*pAge = 20;
   (obj.*pDisplay)();
   (pobj->*pDisplay)();

   delete pobj;
   return 0;
}
```

POINTERS ARITHMETIC

- Allowed Operations
  - Incremented ++
  - Decremented --
  - Integer Addition + Or +=
  - Integer Subtraction - Or -=
  - One Pointer subtracted from Another
    - Only with Arrays