

# INTRO TO OBJECT ORIENTED PROGRAMMING

---

Problem Solving with Computers-II

C++

```
#include <iostream>
using namespace std;

int main(){
    cout<<"Hola Facebook!n";
    return 0;
}
```

GitHub



# Today's goals

- Intro to Object Oriented Programming
- Defining classes and declaring objects
- Access specifiers: private, public
- Different ways of initializing objects and when to use each:
  - Default constructor
  - Parametrized constructor
  - Parameterized constructor with default value
- Operator overloading
  - what is operator overloading?
  - why/when would we need to overload operators?
  - how to overload operators in C++ ?

# Procedural Programming

- Break down a problem into sub tasks (functions)
- Algorithm to bake a cake

Preheat the oven to 350F

Get the ingredients: 2 eggs, 1 cup flour, 1 cup milk

Mix ingredients in a bowl

Pour the mixture in a pan

Place in the over for 30 minutes

# Object Oriented Programming: A cake baking example

- Solution to a problem is a system of interacting **objects**
- An object has attributes and behavior
- What are the objects in this example?
  1. Preheat the oven to 350F
  2. Get the ingredients: 2 eggs, 1 cup flour, 1 cup milk
  3. Mix ingredients in a bowl
  4. Pour the mixture in a pan
  5. Place in the over for 30 minutes

# Objects have attributes and behavior:

## A cake baking example

<b>Object</b>	<b>Attributes</b>	<b>Behaviors</b>
Oven	Size Temperature Number of racks	Turn on Turn off Set temperature
Bowl	Capacity Current amount	Pour into Pout out
Egg	Size	Crack Separate(white from yolk)

# A class: pattern for describing similar objects

A generic pattern that is used to describe objects that have similar attributes and behaviors

e.g. a bowl and a pan may be described by the same class

```
class Dish{
    void pourIn( double amount);
    void pourOut(double amount);
    double capacity;
    double currentAmount;
};
```

# Objects vs classes

```
class Dish{
    void pourIn( double amount);
    void pourOut(double amount);
    double capacity;
    double currentAmount;
};
//Creating objects of this class
```

## Concept: Classes describe objects

- Every object belongs to (is an **instance** of) a **class**
- An object may have **fields**, or **variables**
  - The class describes those fields
- An object may have **methods**
  - The class describes those methods
- A class is like a template, or cookie cutter



# Abstract Data Types (ADT)

- Abstract Data Type is defined by data + operations on the data.
- Key features
  - **Abstraction:** hide implementation details
  - **Encapsulation:** bundle data and operations on the data, restrict access to data only through permitted operations

```
class Dish{
public:
    void pourIn( double amount);
    void pourOut(double amount);
private:
    double capacity;
    double currentAmount;
};
```

# Approximate Terminology

- instance = object
- field = instance variable
- method = function
- sending a message to an object = calling a function

## How many objects of the ADT Complex are created in main()?

```
int main(){
    Complex p;
    Complex w;
    w.setReal(1);
    w.setImag(2);
    p = w;
    p.conjugate();
    p.print();
}
```

```
class Complex
{
private:
    double real;
    double imag;
public:
    double getMagnitude() const;
    double getReal() const;
    double getImaginary() const;
    void print() const;
    void conjugate();
    void setReal(double r);
    void setImag(double r);
};
```

- A. One
- B. Two
- C. Three
- D. Four
- E. I am not sure . . .

# Will this code compile?

```
int main(){
    Complex p;
    Complex w(1, 2);
    p = w;
    p.conjugate();
    p.print();
}
```

- A. Yes
- B. No
- C. I am not sure . . .

```
class Complex
{
private:
    double real;
    double imag;
public:
    double getMagnitude() const;
    double getReal() const;
    double getImaginary() const;
    void print() const;
    void conjugate();
    void setReal(double r);
    void setImag(double r);
};
```

# Will this code compile?

```
int main(){
    Complex p;
    Complex w(1, 2);
    p = w;
    p.conjugate();
    p.print();
}
```

- A. Yes
- B. No: We need a parametrized constructor
- C. I am not sure . . .

```
class Complex
{
private:
    double real;
    double imag;
public:
    Complex(double re = 0, double im = 0);
    double getMagnitude() const;
    double getReal() const;
    double getImaginary() const;
    void print() const;
    void conjugate();
    void setReal(double r);
    void setImag(double r);
};
```

# Clever things in C++

# New method: add()

```
int main(){
    Complex p;
    Complex q(2, 3);
    Complex w(10, -5);
    w.conjugate();
    p = _____;
    p.print();
}
```

Approach 1

```
int main(){
    Complex p;
    Complex q(2, 3);
    Complex w(10, -5);
    w.conjugate();
    p = _____;
    p.print()
}
```

Approach 2

# New method: add()

```
int main(){  
    Complex p;  
    Complex q(2, 3);  
    Complex w(10, -5);  
    w.conjugate();  
    p = add(q, w);  
    p.print();  
}
```

Approach 1

```
int main(){  
    Complex p;  
    Complex q(2, 3);  
    Complex w(10, -5);  
    w.conjugate();  
    p = q.add(w);  
    p.print();  
}
```

Approach 2



# Overloading the + operator for Complex objects

```
p = add(q, w);
```

```
p = q.add(w);
```

```
p = x + w;
```

Goal: We want to apply the + operator to Complex type objects

# Overloading the << operator

```
int main(){  
    Complex w(10, -5);  
    w.conjugate();  
    w.print();  
}
```

Before overloading the << operator

```
int main(){  
    Complex w(10, -5);  
    w.conjugate();  
    cout << w;  
}
```

After overloading the << operator

```
cout << w;
```

Select any equivalent C++ statement:

```
w.operator<<(cout);
```

A

```
cout.operator<<(w);
```

B

```
operator<<(cout, w);
```

C

```
operator<<(cout, w);
```

Select the function declaration that does NOT match the above call

A 

```
void operator<<(ostream &out,  
               const Complex &c);
```

B 

```
void Complex::operator<<(ostream &out);
```

C 

```
Complex operator<<(ostream &out,  
                  Complex c);
```

# Operator Overloading

We would like to be able to perform operations on two objects of the class using the following operators:

<<

==

!=

+

-

and possibly others

# Some advice on designing classes

- Always, *always* strive for a narrow interface
- Follow the **principle of abstraction and encapsulation**:
  - the caller should know as little as possible about how the method does its job
  - the method should know little or nothing about where or why it is being called
  - Your class is responsible for its own data; don't allow other classes to easily modify it! Make as much as possible **private**

# What we have spoken about so far?

- Class = Data + Member Functions.
- Abstract Data Type = abstraction + encapsulation (uses classes)
- How to call member functions.
- How to implement a class's methods.