Lab #5

- Write a program that allows the user to type any amount of input numbers (-1 to stop):
  - Print out the sorted result

- Start by creating a dynamic array of size 5.

- Whenever the user enters too many numbers, grow the size of the array by 2
  - For example, when the 6th number is entered:
    - Create a new dynamic array of size 10
    - Transfer the existing 5 numbers
    - **Delete** the old array, change the pointer.
    - Add the 6th number
Today’s Topics

- Introduction to C++ Objects
  - Creating a data type of our own
  - Using properties and functions
  - Pre-Defined classes
Objects and Variables

- We have seen a number of different data types
  - int, double, float, char, bool

- In a way, they represent *abstract* ideas of “things”
  - Integers -> 01011001
  - ‘A’ -> 110100
  - True -> 1
Data Types can be *instantiated* as variables - or *instances*.

```
int x, y; // x and y are instances
```

You can have many instances of the same data-type

- Each instance (or variable) has its own memory
- Each instance conforms to the same “rules” as all other instances of the common data type

Built-in data-types are also called *primitives*
Objects and Variables

- Often, we deal with programs that have *higher level* concepts involved.
  - We write programs that deal with:
    - Students and grades
    - Circles and rectangles (area, volume, etc.)
    - Bank loans and interest calculations
    - Etc.

- As our programs get larger, it is useful to model these ideas *directly* in C++
  - We will consider these ideas *objects*. 
Objects

- Objects can be composed of two types of things:
  - State (Properties)
  - Behaviors (Actions)

- For example, we can consider a circle
  - Property: radius
  - Action: calculate area

- Or a loan...
  - Property: Principle, Interest Rate
  - Action: calculate monthly payment, print balance, etc.
Object-Oriented Programming

- OOP is the predominant style of modern programming
  - C++ was one of the first languages to stress this
  - C#, Java are other common OOP languages
  - Many languages have been given support:
    - JavaScript, PHP, Ruby, Groovy
Classes

- A class is a user-defined data-type.
  - A class’ s instances are called **objects**

**Object** is to **class** as **variable** is to **data type**

- Let's create a Circle class in C++
  - Property: Radius
  - Action: getArea()
```cpp
class Circle {
public:

    double radius;

    Circle() {
        radius = 1;
    }

    Circle(double r) {
        radius = r;
    }

    double getArea() {
        return radius * radius * 3.14159;
    }
};
```
A class definition is **not** a program

- Its job is to define a new type (or thing)
- It is a *blueprint* for how its instances are built and behave

To create an instance, you use the same syntax as regular variables (mostly)

- `int x;`
- `Circle c;`
Properties: Member Variables

- The semantics of Properties are different than you are used to.

- Think of an object as a collection of member variables
  - Each Circle instance has its own radius

- Later we will add more properties and functionality to our classes - we will come to think of them as “containers”

```java
int x = 7;
int y = 5;
Circle c1;
Circle c2;
c1.radius = 7;
```
Constructors

- When creating a primitive, C++ allocates memory for you, but that’s it..
- When creating an object, C++ provides a way for you to perform some initialization to prepare your object.
  - Each class defines one or more constructors.
  - Special “functions” that automatically get called whenever an instance/object is created.
Constructors have **no** return type.
- Their return is implicit - it’s the newly initialized object

**Default constructors**
- No parameters
- Called somewhat “magically”

```cpp
class Circle {
public:
    double radius;

    Circle() {
        radius = 1;
    }
};

int main() {
    Circle c;
    cout << c.radius << endl;
}
```
You can provide “custom” constructors which accept arguments as parameters also.

If you have custom constructors, you must write a default constructor.

class Circle {
    public:
        double radius;

        Circle() {
            radius = 1;
        }
        Circle(double r) {
            radius = r;
        }
    
    int main() {
        Circle c(15);
        cout << c.radius << endl;
    }
Our objects can perform actions - which we call **member functions**.

The operate just like normal functions - but:
- They have direct access to the member variables
- They are called on **specific** instances of our objects

```cpp
class Circle {
public:
    double radius;
    ...
    double getArea() {
        return radius * radius * 2 * PI;
    }
};

int main() {
    Circle c;
    cout << c.getArea() << endl;
}
```
Using instances

- Write a program that creates two circles
  - One using default constructor
  - User enters radius of the other
  - Print area of both circles
When primitives are involved, the assignment operator works fairly simply:

```
int x = 5;
int y;
y = x;
```

- For classes, the assignment operator performs a little more work on your behalf
  - Each member variable is copied

```
Circle c1;
Circle c2(4);
c1 = c2;  (copies c2’s radius into c1’s radius)
```
Classes often contain many member variables and functions.

To use someone else’s class, you don’t need to know how each function is implemented

You just need to know what is available.
```cpp
class Circle {
public:
    double radius;

    Circle();
    Circle(double r);
    double getArea();
};

#include "Circle.h"

Circle :: Circle() {
    radius = 1;
}

Circle :: Circle(double r) {
    radius = r;
}

double Circle :: getArea() {
    return 2 * PI *
           radius * radius;
}
```
Data Encapsulation

```cpp
class Circle {
public:
    double radius;
    Circle();
    Circle(double r);
    double getArea();
};
```

- The keyword `public` tells C++ to allow “others” to use the variables and functions below.
- Often it would be nice to keep all the variables, and some of the functions, private.
- Private means **only** member functions within `Circle` can use those variables and functions.
Data Encapsulation

- Let's modify our Circle class to do the following:
  - Prevent "user" from setting negative radius
  - Pre-Calculate area so it doesn’t need to be recomputed if radius has not changed since the last time getArea() was called.
Variable Scope

- Variable scope is a potential issue when dealing with member functions.
  - Member variables are “global” to all member functions
  - Local parameters or variables within member functions will hide member variables of the same name.

```cpp
double setRadius(double radius) {
    radius = radius; ???????
}
```
Lab 6

- Create a BankAccount Class
  - Properties (Member Variables):
    - Balance
    - Interest Rate (yearly)
  - Actions (Member Functions):
    - Withdrawl(double amountToWithdraw)
    - Deposit(double amountToDeposit)
    - ApplyYearlyInterest() use Deposit function

- Write a main program that uses your class and lets the user perform the three actions.
“Example” main

```cpp
int main() {
    BankAccount account;
    account.Deposit(400);
    cout << "Balance: $" << account.getBalance() << endl;
    account.ApplyInterest();
    cout << "Balance: $" << account.getBalance() << endl;
    account.Withdraw(50);
    cout << "Balance: $" << account.getBalance() << endl;
    system("pause");
}
```