PIC 10A

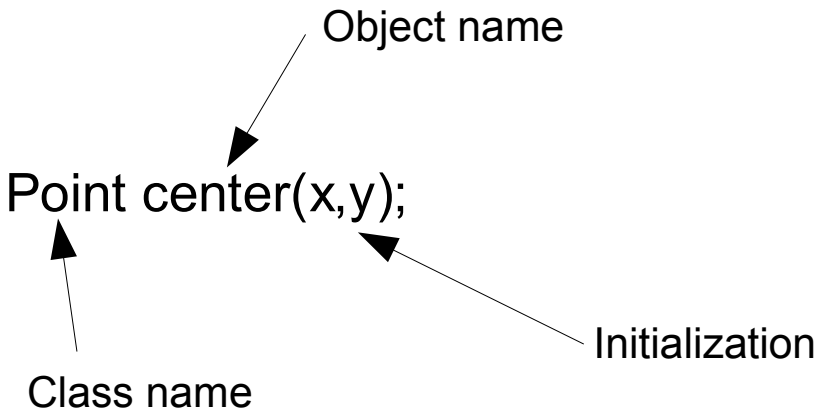
Lecture 15: User Defined Classes

Intro to classes

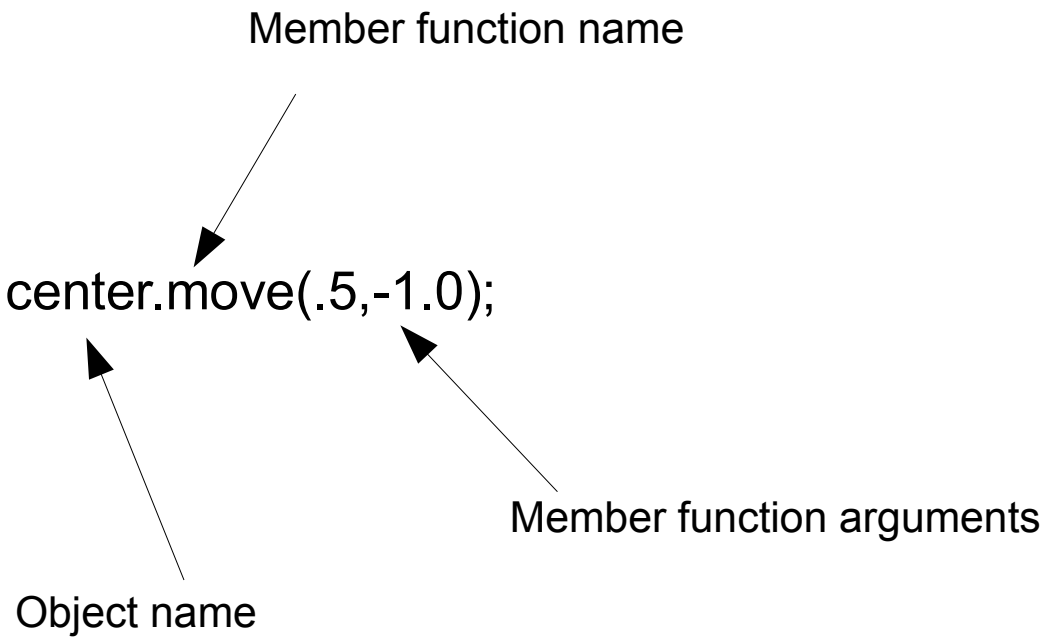
We have already had some practice with classes.

- Employee
- Time
- Point
- Line

Recall that a class is like a souped up variable that can store data, but it also comes with member functions.



Member functions



Classes are awesome!

- Classes make programming easier
- Classes shorten code
- Classes are reusable
- Classes reduce bugs

What other graphics classes would you have liked?

Declaring a class

Anatomy of a class declaration:

```
class class_name{

public:
constructor declarations
member function declarations

private:
member variables
};
```

Public section of the class. These are the constructors and member functions we can access via the dot operator

Internal data of the class. These variables and functions are not accessible outside the class functions.

What goes into public section

Constructor declarations

Constructor is a function and needs to be declared. We need to have a prototype for the constructor.

Recall how we use a constructor:

Point center(.5,1.7); This command evokes the constructor.

Member functions

Any member functions that we can call with the . operator are declared in the public section. Their prototypes are put into the public section.

What goes into the private section

Private variables

These are variables that store essential data of the class. The variables are only accessible by the class itself. We cannot access them (in main) via the dot operator.

Helper functions (private functions)

Sometimes a class will use internal functions that are not member functions themselves.

Point class example

Recall the Point class.

`Point (double x , double y)`	Constructs a point at location (x,y).
`p.move(double dx, double dy)`	Moves the point p by (dx , dy).
`double p.get_y ( )`	Returns the y-coordinate of p.
`double p.get_x ( )`	Returns the x-coordinate of p.

Let us write this class!

Point class declaration

```
class Point
{
public:
    Point(double new_x, double new_y);
    void move(double dx, double dy);
    double get_x() const;
    double get_y() const;
private:
    double x;
    double y;
};
```

Constructor declaration

Mutator member function

Accessors

Private variables. Notice how this is the only data needed to define a Point.

Don't forget the semi-colon!

Point class definitions

We have declared member functions but we have not yet written their definition

We can place the member function definitions immediately below our class declaration.

The general form of a member function definition is:

```
return_type ClassName::function_name (parameters) const
{
** STATEMENTS **
}
```

We add the const modifier at the end if it's an accessor. Should match the cons used in the declaration.

By adding ClassName:: before the function name, we make sure that the program knows this is a member function for that class.

move member function definition

```
void Point::move(double dx, double dy)
{
x+=dx;
y+=dy;
}
```

Functions of this type are called mutators because they change the private variables.

Member functions cont.

```
double Point::get_x() const
{
return x;
}

double Point::get_y() const
{
return y;
}
```

These type of functions are called accessors because they access the private variables for us.

Encapsulation

What is the idea behind this convoluted way of doing things?

One of the ideas is that we don't have to know how the variables in the class are stored or how the class manipulates the variables.

We don't know. We don't have to know. We don't care. This is the idea of ***encapsulation***.

The programmer is isolated from the data and all manipulation done via function calls.

Encapsulation example

```
class DayOfYear
{
public:
              void set(int newmonth, int newday);

Private:
              int month;
              int day;
};
```

Note: This is just a bare bones example. Lets define the set function.

```
void DayOfYear::set(int newmonth, int newday)
{
              month=newmonth;
              day=newday;
}
```

Example cont.

```
int main()
{
DayOfYear today;  // Declares a blank DayOfYear object


today.set(2,15);  // initializes today by using the
                   // set member functions.
}
```

Ok this is all well and good. But I could have written:

```
today.set(435435,3249834);
```

An improved version of set

```
void DayOfYear::set(int newmonth, int newday)
{
    if (newmonth >= 1 && newmonth <=12))
        month=newmonth;
    else
    {
        cout << "Illegal month!";
        exit(1);  // Ends program requires <cstdlib>
    }

    if (newday >=1 && newday <=31)
        day=newday;
    else
    {
        cout << "Illegal day";
        exit(1);
    }
}
```

Constructors

Constructor is a function that declares the instance of the class. It allocates the memory for the object and it may initialize the private variables.

There are two types of constructors. The default constructor which just creates the object and either leaves variables uninitialized or initializes always to default values.

Default constructor syntax:

Declaration:

```
class class_name
{
   public:
      class_name(); // Default constructor
...

};
```

Definition:

```
class_name::class_name()
{
// Private variable initializations
}
```

Example

```
class DayOfYear{

public:
   DayOfYear(); // constructor

   /*
    Other member functions
   */

private:
   int month;
   int day;
};
```

Default constructor definition

```
DayOfYear::DayOfYear()
{
month = 1;
day = 1;
}
```

Example

How to use a default constructor:

```
int main()
{

DayOfYear today;

...

}
```

`today` is a `DayOfYear` object that our constructor creates and initializes `month` to be 1 and `day` to be 1.

Constructor syntax:

Declaration:

```
class class_name
{
   public:
      class_name(type par1, type par2...);
...

};
```

Definition:

```
class_name::class_name(type par1, type par2)
{
// Private variable initializations
}
```

Constructors with arguments example

Declaration:

```
class DayOfYear
{
public:
    DayOfYear(int new_month, int new_day);

    /*
     Other member functions
    */

private:
    int month;
    int day;
};
```

Definition:

```
DayOfYear::DayOfYear(int new_month, int new_day)
{
month= new_month;
day = new_day;
}
```

Using a constructor with arguments

```
int main()
{

    DayOfYear today(5,3);

}
```

Few notes about constructors

If you don't write any constructors the computer creates a default constructor for you!

This default constructor creates the object but does not initialize the variables.

You can have several constructors for your class. One constructor could be the default constructor and the second one that takes arguments.

If you write a constructor that takes arguments you no longer have the computer created default constructor.