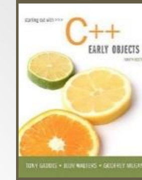
COP2334

Introduction to Object Oriented Programming with C++

D. Singletary



Module 7

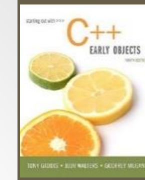
Ch. 7 Introduction to Classes and Objects

FLORIDA STATE COLLEGE at Jacksonville

Abstract Data Types

- An abstract data type (ADT) is a data type that specifies the values the data type can hold and operations it can perform on them without revealing details of how the data type is implemented.
  - An automobile is an example of abstraction; most drivers know how to use them but few know the internal details of how they work.
- In software development we use abstraction to represent objects which exist in an organization's domain
  - The objects contain data ("state") and the operations ("behavior") that can be performed on that data

Using UML to Diagram Abstract Data Types



- Here is an informal (and vastly oversimplified) UML (Unified Modeling Language) diagram of an automobile abstraction:

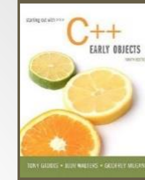
Automobile	
moving: boolean **engineRunning: boolean** **steeringWheel : Wheel** **gasPedal : Pedal** **brakePedal : Pedal** **turnSignalLever : Lever** **engine : Engine**	**data (member variables)**
turn(direction : EnumDirection) : void **accelerate(rate : int, speed : int) : void** **brake() : void** **signal(direction: EnumDirection, time : int) : void** **startOrStopEngine(choice: boolean) : void** **isMoving() : boolean** **isEngineRunning() : boolean**	**behavior (member functions)**

C++ Classes

- Abstract data types in C++ are represented by <u>classes</u>

- Consider this UML diagram of a Circle:

Circle
PI : const double **radius : double**
getRadius() : double **setRadius(radius : double) : void** **calculateArea() : double**



```
class Circle
{
    private:
        const double PI = 3.14159;
        double radius;

    public:
        double getRadius() { return radius; }
        void setRadius(double r)  { radius = r; }
        double calculateArea()  {
                return PI * radius * radius;
        }
};
```

Access Specifiers

- Access specifiers designate who can access class members
  - <u>public</u> members can be accessed by outside functions (member variables and member functions)
  - <u>private</u> members can only be accessed by other members of the same class
  - <u>protected</u> members can only be accessed by same class members or extended class members
- The **radius** member variable in the Circle class can <u>only</u> be accessed by the getRadius, setRadius and calculateArea member functions since it is private
- The **getRadius**, **setRadius** and **calculateArea** functions can be accessed by external functions since they are public

- If access specifiers are omitted, access defaults to private
- By convention, member variables are <u>usually</u> (but not always) declared as private and member functions are usually declared as public
  - Restricting access to data in this manner allows us to control how the data is modified, which enhances maintability and code portability.
  - Attempting to access a private member outside of the class results in a compiler error

error: 'double Circle::radius' is private

```
// access violation example
int main()
{
    Circle c;
    c.radius = 5.0; // radius is private
    return 0;
}
```

error: 'double Circle::radius' is private

- Members can be specified in any order:

```
class Circle
{
    public:
        double getRadius() { return radius; }
        void setRadius(double r)  { radius = r; }
        double calculateArea()  {
            return PI * radius * radius;
        }
    private:
        const double PI = 3.14159;
        double radius;
};
```

- Access specifiers can be split

```
class Circle
{
    public:
        double getRadius() { return radius; }
         void setRadius(double r)  { radius = r; }
    private:
        const double PI = 3.14159;
        double radius;
    public:
        double calculateArea()  {
            return PI * radius * radius;
        }
};
```

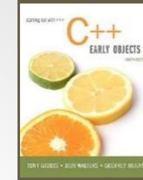
- By convention, private members are grouped together at the top of the class and public members are grouped <u>after</u> the private block
  - <u>This is the convention we will follow in this course</u>

```
class Circle
{
private:
    const double PI = 3.14159;
    double radius;
public:
    double getRadius() { return radius; }
    void setRadius(double r)  { radius = r; }

    double calculateArea()  {
        return PI * radius * radius;
    }
};
```

private block

public block

Creating and Using Objects



- Objects are instances of classes
  - They do not exist in memory until they are instantiated
- A class can be considered a blueprint
  - An object is an implementation of the thing described by the blueprint
- We can instantiate objects by using simple declarations, just as we do for basic variables

```
Circle circle1, circle2;
```

- We use the dot operator to access a class's public members

```
Circle circle1, circle2;

circle1.setRadius(5.0);
circle2.setRadius(7.5);

cout << "circle 1 area = " <<
        circle1.calculateArea() << endl;
cout << "circle 2 area = " <<
        circle2.calculateArea() << endl;
```

- If radius was public in the Circle class, we could access it directly, also using the dot operator:

```
Circle circle1, circle2;

circle1.radius = 5.0;
circle2.radius = 7.5;
```

- But since it is private, we can't do this (which is a good thing -- why?)

- The dot operator is not necessary for members inside the class definition. The radius member variable in the Circle class is accessed without a dot:

```
double getRadius() { return radius; }
void setRadius(double r)  { radius = r; }
double calculateArea()  {
        return PI * radius * radius;
}
```

```
// circledemo.cpp
// demonstrates the circle class
#include <iostream>
#include <iomanip>
using namespace std;

// declare the Circle class
class Circle
{
	private:
		const double PI = 3.14159;
		double radius = 0;

	public:
		double getRadius() { return radius; }
		void setRadius(double r) { radius = r; }
		double calculateArea()  { return PI * radius * radius; }
};
```

// main() is separate from the class definition!

```
int main()
{
      Circle circle1, circle2;  // instantiate 2 circles

       // initialize and display area of both circles
      circle1.setRadius(5.0);
      circle2.setRadius(7.5);
      cout << fixed << setprecision(2) << "circle 1 area = " <<
              circle1.calculateArea() << endl;
      cout << fixed << setprecision(2) << "circle 2 area = " <<
              circle2.calculateArea() << endl;

      return 0;
}
```

Accessors and Mutators

- The getRadius member function in the Circle class allows a caller to access the radius (via a return statement) but does not modify it. This is an accessor, or getter function:

  double getRadius() { return radius; }

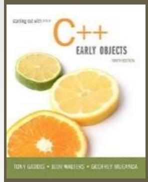
- By convention, accessor names use a combination of "get" and the member variable name ("is" for boolean member variables)

- The setRadius member function in the Circle class is known as a <u>mutator</u>, or <u>setter</u>, because it modifies the value of the radius member variable.

void setRadius(double r) { radius = r; }

- Mutators are preferable to directly accessing the member data because it makes our code more portable and easier to debug by <u>encapsulating</u> (protecting) our data.
  - We can also validate before assigning
- By convention, mutator names use a combination of "set" and the member variable name

Defining Member Functions

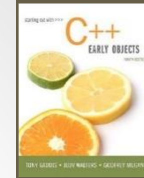


- Class member functions can be declared either inside or outside the class declaration
- Inline functions are member functions that are defined within the class declaration (getRadius(), setRadius() and calculateArea() from the Circle class).
  - These are suitable for short methods, particularly accessors and mutators
  - Avoid making longer function bodies inline since this can clutter the class declaration

FLORIDA STATE COLLEGE at Jacksonville

- To declare a function outside of the class declaration, use a function prototype in the class and then use a <u>function implementation block</u> containing the body

```
class Circle
{
...
      public:
            double getRadius() { return radius; }
            void setRadius(double r) { radius = r; }
            double calculateArea();
};

double Circle::calculateArea() {
      return PI * radius * radius;
}
```



- Non-inline functions are implemented as

returnType className::funcName(parameters) { body }

- The :: symbol is the scope resolution operator, used to tell the compiler that this is a member function and what class it belongs to.

- Correct:

double Circle::calcArea() {

- Incorrect:

double calcArea() { // missing class name and ::

Circle::double calcArea() { // return type not at beginning

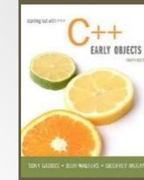
The Circle Class Using Inline Prototypes

```
// declare the Circle class
class Circle
{
   private:
      const double PI = 3.14159;
      double radius = 0;
   public:
      double getRadius();
      void setRadius(double r);
      double calculateArea();
};
```

Declarations

```
double Circle::getRadius() { return radius; }
void Circle::setRadius(double r) { radius = r; }
double Circle::calculateArea() { return PI * radius * radius; }
```

Implementations



Constructors

- A constructor is a member function that is automatically called when a class object is created
- If not provided explicitly by the programmer, an implicit default (0-arg) constructor is provided
- A constructor has the same name as the class and has no return type (not even void)

```
Circle::Circle() { radius = 1.0; }
```

The Circle Class with a Constructor

```
// declare the Circle class
class Circle
{
   private:
      const double PI = 3.14159;
      double radius = 0;
   public:
      Circle();
      double getRadius();
      void setRadius(double r);
      double calculateArea();
};

Circle::Circle() { radius = 1.0; }
double Circle::getRadius() { return radius; }
void Circle::setRadius(double r) { radius = r; }
double Circle::calculateArea() { return PI * radius * radius; }
```

```
int main()
{
   Circle circle1, circle2;  // instantiate 2 circles

   cout << fixed << setprecision(2) <<
        "default circle radius for circle1 = " <<
        circle1.getRadius() << endl;
   cout << fixed << setprecision(2) <<
        "default circle radius for circle2 = " <<
        circle2.getRadius() << endl;
   ...
```

```
default circle radius for circle1 = 1.00
default circle radius for circle2 = 1.00
```

Overloading Constructors

```
class Circle
{
  private:
    const double PI = 3.14159;
    double radius = 0;
  public:
    Circle() { radius = 1.0; }
    Circle(double r) { radius = r; }
    double getRadius();
    void setRadius(double r);
    double calculateArea();
};
```

```
int main()
{
    Circle circle1, circle2(5.0);  // instantiate 2 circles

    cout << fixed << setprecision(2) <<
        "radius for circle1 = " <<
        circle1.getRadius() << endl;
    cout << fixed << setprecision(2) <<
        "radius for circle2 = " <<
        circle2.getRadius() << endl;
   ...
```

```
radius for circle1 = 1.00
radius for circle2 = 5.00
```

Default Constructors

- If we create a constructor with one or more arguments (ie. overload the default constructor), we must explicitly create a default (0-argument) constructor if we instantiate the object without a constructor in our code:

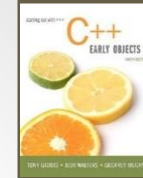
```
class Circle
{
   ...
   public:
      // commented out to test
      // Circle() { radius = 1.0; }
      Circle(double r) { radius = r; }
   ...
};
```

error: no matching function for call to 'Circle::Circle()'

Destructors

- A destructor is a member function that is automatically called when an object is destroyed
  - Destructors names begin with a tilde (~)
  - This gives us an opportunity to perform cleanup activities that may be necessary (e.g. closing a file, releasing memory)

```
class Circle
{
    ...
    public:
        Circle();
        Circle(double r);
        ~Circle();
    ...
};
```



- Destructors have no return type
- Destructors never have a parameter list
- There can only be one destructor

```
Circle::Circle() {
    radius = 1.0; cout << "default constructor called" << endl;
}

Circle::Circle(double r) {
    radius = r; cout << "1-arg constructor called" << endl;
}

Circle::~Circle() {
    cout << "destructor for radius " << radius << " called" << endl;
}
```

```
int main()
{
    Circle circle1, circle2(5.0);  // instantiate 2 circles

    cout << fixed << setprecision(2) <<
        "radius for circle1 = " <<
        circle1.getRadius() << endl;
    cout << fixed << setprecision(2) <<
        "radius for circle2 = " <<
        circle2.getRadius() << endl;

    return 0;
}
```

```
default constructor called
1-arg constructor called
radius for circle1 = 1.00
radius for circle2 = 5.00
destructor for radius 5.00 called
destructor for radius 1.00 called
```

Passing Objects to Functions

- Objects can be passed to functions:

```
void displayCircle(Circle c)
{
   cout << fixed << setprecision(2) <<
         "displayCircle: radius = " << c.getRadius() <<
         " and area = " << c.calculateArea() << endl;
}
```

- In this example the Circle object is being passed by value (a copy is being made)

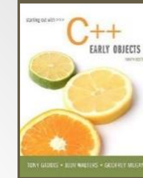
```
void displayCircle(Circle c);

int main()
{
   Circle circle1, circle2(5.0);  // instantiate 2 circles

   displayCircle(circle1);
   displayCircle(circle2);

   return 0;
}

void displayCircle(Circle c)
{
   cout << fixed << setprecision(2) <<
         "displayCircle: radius = " << c.getRadius() <<
         " and area = " << c.calculateArea() << endl;
}
```

```
displayCircle: radius = 1.00 and area = 3.14
displayCircle: radius = 5.00 and area = 78.54
```

- Pass by value means modifications to the object only apply locally

```
void displayCircle(Circle c);

int main()
{
    Circle circle1;

    displayCircle(circle1);
    cout << "back in main: radius is " << circle1.getRadius() << endl;

    return 0;
}

void displayCircle(Circle c)
{
    cout << "in displayCircle: setting radius to 10" << endl;
    c.setRadius(10);
    cout << fixed << setprecision(2) <<
        "displayCircle: radius = " << c.getRadius() <<
        " and area = " << c.calculateArea() << endl;
}
```

```
in displayCircle: setting radius to 10
displayCircle: radius = 10.00 and area = 314.16
back in main: radius is 1.00
```



- Pass by reference to modify the calling object

```
void setAndDisplayCircle(Circle& c);

void setAndDisplayCircle(Circle& c)
{
    cout << "in setAndDisplayCircle: setting radius to 10" << endl;
    c.setRadius(10);
    cout << fixed << setprecision(2) <<
        "displayCircle: radius = " << c.getRadius() <<
        " and area = " << c.calculateArea() << endl;
}
```

```
void setAndDisplayCircle(Circle& c);

int main()
{
	Circle circle1;

	setAndDisplayCircle(circle1);
	cout << "back in main: radius is " << circle1.getRadius() << endl;

	return 0;
}
```

```
in setAndDisplayCircle: setting radius to 10
displayCircle: radius = 10.00 and area = 314.16
back in main: radius is 10.00
```

```
void setAndDisplayCircle(Circle& c)
{
	cout << "in setAndDisplayCircle: setting radius to 10" << endl;
	c.setRadius(10);
	cout << fixed << setprecision(2) <<
		"displayCircle: radius = " << c.getRadius() <<
		" and area = " << c.calculateArea() << endl;
}
```

Separating Class Declarations

- Usually class declarations are stored in their own header files and member function definitions are stored in their own .cpp file
  - A header file containing a class declaration is known as a class specification file. The file will usually have the same name as the class with a '.h' extension, e.g. "Rectangle.h".
  - The functions are implemented in a class implementation file, e.g. "Rectangle.cpp".

The #define Preprocessor Directive

- Preprocessor directives are processed by the preprocessor <u>before a file is compiled</u>
- #define is used to define a textual substitution operation before code is compiled

```
#define HELLO "Hello"  // replace HELLO in code
#define DEBUG 1        // replace DEBUG with 1
#define YES "no"       //  fireable offense
```

- #define is an "old-school" method of defining constants (e.g. C language)
  - can be used anywhere, but usually at top of files (after #include directives)

The #ifdef Preprocessor Directive

- #ifdef enables conditional compilation
  - if the specified symbol has been defined, the code is included and compiled, otherwise it is ignored
- Must be terminated with #endif

```
#ifdef SYMBOL
cout << "Hello!" << endl;  // only prints if SYMBOL is defined
#endif
```

```
#define DEBUG 1

int main()
{
      string s = "0123456789";

#ifdef DEBUG
      // DEBUG defined, this will compile and run!
      cout << "s = " << s << endl;
#endif // DEBUG_MY_CODE

      return 0;
}
```

The #ifndef Preprocessor Directive

- #ifndef works the opposite as #ifdef: if a symbol has been defined, the enclosed code is ignored

```
int main()
{
	string s = "0123456789";
#ifndef IGNORE_ME
	// IGNORE_ME not defined, this will compile and run!
	cout << "s = " << s << endl;
#endif // IGNORE_ME
```

Using #ifndef and #define in Header Files

- #ifndef and #define are frequently used together as include guards to prevent contents of header files from being included multiple times
- e.g. the include file iostream:

```
#ifndef _GLIBCXX_IOSTREAM
#define _GLIBCXX_IOSTREAM 1
...  // contents of iostream
#endif /* _GLIBCXX_IOSTREAM */
```

- Convention is to use name of header file with underscores as shown above

Rectangle.h

```
// Rectangle.h - Rectangle class specification
#ifndef _RECTANGLE_H
#define _RECTANGLE_H

class Rectangle
{
        private:
                ...
        public:
                ...
};
#endif // _RECTANGLE_H
```

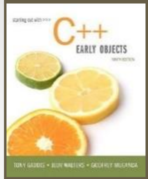
include guard

Rectangle.cpp

```
// Rectangle.cpp - Rectangle class implementation
using namespace std;
#include "Rectangle.h"

// default constructor
Rectangle::Rectangle() { width = 1; height = 1; }

// 2-arg (overloaded constructor)
Rectangle::Rectangle(int w, int h) {
	width = 1;
	height = 1;
}
...
```

Structures

- C++ allows a set of variables to be combined into a single unit called a structure
  - Structures are common in C, not so common in C++
  - Although classes are more commonly used in C++, structures are still used
  - Defined using the keyword <u>struct</u> instead of <u>class</u>



FLORIDA STATE COLLEGE at Jacksonville

```
struct Payroll
{
    int       empNumber;
    string   name;
    double hours, payRate, grossPay;
};
```

- Important difference: members of a structure are <u>public</u> by default

```
// structdemo.cpp
// demonstrates a structure in C++

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

struct Payroll
{
   int       empNumber;
   string    name;
   double    hours, payRate, grossPay;
};
```

```
int main()
{
   struct Payroll pRoll;

   pRoll.empNumber = 50;
   pRoll.name = "Smith, John";
   pRoll.hours = 40;
   pRoll.payRate = 9.40;
   pRoll.grossPay = pRoll.hours * pRoll.payRate;

   cout << "Pay for " << pRoll.name << " = " <<
          setprecision(2) << fixed << pRoll.grossPay;

   return 0;
}
```

```
Pay for Smith, John = 376.00
```

- Structures can declare member functions (including constructors) just as classes can:

```
struct Payroll
{
   int      empNumber;
   string   name;
   double hours, payRate, grossPay;

   Payroll(int eNum, string nm,
           double h, double pRate)
   {
         empNumber = eNum;
         name = nm;
         hours = h;
         payRate = pRate;
         grossPay = hours * payRate;
   }
};
```

```
struct Payroll
{
	int	empNumber;
	string name;
	double hours, payRate, grossPay;

	Payroll(int eNum, string nm,
		double h, double pRate)
	{
		empNumber = eNum;
		name = nm;
		hours = h;
		payRate = pRate;
		grossPay = hours * payRate;
	}
};


int main()
{
	struct Payroll pRoll(50, "Smith, John", 40, 9.40);

	cout << "Pay for " << pRoll.name << " = " << setprecision(2) << fixed << pRoll.grossPay;

	return 0;
}
```

Struct or Class?

- Other than the default member visibility difference, structs are effectively identical to classes. Which to use?
- Conventionally, structs are used for features requiring "plain old data"
  - data structures with no need for associated behavior or other OOP features
- We will use classes in this course, but you need to be aware of structs since they are available for use in the language