# Objects and Classes

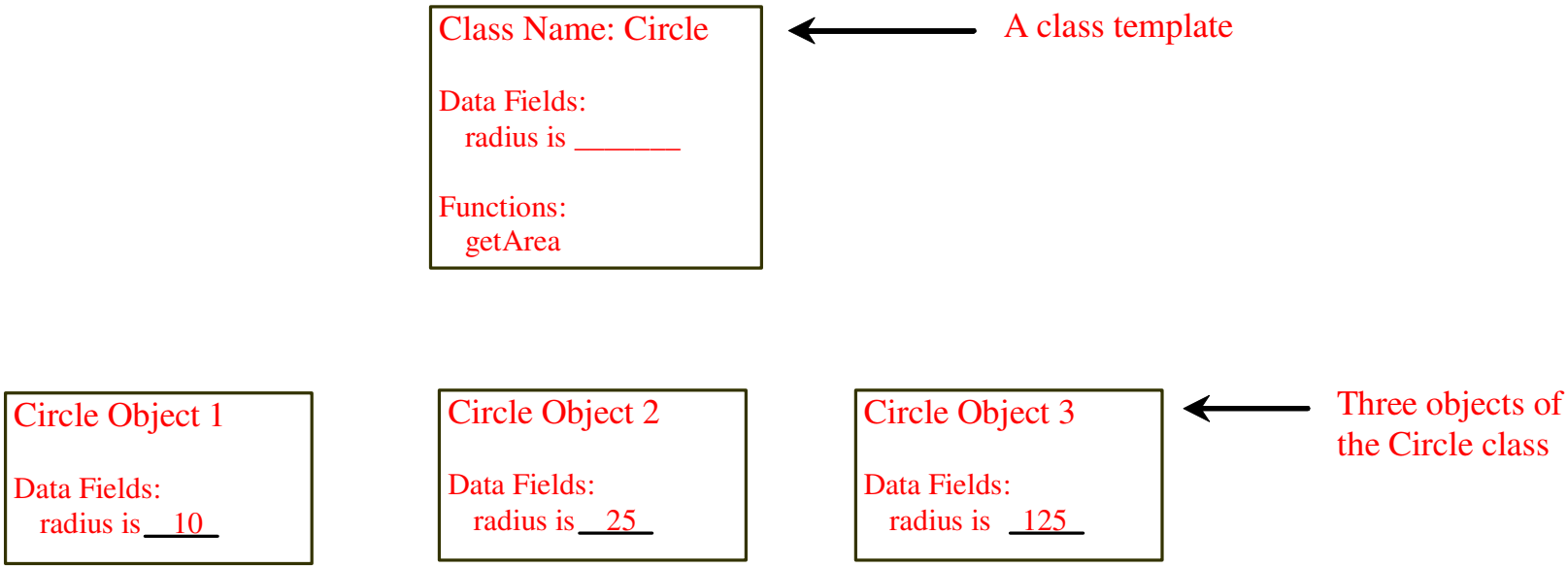
# Objectives

- To describe objects and classes, and to use classes to model objects.
- To use UML graphical notations to describe classes and objects.
- To demonstrate defining classes and creating objects.
- To create objects using constructors.
- To access data fields and invoke functions using the object member access operator (**.**).
- To separate a class definition from a class implementation.
- To prevent multiple inclusions of header files using the **#ifndef** inclusion guard directive.
- To know what inline functions in a class are.
- To declare private data fields with appropriate **get** and **set** functions for data field encapsulation and make classes easy to maintain.
- To understand the scope of data fields.
- To apply class abstraction to develop software.

# OO Programming Concepts

Object-oriented programming (OOP) involves programming using objects. An *object* represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects. An object has a unique identity, state, and behaviors. The *state* of an object consists of a set of *data fields* (also known as *properties*) with their current values. The *behavior* of an object is defined by a set of functions.

# Objects



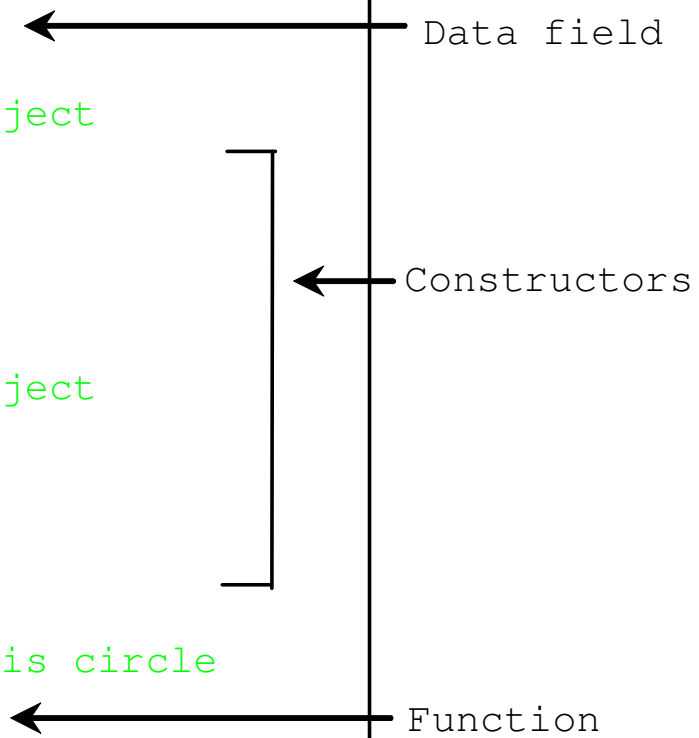

An object has both a state and behavior. The state defines the object, and the behavior defines what the object does.

# Classes

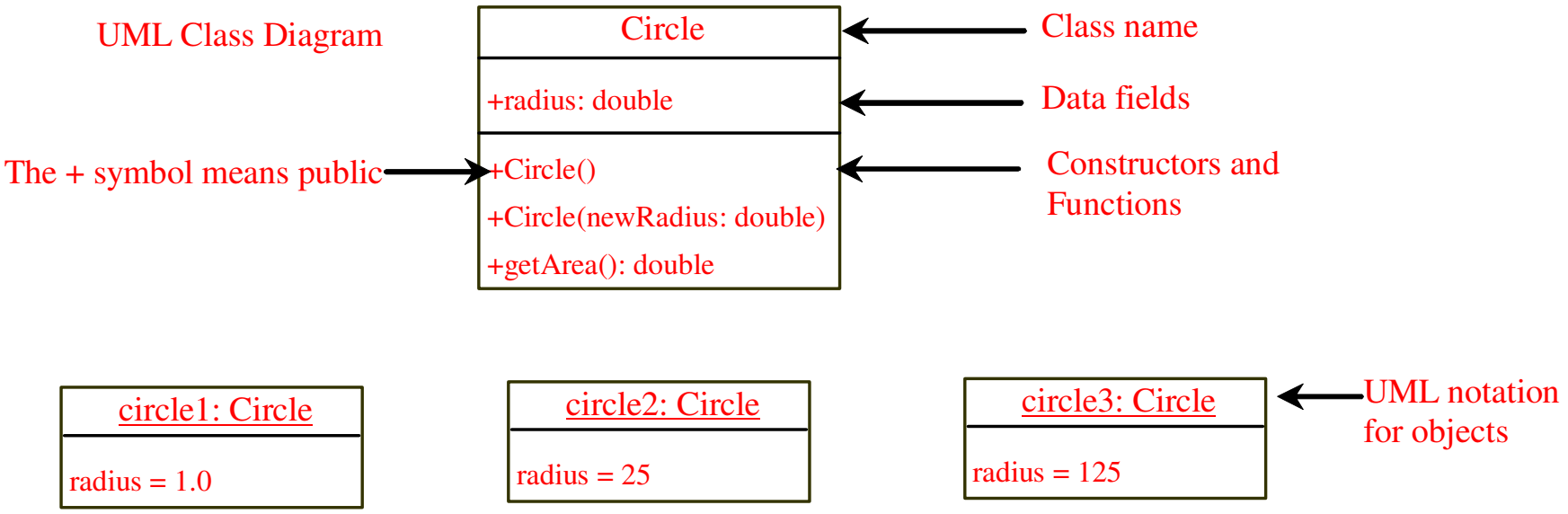
*Classes* are constructs that define objects of the same type. A class uses variables to define data fields and functions to define behaviors. Additionally, a class provides a special type of functions, known as constructors, which are invoked to construct objects from the class.

# Classes

```
class Circle
{
public:
  // The radius of this circle
  double radius;

  // Construct a circle object
  Circle()
  {
    radius = 1;
  }

  // Construct a circle object
  Circle(double newRadius)
  {
    radius = newRadius;
  }

  // Return the area of this circle
  double getArea()
  {
    return radius * radius * 3.14159;
  }
};
```



# UML Class Diagram



# A Simple Circle Class

- Objective: Demonstrate creating objects, accessing data, and using functions.

TestCircle

# Example: Defining Classes and Creating Objects

| TV | |
|---|---|
| channel: int | The current channel (1 to 120) of this TV. |
| volumeLevel: int | The current volume level (1 to 7) of this TV. |
| on: bool | Indicates whether this TV is on/off. |
| +TV() | Constructs a default TV object. |
| +turnOn(): void | Turns on this TV. |
| +turnOff(): void | Turns off this TV. |
| +setChannel(newChannel: int): void | Sets a new channel for this TV. |
| +setVolume(newVolumeLeve: int): void | Sets a new volume level for this TV. |
| +channelUp(): void | Increases the channel number by 1. |
| +channelDown(): void | Decreases the channel number by 1. |
| +volumeUp(): void | Increases the volume level by 1. |
| +volumeDown(): void | Decreases the volume level by 1. |

TV

# Constructors

The constructor has exactly the same name as the defining class. Like regular functions, constructors can be overloaded (i.e., multiple constructors with the same name but different signatures), making it easy to construct objects with different initial data values.

A class normally provides a constructor without arguments (e.g., Circle()). Such constructor is called a *no-arg* or *no-argument constructor*.

A class may be declared without constructors. In this case, a no-arg constructor with an empty body is implicitly declared in the class. This constructor, called *a default constructor*, is provided automatically *only if no constructors are explicitly declared in the class*.

# Constructors, cont.

A constructor with no parameters is referred to as a *no-arg constructor*.

- Constructors must have the same name as the class itself.
- Constructors do not have a return type—not even void.
- Constructors play the role of initializing objects.

# Copy Constructors

- Special constructor used when a newly created object is initialized to the data of another object of same class
- Default copy constructor copies field-to-field, using memberwise assignment
- The default copy constructor works fine in most cases

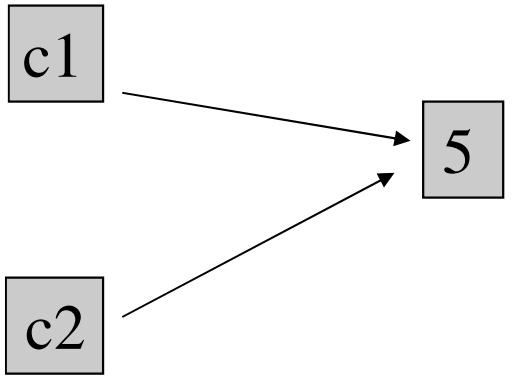
# Copy Constructors

Problems occur when objects contain pointers to dynamic storage:

```
class MyClass {

  private:
   int *p;

  public:
   MyClass(int v=0){ p= new int; *p= v;}
  ~MyClass(){delete p;}
 };
```

# Default Constructor Causes Sharing of Storage

```
MyClass c1(5);
if (true)
{
  MyClass c2=c1;
}
// c1 is corrupted
// when c2 goes
// out of scope and
// its destructor
// executes
```



# Problems of Sharing Dynamic Storage

- Destructor of one object deletes memory still in use by other objects
- Modification of memory by one object affects other objects sharing that memory

# Programmer-Defined Copy Constructors

- A copy constructor is one that takes a reference parameter to another object of the same class
- The copy constructor uses the data in the object passed as parameter to initialize the object being created
- Reference parameter should be **`const`** to avoid potential for data corruption

# Programmer-Defined Copy Constructors

- The copy constructor avoids problems caused by memory sharing
- Can allocate separate memory to hold new object's dynamic member data
- Can make new object's pointer point to this memory
- Copies the data, not the pointer, from the original object to the new object

# Copy Constructor Example

```
class MyClass
{
    int *p;
  public:
    MyClass(const MyClass &obj)
     { p= new int; *p= *obj.p; }

    MyClass(int v=0){ p= new int; *p= v; }
    ~MyClass(){delete p;}
};
```

# Copy Constructor – When Is It Used?

A copy constructor is called when

- An object is initialized from an object of the same class
- An object is passed by value to a function
- An object is returned using a **`return`** statement from a function

# Object Names

In C++, you can assign a name when creating an object. A constructor is invoked when an object is created. The syntax to create an object using the no-arg constructor is

```
ClassName objectName;
```

For example,

```
Circle circle1;
```

# Constructing with Arguments

The syntax to declare an object using a constructor with arguments is

```
ClassName objectName(arguments);
```

For example, the following declaration creates an object named circle2 by invoking the Circle class's constructor with a specified radius 5.5.

```
Circle circle2(5.5);
```

# Access Operator

After an object is created, its data can be accessed and its functions invoked using the dot operator (.), also known as the *object member access operator*:

objectName.dataField references a data field in the object.

objectName.function(arguments) invokes a function on the object.

# Naming Objects and Classes

When you declare a custom class, capitalize the first letter of each word in a class name; for example, the class names Circle, Rectangle, and Desk. The class names in the C++ library are named in lowercase. The objects are named like variables.

# Class is a Type

You can use primitive data types to declare variables. You can also use class names to declare object names. In this sense, a class is also a data type.

# Memberwise Copy

In C++, you can also use the assignment operator = to copy the contents from one object to the other. By default, each data field of one object is copied to its counterpart in the other object. For example,

```
circle2 = circle1;
```

copies the radius in circle1 to circle2. After the copy, circle1 and circle2 are still two different objects, but with the same radius.

# Constant Object Name

Object names are like array names. Once an object name is declared, it represents an object. It cannot be reassigned to represent another object. In this sense, an object name is a constant, though the contents of the object may change.

# Anonymous Object

Most of the time, you create a named object and later access the members of the object through its name. Occasionally, you may create an object and use it only once. In this case, you don't have to name the object. Such objects are called *anonymous objects*.

The syntax to create an anonymous object using the no-arg constructor is

```
ClassName()
```

The syntax to create an anonymous object using the constructor with arguments is

```
ClassName(arguements)
```

# Class Replaces struct

The C language has the struct type for representing records. For example, you may define a struct type for representing students as shown in (a).

```
struct Student
{
  int id;
  char firstName[30];
  char mi;
  char lastName[30];
};
```

(a)

```
class Student
{
public:
  int id;
  char firstName[30];
  char mi;
  char lastName[30];
};
```

(b)

# Separating Definition from Implementation

C++ allows you to separate class definition from implementation. The class definition describes the contract of the class and the class implementation implements the contract. The class declaration simply lists all the data fields, constructor prototypes, and the function prototypes. The class implementation implements the constructors and functions. The class declaration and implementation are in two separate files. Both files should have the same name, but with different extension names. The class declaration file has an extension name .h and the class implementation file has an extension name .cpp.

Circle.h

Circle.cpp

TestCircleWithHeader.cpp

# Preventing Multiple Inclusions

It is a common mistake to include the same header file in a program multiple times inadvertently. Suppose Head1.h includes Circle.h and TestHead1.cpp includes both Head1.h and Circle.h, as shown in next slide.

# Preventing Multiple Declarations

**Head1.h**

```
#include "Circle.h"
// Other code in Head1.h omitted
```

**TestHead1.cpp**

```
#include "Circle.h"
#include "Head1.h"
int main()
{
   // Other code in TestHead.cpp
   // omitted
}
```

# inclusion guard

```
#ifndef CIRCLE_H
#define CIRCLE_H
class Circle
{
public:
  // The radius of this circle
  double radius;
  // Construct a default circle object
  Circle();
  // Construct a circle object
  Circle(double);
  // Return the area of this circle
  double getArea();
}; // Semicolon required
#endif
```

# Inline Declaration

Inline functions play an important role in class declarations. When a function is implemented inside a class declaration, it automatically becomes an inline function. This is also known as *inline declaration*.

Short functions are good candidates for inline functions, but long functions are not.

# Inline Declaration Example

For example, in the following declaration for class A, the constructor and function f1 are automatically inline functions, but function f2 is a regular function.

```
class A
{
public:
  A()
  {
    // do something;
  }
  double f1()
  {
    // return a number
  }

  double f2();
};
```

# Inline Functions in Implementation File

There is another way to declare inline functions for classes. You may declare inline functions in the class's implementation file. For example, to declare function f2 as an inline function, precede the inline keyword in the function header as follows:

```
// Implement function as inline
inline double A::f2()
{
  // return a number
}
```

# Data Field Encapsulation

The data fields radius in the Circle class can be modified directly (e.g., circle1.radius = 5). This is not a good practice for two reasons:

- First, data may be tampered.
- Second, it makes the class difficult to maintain and vulnerable to bugs. Suppose you want to modify the Circle class to ensure that the radius is non-negative after other programs have already used the class. You have to change not only the Circle class, but also the programs that use the Circle class. Such programs are often referred to as *clients*. This is because the clients may have modified the radius directly (e.g., myCircle.radius = -5).

# Accessor and Mutator

Colloquially, a get function is referred to as a *getter* (or *accessor*), and a set function is referred to as a *setter* (or *mutator*).

A get function has the following signature:

returnType get*PropertyName*()

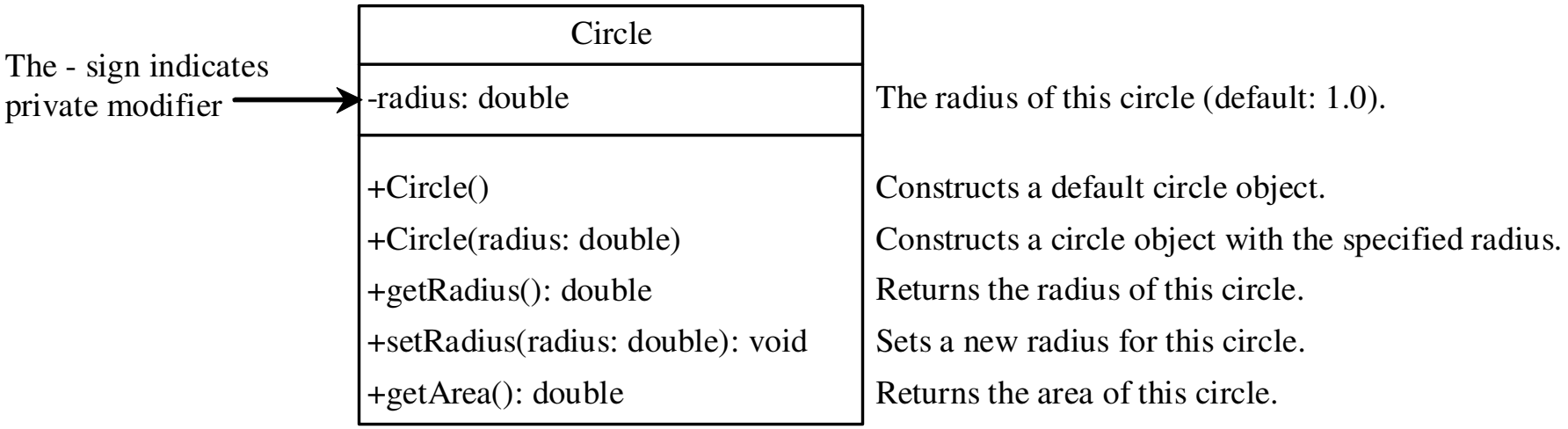
If the returnType is bool, the get function should be defined as follows by convention:

**bool** is*PropertyName*()

A set function has the following signature:

**public void** set*PropertyName*(*dataType propertyValue*)

# Example: New Circle Class




CircleWithPrivateDataFields.h

CircleWithPrivateDataFields.cpp

TestCircleWithPrivateDataFields

# The Scope of Variables

With “Functions,” we discussed the scope of global variables and local variables. Global variables are declared outside all functions and are accessible to all functions in its scope. The scope of a global variable starts from its declaration and continues to the end of the program. Local variables are defined inside functions. The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable.

# The Scope of Variables

The data fields are declared as variables and are accessible to all constructors and functions in the class. In this sense, data fields are like global variables. However, data fields and functions can be declared in any order in a class. For example, all the following declarations are the same:

```
class Circle
{
public:
  Circle();
  Circle(double);
  double getArea();
  double getRadius();
  void setRadius(double);

private:
  double radius;
};
```

(a)

```
class Circle
{
public:
  Circle();
  Circle(double);

private:
  double radius;

public:
  double getArea();
  double getRadius();
  void setRadius(double);
};
```

(b)

```
class Circle
{
private:
  double radius;

public:
  double getArea();
  double getRadius();
  void setRadius(double);

public:
  Circle();
  Circle(double);
};
```

(c)

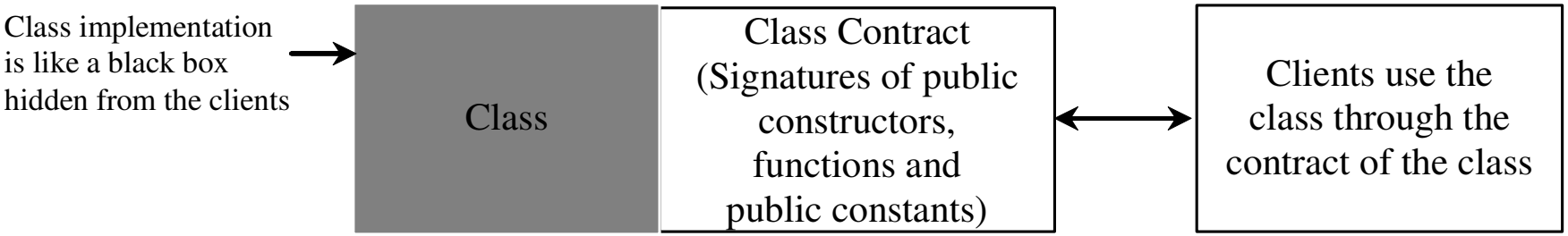
# The Scope of Variables

Local variables are declared and used inside a function locally. If a local variable has the same name as a data field, the local variable takes precedence and the data field with the same name is hidden. For example, in the following program, x is defined as a data field and as a local variable in the function.

HideDataField

# Class Abstraction and Encapsulation

Class abstraction means to separate class implementation from the use of the class. The creator of the class provides a description of the class and let the user know how the class can be used. The user of the class does not need to know how the class is implemented. The detail of implementation is encapsulated and hidden from the user.



# Example: The Loan Class

| Loan | |
|---|---|
| -annualInterestRate: double | The annual interest rate of the loan (default: 2.5). |
| -numberOfYears: int | The number of years for the loan (default: 1) |
| -loanAmount: double | The loan amount (default: 1000). |
| +Loan() | Constructs a default Loan object. |
| +Loan(annualInterestRate: double, numberOfYears: int, loanAmount: double) | Constructs a loan with specified interest rate, years, and loan amount. |
| +getAnnualInterestRate(): double | Returns the annual interest rate of this loan. |
| +getNumberOfYears(): int | Returns the number of the years of this loan. |
| +getLoanAmount(): double | Returns the amount of this loan. |
| +setAnnualInterestRate( annualInterestRate: double): void | Sets a new annual interest rate to this loan. |
| +setNumberOfYears( numberOfYears: int): void | Sets a new number of years to this loan. |
| +setLoanAmount( loanAmount: double): void | Sets a new amount to this loan. |
| +getMonthlyPayment(): double | Returns the monthly payment of this loan. |
| +getTotalPayment(): double | Returns the total payment of this loan. |

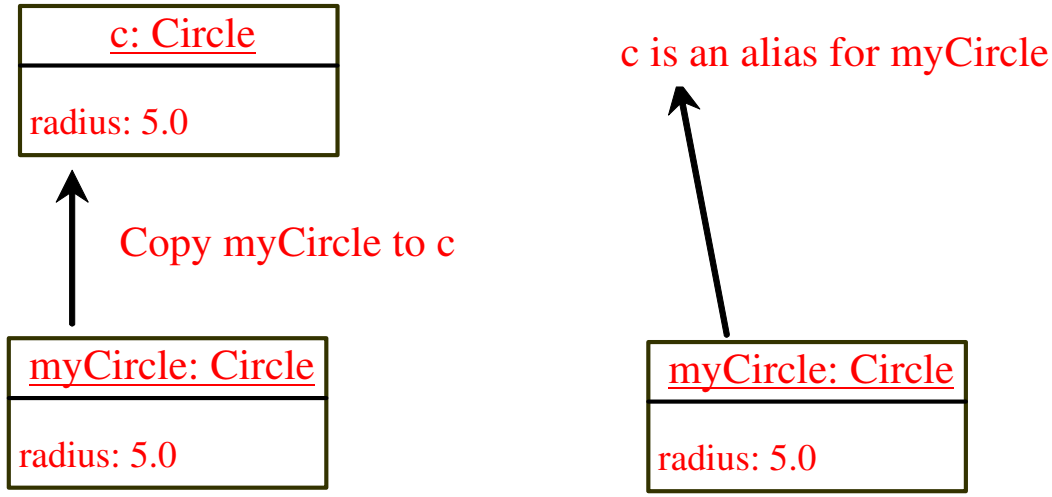
Loan.h

Loan.cpp

TestLoanClass

# Passing Objects to Functions

You can pass objects by value or by reference.




PassObjectByValue

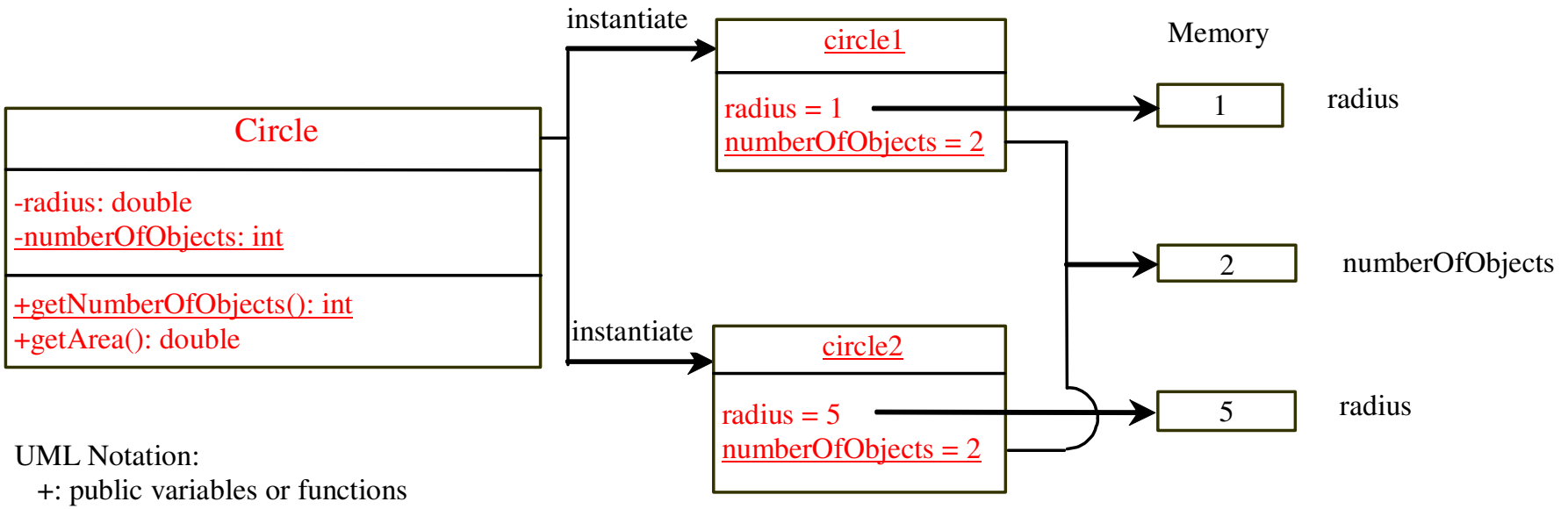
PassObjectByReference

# Array of Objects

Circle circleArray[3] = {Circle(3), Circle(4), Circle(5)};

TotalArea

# Instance and Static Members




UML Notation:
+: public variables or functions
-: private variables or functions
underline: static variables or functions

CircleWithStaticDataFields.h

TestCircleWithStaticDataFields

CircleWithStaticDataFields.cpp

# Use Class Name

Use ClassName::functionName(arguments) to invoke a static function and ClassName::staticVariable. This improves readability because the user can easily recognize the static function and data in the class.

# Instance or Static?

How do you decide whether a variable or function should be instance or static? A variable or function that is dependent on a specific instance of the class should be an instance variable or function. A variable or function that is not dependent on a specific instance of the class should be a static variable or function. For example, every circle has its own radius. Radius is dependent on a specific circle. Therefore, radius is an instance variable of the Circle class. Since the getArea function is dependent on a specific circle, it is an instance function. Since numberOfObjects is not dependent on any specific instance, it should be declared static.

# Constant Member Functions

You can use the const keyword to specify a constant parameter to tell the compiler that the parameter should not be changed in the function. C++ also enables you to specify a constant member function to tell the compiler that the function should not change the value of any data fields in the object. To do so, place the const keyword at the end of the function header.

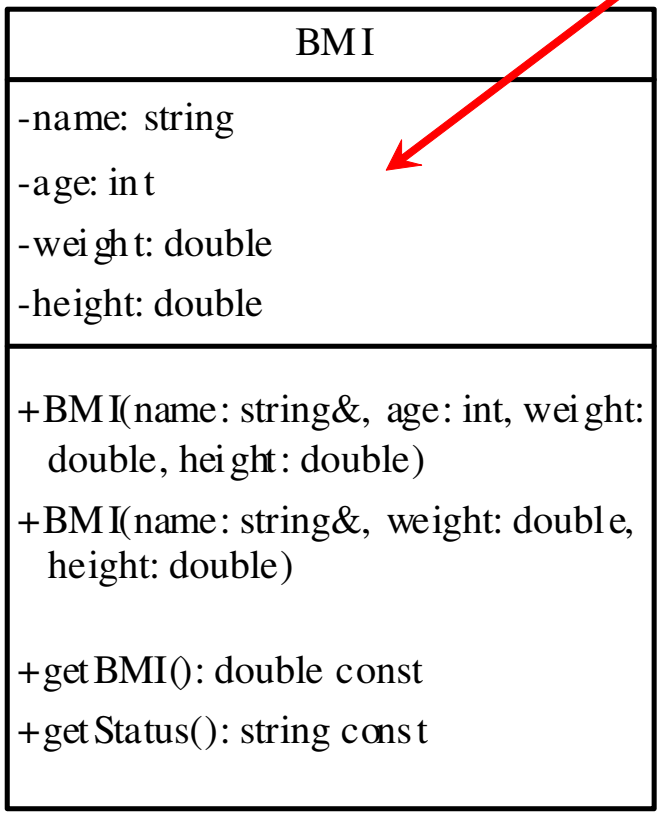
CircleWithConstantMemberFunction.h

CircleWithConstantMemberFunction.cpp

# Object-Oriented Thinking

We have introduced fundamental programming techniques for problem solving using loops, functions, and arrays. The study of these techniques lays a solid foundation for object-oriented programming. Classes provide more flexibility and modularity for building reusable software. Lets see an example!

# Another OOP example: The BMI Class




The get functions for these data fields are provided in the class, but omitted in the UML diagram for brevity.

The name of the person.

The age of the person.

The weight of the person in pounds.

The height of the person in inches.

Creates a BMI object with the specified name, age, weight, and height.

Creates a BMI object with the specified name, weight, height, and a default age 20.

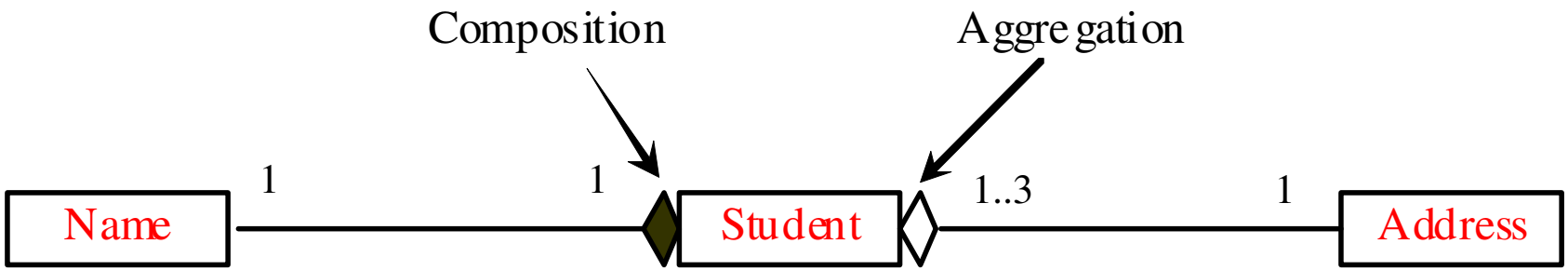
Returns the BMI.

Returns the BMI status (e.g., Normal, Overweight, etc.)

BMI.h BMI.cpp UseBMIClass

# Object Composition

Composition is actually a special case of the aggregation relationship. Aggregation models *has-a* relationships and represents an ownership relationship between two objects. The owner object is called an *aggregating object* and its class an *aggregating class*. The subject object is called an *aggregated object* and its class an *aggregated class*.



# Class Representation

An aggregation relationship is usually represented as a data field in the aggregating class. For example, the relationship in the preceding figure can be represented as follows:

```
class Name
{
  ...
}
```

Aggregated class

```
class Student
{
private:
  Name name;
  Address address;

  ...
}
```

Aggregating class

```
class Address
{
  ...
}
```

Aggregated class

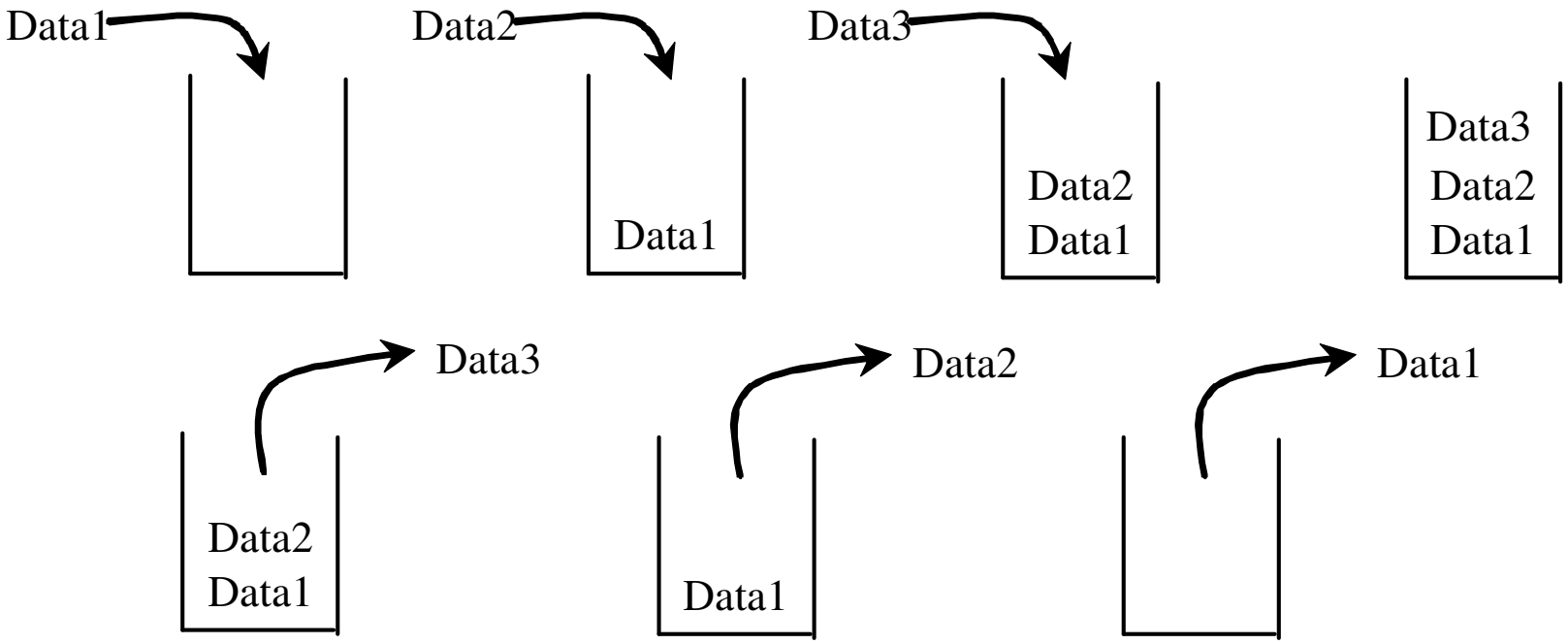
# Aggregation or Composition

Since aggregation and composition relationships are represented using classes in the same way, we don’t differentiate them and call both compositions.

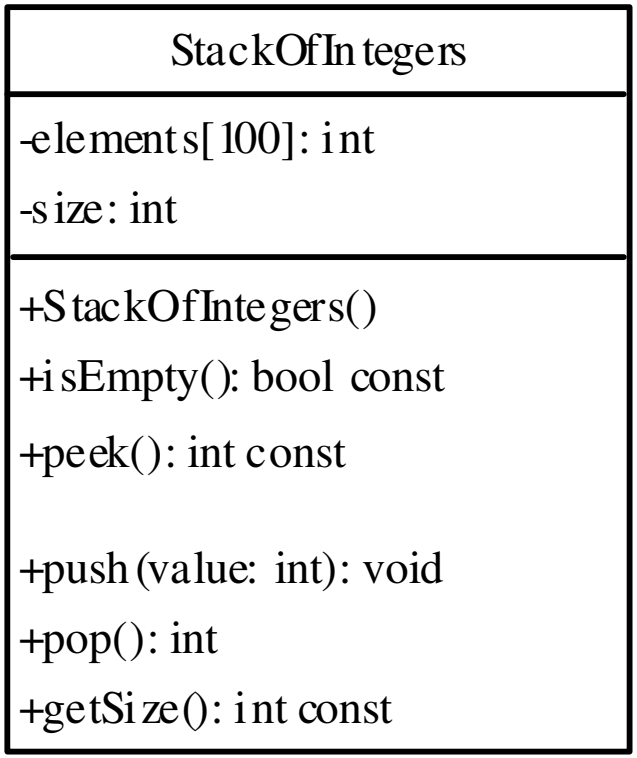
# The StackOfInteger Class

A *stack* is a data structure that holds objects in a last-in first-out fashion, as illustrated in the next figure. It has many applications. For example, the compiler uses a stack to process function invocations. When a function is invoked, the parameters and local variables of the function are pushed into a stack. When a function calls another function, the new function's parameters and local variables are pushed into the stack. When a function finishes its work and returns to its caller, its associated space is released from the stack.

# The StackOfInteger Class



# Example: The `StackOfIntegers` Class




An array to store integers in the stack.

The number of integers in the stack.

Constructs an empty stack.

Returns true if the stack is empty.

Returns the integer at the top of the stack without removing it from the stack.

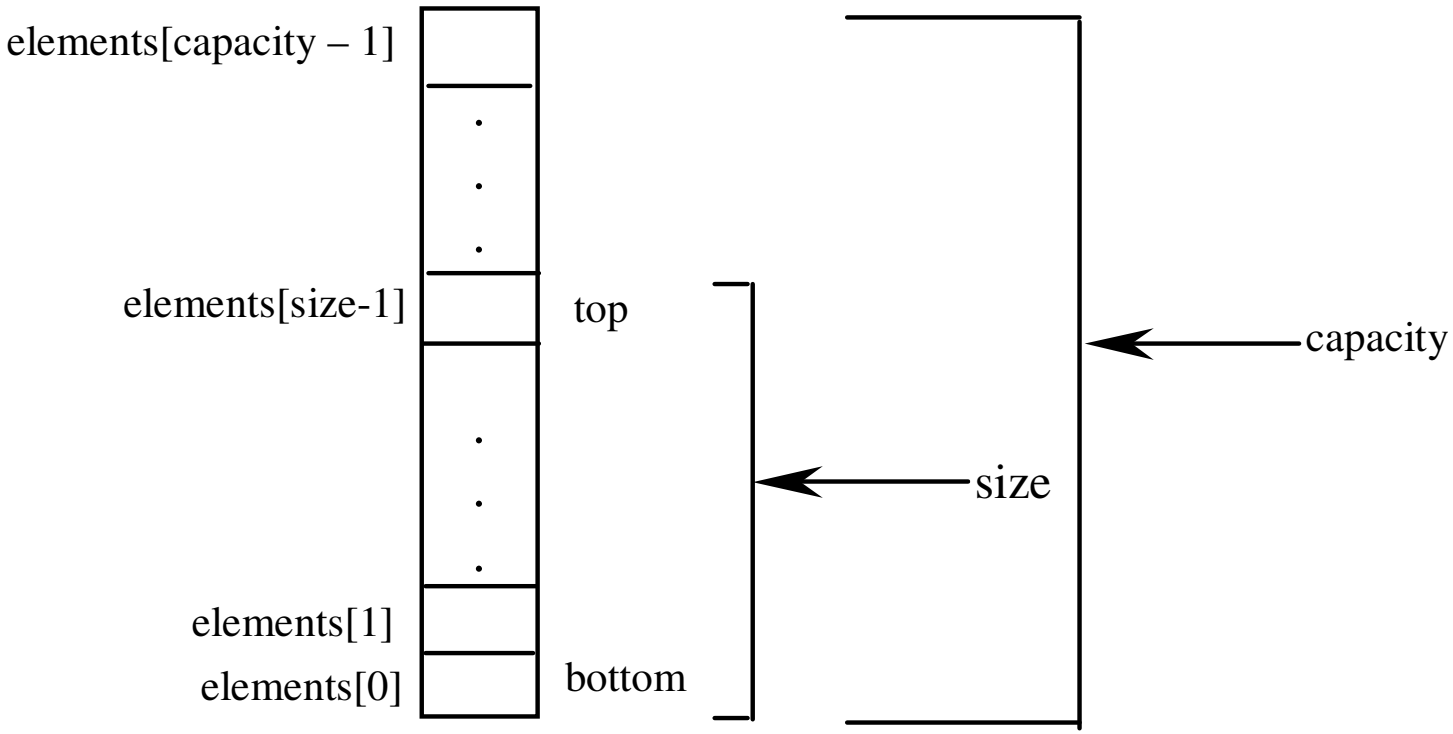
Stores an integer into the top of the stack.

Removes the integer at the top of the stack and returns it.

Returns the number of elements in the stack.

StackOfIntegers.h | StackOfIntegers.cpp | TestStackOfIntegers

# Implementation



# 11.3 Friends of Classes

- Friend function: a function that is not a member of a class, but has access to private members of the class
- A friend function can be a stand-alone function or a member function of another class
- It is declared a friend of a class with the **`friend`** keyword in the function prototype

# Friend Function Declarations

1) Friend function may be a stand-alone function:

```
class aClass
{
  private:
     int x;
     friend void fSet(aClass &c, int a);
};

void fSet(aClass &c, int a)
{
    c.x = a;
}
```

# Friend Function Declarations

2) Friend function may be a member of another class:

```
class aClass
{ private:
    int x;
    friend void OtherClass::fSet
                          (aClass &c, int a);
};
class OtherClass
{ public:
     void fSet(aClass &c, int a)
     { c.x = a; }
};
```

# Friend Class Declaration

3) An entire class can be declared a friend of a class:

```
class aClass
{private:
   int x;
   friend class frClass;
};
class frClass
{public:
   void fSet(aClass &c,int a){c.x = a;}
   int fGet(aClass c){return c.x;}
};
```

# Designing a Class: Cohesion

- A class should describe a single entity or a set of similar operations. A single entity with too many responsibilities can be broken into several classes to separate responsibilities.

# Designing a Class: Consistency

- Follow standard programming style and naming conventions. Choose informative names for classes, data fields, and functions. A popular style in C++ is to place the data declaration after the functions, and place constructors before functions.

# Designing a Class: Encapsulation

- A class should use the private modifier to hide its data from direct access by clients. This makes the class easy to maintain.
- Provide a get function only if you want the field to be readable, and provide a set function only if you want the field to be updateable. A class should also hide functions not intended for client use. Such functions should be defined as private.

# Designing a Class: Clarity

- Users can incorporate classes in many different combinations, orders, and environments. Therefore, you should design a class that imposes no restrictions on what or when the user can do with it, design the properties in a way that lets the user set them in any order and with any combination of values, and design functions independently of their order of occurrence. For example, the Loan class contains the functions setLoanAmount, setNumberOfYears, and setAnnualInterestRate. The values of these properties can be set in any order.

# Designing a Class: Completeness

- Classes are designed for use by many different customers. In order to be useful in a wide range of applications, a class should provide a variety of ways for customization through properties and functions. For example, the string class contains more than 20 functions that are useful for a variety of applications.

# Designing a Class: Instance vs. Static

- A variable or function that is dependent on a specific instance of the class should be an instance variable or function. A variable that is shared by all the instances of a class should be declared static. For example, the variable numberOfObjects in Circle, is shared by all the objects of the Circle class, and therefore is declared static. A function that is not dependent on a specific instance should be declared as a static function. For instance, the getNumberOfObjects function in Circle is not tied to any specific instance, and therefore is declared as static functions.