

Chapter 10 Object-Oriented Thinking



Objectives

- ❑ To process strings using the **string** class (§10.2).
- ❑ To develop functions with object arguments (§10.3).
- ❑ To store and process objects in arrays (§10.4).
- ❑ To distinguish between instance and static variables and functions (§10.5).
- ❑ To define constant functions to prevent data fields from being modified accidentally (§10.6).
- ❑ To explore the differences between the procedural paradigm and object-oriented paradigm (§10.7).
- ❑ To design a class for body mass index (§10.7).
- ❑ To discover the relationships between classes (§10.8).
- ❑ To design a class for a stack (§10.9).
- ❑ To initialize objects using constructor initializer lists (§10.10).
- ❑ To design classes that follow the class-design guidelines (§10.11).



The C++ string Class



Constructing a String

You can create an empty string using string's no-arg constructor like this one:

```
string newString;
```

You can create a string object from a string value or from an array of characters. To create a string from a string literal, use a syntax like this one:

```
string newString(stringLiteral);
```

Same as `string newString = stringLiteral;`



Appending a String

string
+append(s: string): string
+append(s: string, index: int, n: int): string
+append(s: string, n: int): string
+append(n: int, ch: char): string

Appends string s into this string object.

Appends n number of characters in s starting at the position index to this string.

Appends the first n number of characters in s to this string.

Appends n copies of character ch to this string.



Appending a String Examples

You can use several overloaded functions to add new contents to a string. For example, see the following code:

```
string s1("Welcome");  
s1.append(" to C++"); // appends " to C++" to s1  
cout << s1 << endl; // s1 now becomes Welcome to C++  
string s2("Welcome");  
s2.append(" to C and C++", 0, 5); // appends " to C" to s2  
cout << s2 << endl; // s2 now becomes Welcome to C  
string s3("Welcome");  
s3.append(" to C and C++", 5); // appends " to C" to s3  
cout << s3 << endl; // s3 now becomes Welcome to C  
string s4("Welcome");  
s4.append(4, 'G'); // appends "GGGG" to s4  
cout << s4 << endl; // s4 now becomes WelcomeGGGG
```



Assigning a String

string
+assign(s[:]: char): string
+assign(s: string): string
+assign(s: string, index: int, n: int): string
+assign(s: string, n: int): string
+assign(n: int, ch: char): string

Assigns array of characters or a string s to this string.

Assigns string s to this string.

Assigns n number of characters in s starting at the position index to this string.

Assigns the first n number of characters in s to this string.

Assigns n copies of character ch to this string.



Assigning a String Examples

You can use several overloaded functions to assign new contents to a string. For example, see the following code:

```
string s1("Welcome");  
s1.assign("Dallas"); // assigns "Dallas" to s1  
cout << s1 << endl; // s1 now becomes Dallas  
string s2("Welcome");  
s2.assign("Dallas, Texas", 0, 5); // assigns "Dalla" to s2  
cout << s2 << endl; // s2 now becomes Dalla  
string s3("Welcome");  
s3.assign("Dallas, Texas", 5); // assigns "Dalla" to s3  
cout << s3 << endl; // s3 now becomes Dalla  
string s4("Welcome");  
s4.assign(4, 'G'); // assigns "GGGG" to s4  
cout << s4 << endl; // s4 now becomes GGGG
```



Functions at, clear, erase, and empty

string
+at(index: int): char
+clear(): void
+erase(index: int, n: int): string
+empty(): bool
+front(): char
+back(): char

Returns the character at the position index from this string.

Removes all characters in this string.

Removes n characters from this string starting at position index.

Returns true if this string is empty.

Returns the first character in the string.

Returns the last character in the string.

C++11: front() and back()
are defined in C++11



Functions at, clear, erase, and empty

Examples

You can use the at(index) function to retrieve a character at a specified index, clear() to clear the string, erase(index, n) to delete part of the string, and empty() to test if a string is empty. For example, see the following code:

```
string s1("Welcome");  
cout << s1.at(3) << endl; // s1.at(3) returns c  
cout << s1.erase(2, 3) << endl; // s1 is now Weme  
s1.clear(); // s1 is now empty  
cout << s1.empty() << endl; // s1.empty returns 1 (means  
true)
```



Functions length, size, capacity, and c_str()

string
+length(): int
+size(): int
+capacity(): int
+c_str(): char[]
+data(): char[]

Returns the number of characters in this string.

Same as length().

Returns the size of the storage allocated for this string.

Returns a C-string for this string.

Same as c_str().



Functions length, size, capacity, and c_str() Examples

```
string s1("Welcome");  
cout << s1.length() << endl; // Length is 7  
cout << s1.size() << endl; // Size is 7  
cout << s1.capacity() << endl; // Capacity is 7  
s1.erase(1, 2);  
cout << s1.length() << endl; // Length is now 5  
cout << s1.size() << endl; // Size is now 5  
cout << s1.capacity() << endl; // Capacity is still 7
```



Comparing Strings

string
+compare(s: string): int
+compare(index: int, n: int, s: string): int

Returns a value greater than 0, 0, or less than 0 if this string is greater than, equal to, or less than s.

Compares this string with s(index, index+1, ..., index + n -1).



Comparing Strings

Often, in a program, you need to compare the contents of two strings. You can use the compare function. This function works in the same way as the C-string strcmp function and returns a value greater than 0, 0, or less than 0. For example, see the following code:

```
string s1("Welcome");  
string s2("Welcomg");  
cout << s1.compare(s2) << endl; // returns -2  
cout << s2.compare(s1) << endl; // returns 2  
cout << s1.compare("Welcome") << endl; // returns 0
```



Obtaining Substrings

string
+substr(index: int, n: int): string
+substr(index: int): string

Returns a substring of n characters from this string starting at position index.

Returns a substring of this string starting at position index.



Obtaining Substrings

You can obtain a single character from a string using the at function. You can also obtain a substring from a string using the substr function. For example, see the following code:

```
string s1("Welcome");  
cout << s1.substr(0, 2) << endl; // returns We  
cout << s1.substr(3) << endl; // returns come  
cout << s1.substr(3, 3) << endl; // returns com
```



Find Substrings

string
+find(ch: char): unsigned
+find(ch: char, index: int): unsigned
+find(s: string): unsigned
+find(s: string, index: int): unsigned

Returns the position of the first matching character for ch.

Returns the position of the first matching character for ch at or from the position index.

Returns the position of the first matching substring s.

Returns the position of the first matching substring s starting at or from the position index.

Note: The find function returns `string::npos`, if no matching is found. `string::npos` is a constant defined in the string class.



Searching in a String

You can use the find function to search for a substring or a character in a string. For example, see the following code:

```
string s1("Welcome to HTML");  
cout << s1.find("co") << endl; // returns 3  
cout << s1.find("co", 6); // returns string::npos  
cout << s1.find('o') << endl; // returns 4  
cout << s1.find('o', 6) << endl; // returns 9
```



Inserting and Replacing Strings

String	
<code>+insert(index: int, s: string): string</code>	Inserts the string <code>s</code> into this string at position <code>index</code> .
<code>+insert(index: int, n: int, ch: char): string</code>	Inserts the character <code>ch</code> <code>n</code> times into this string at position <code>index</code> .
<code>+replace(index: int, n: int, s: string): string</code>	Replaces the <code>n</code> characters starting at position <code>index</code> in this string with the string <code>s</code> .



Inserting and Replacing Strings

Here are the examples to use the insert and replace functions:

```
string s1("Welcome to HTML");  
s1.insert(11, "C++ and ");  
cout << s1 << endl; // s1 becomes Welcome to C++ and HTML  
string s2("AA");  
s2.insert(1, 4, 'B');  
cout << s2 << endl; // s2 becomes to ABBBBBA  
string s3("Welcome to HTML");  
s3.replace(11, 4, "C++");  
cout << s3 << endl; // returns Welcome to C++
```



String Operators

Operator	Description
[]	Accesses characters using the array subscript operators.
=	Copies the contents of one string to the other.
+	Concatenates two strings into a new string.
+=	Appends the contents of one string to the other.
<<	Inserts a string to a stream
>>	Extracts characters from a stream to a string delimited by a whitespace or the null terminator character.
==, !=, <, <=, >, >=	Six relational operators for comparing strings.



Converting Numbers to Strings

You can also use the **itoa** function to convert an integer to a string. Sometimes you need to convert a floating-point number to a string. You can write a function to perform the conversion. However, a simple approach is to use the **stringstream** class in the **<sstream>** header.

```
1 stringstream ss;  
2 ss << 3.1415;  
3 string s = ss.str();
```



Splitting Strings

Often you need to extract the words from a string. Assume that the words are separated by whitespaces. You can use the **stringstream** class discussed in the preceding section to accomplish this task.

```
string text("Programming is fun");  
stringstream ss(text);  
cout << "The words in the text are " <<  
endl;  
string word;  
while (!ss.eof())  
{  
    ss >> word;  
    cout << word << endl;  
}
```



Replacing Strings

ReplaceString

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

// Replace oldSubStr in s with newSubStr
bool replaceString(string& s, const string& oldSubStr,
    const string& newSubStr);

int main()
{
    // Prompt the user to enter s, oldSubStr, and newSubStr
    cout << "Enter string s, oldSubStr, and newSubStr: ";
    string s, oldSubStr, newSubStr;
    cin >> s >> oldSubStr >> newSubStr;

    bool isReplaced = replaceString(s, oldSubStr, newSubStr);

    if (isReplaced)
        cout << "The replaced string is " << s << endl;
    else
        cout << "No matches" << endl;

    return 0;
}

bool replaceString(string& s, const string& oldSubStr,
    const string& newSubStr)
{
    bool isReplaced = false;
    int currentPosition = 0;
    while (currentPosition < s.length())
    {
        int position = s.find(oldSubStr, currentPosition);
        if (position == string::npos) // No more matches
            return isReplaced;
        else
        {
            s.replace(position, oldSubStr.length(), newSubStr);
            currentPosition = position + newSubStr.length();
            isReplaced = true; // At least one match
        }
    }

    return isReplaced;
}
```



Reading Strings

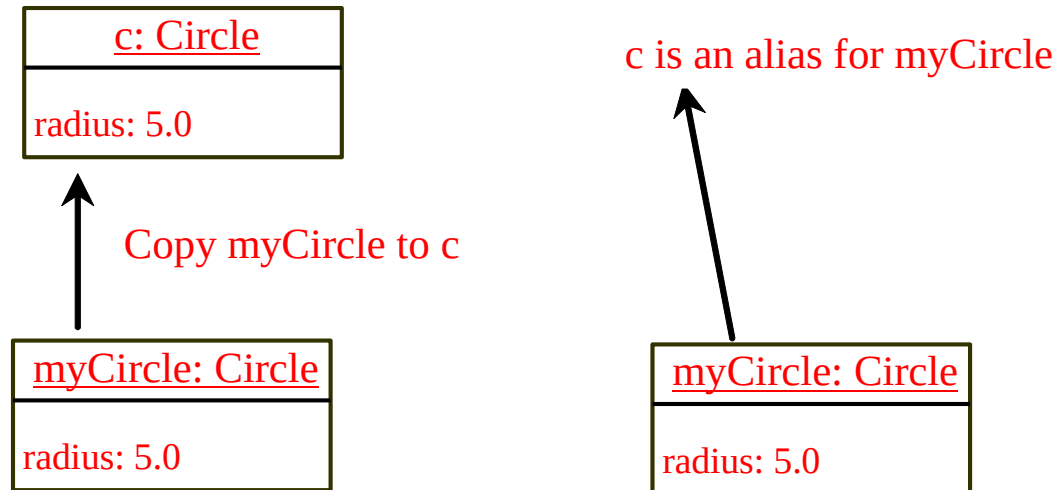
```
string city;  
cout << "Enter a city: ";  
cin >> city; // Read to array city  
cout << "You entered " << city << endl;
```

```
string city;  
cout << "Enter a city: ";  
getline(cin, city, '\n'); // Same as getline(cin, city)  
cout << "You entered " << city << endl;
```



Passing Objects to Functions

You can pass objects by value or by reference.



PassObjectByValue

PassObjectByReference



```

#include <iostream>
// CircleWithPrivateDataFields.h is defined in Listing 9.9
#include "CircleWithPrivateDataFields.h"
using namespace std;

void printCircle(Circle c)
{
    cout << "The area of the circle of "
         << c.getRadius() << " is " << c.getArea() << endl;
}

int main()
{
    Circle myCircle(5.0);
    printCircle(myCircle);

    return 0;
}

```

```

#include <iostream>
#include "CircleWithPrivateDataFields.h"
using namespace std;

void printCircle(Circle& c)
{
    cout << "The area of the circle of "
         << c.getRadius() << " is " << c.getArea() << endl;
}

int main()
{
    Circle myCircle(5.0);
    printCircle(myCircle);

    return 0;
}

```

Array of Objects

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

TotalArea

```
#include <iostream>
#include <iomanip>
#include "CircleWithPrivateDataFields.h"
using namespace std;

// Add circle areas
double sum(Circle circleArray[], int size)
{
    // Initialize sum
    double sum = 0;

    // Add areas to sum
    for (int i = 0; i < size; i++)
        sum += circleArray[i].getArea();

    return sum;
}

// Print an array of circles and their total area
void printCircleArray(Circle circleArray[], int size)
{
    cout << setw(35) << left << "Radius" << setw(8) << "Area" << endl;
    for (int i = 0; i < size; i++)
    {
        cout << setw(35) << left << circleArray[i].getRadius()
            << setw(8) << circleArray[i].getArea() << endl;
    }

    cout << "-----" << endl;

    // Compute and display the result
    cout << setw(35) << left << "The total area of circles is"
        << setw(8) << sum(circleArray, size) << endl;
}
```

```
int main()
{
    const int SIZE = 10;

    // Create a Circle object with radius 1
    Circle circleArray[SIZE];

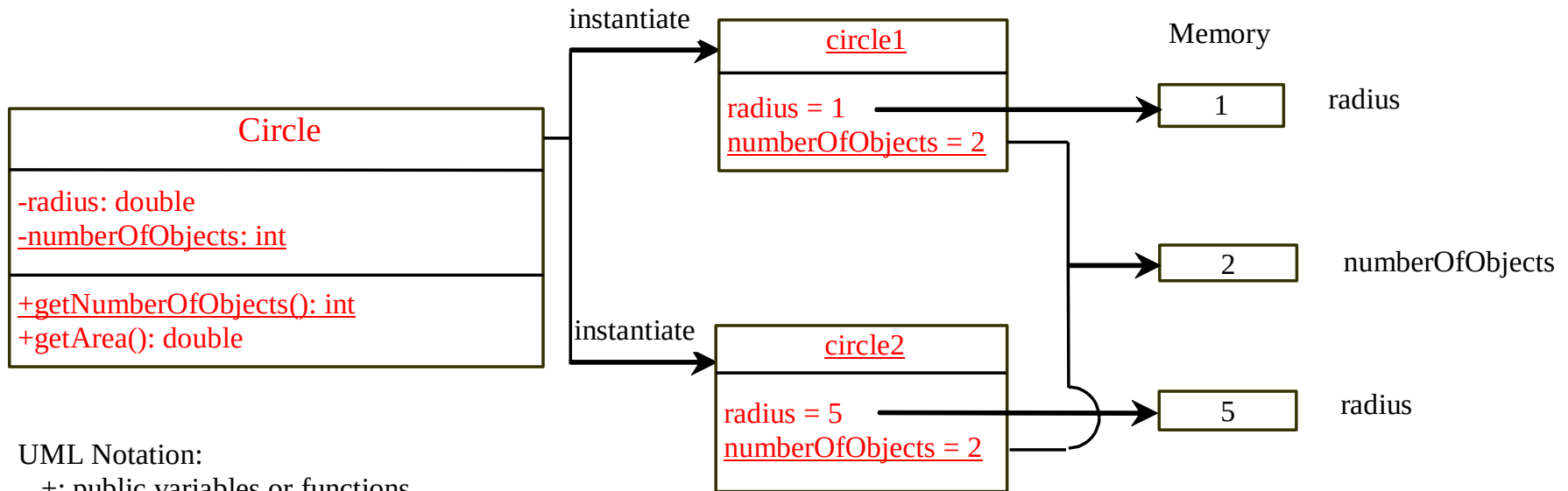
    for (int i = 0; i < SIZE; i++)
    {
        circleArray[i].setRadius(i + 1);
    }

    printCircleArray(circleArray, SIZE);

    return 0;
}
```



Instance and Static Members



UML Notation:

- `+`: public variables or functions
- `-`: private variables or functions
- underline: static variables or functions

Interactive Figure

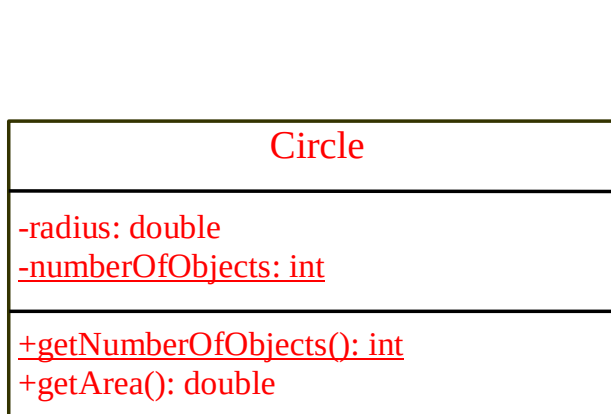
CircleWithStaticDataFields.h

CircleWithStaticDataFields.cpp

TestCircleWithStaticDataFields.cpp



Instance and Static

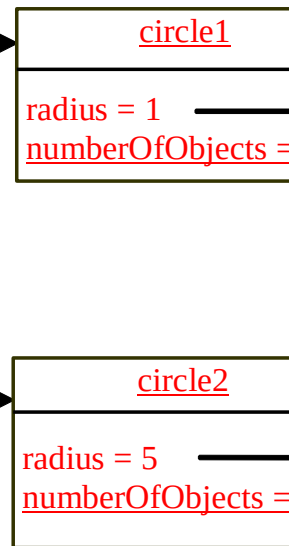


UML Notation:

- `+`: public variables or functions
- `-`: private variables or functions
- underline: static variables or functions

instantiate

instantiate



```
#include "CircleWithStaticDataFields.h"
```

```
int Circle::numberOfObjects = 0;
```

```
// Construct a circle object
Circle::Circle()
{
```

```
    radius = 1;
    numberOfObjects++;
}
```

```
// Construct a circle object
Circle::Circle(double newRadius)
{
    radius = newRadius;
    numberOfObjects++;
}
```

```
// Return the area of this circle
double Circle::getArea()
{
```

```
    return radius * radius * 3.14159;
}
```

```
// Return the radius of this circle
double Circle::getRadius()
{
```

```
    return radius;
}
```

```
// Set a new radius
```

```
void Circle::setRadius(double newRadius)
{
    radius = (newRadius >= 0) ? newRadius : 0;
}
```

```
// Return the number of circle objects
```

```
int Circle::getNumberOfObjects()
{
    return numberOfObjects;
}
```

Interactive Figure

CircleWithStaticData

CircleWithStaticDataF

TestCircleWithStaticData

Instance

```
#include <iostream>
#include "CircleWithStaticDataFields.h"
using namespace std;
```

```
int main()
{
    cout << "Number of circle objects created: "
        << Circle::getNumberOfObjects() << endl;

    Circle circle1;
    cout << "The area of the circle of radius "
        << circle1.getRadius() << " is " << circle1.getArea() << endl;
    cout << "Number of circle objects created: "
        << Circle::getNumberOfObjects() << endl;

    Circle circle2(5.0);
    cout << "The area of the circle of radius "
        << circle2.getRadius() << " is " << circle2.getArea() << endl;
    cout << "Number of circle objects created: "
        << Circle::getNumberOfObjects() << endl;

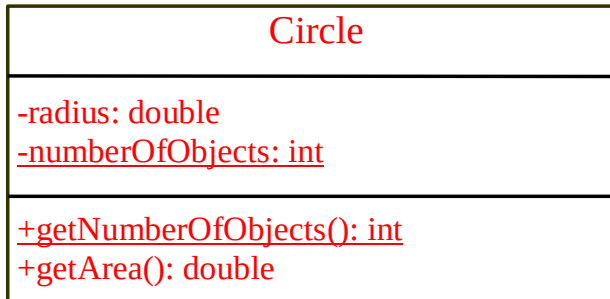
    circle1.setRadius(3.3);
    cout << "The area of the circle of radius "
        << circle1.getRadius() << " is " << circle1.getArea() << endl;

    cout << "circle1.getNumberOfObjects() returns "
        << circle1.getNumberOfObjects() << endl;
    cout << "circle2.getNumberOfObjects() returns "
        << circle2.getNumberOfObjects() << endl;

    return 0;
}
```

ins

in



UML Notation:

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

Interactive Figure

C

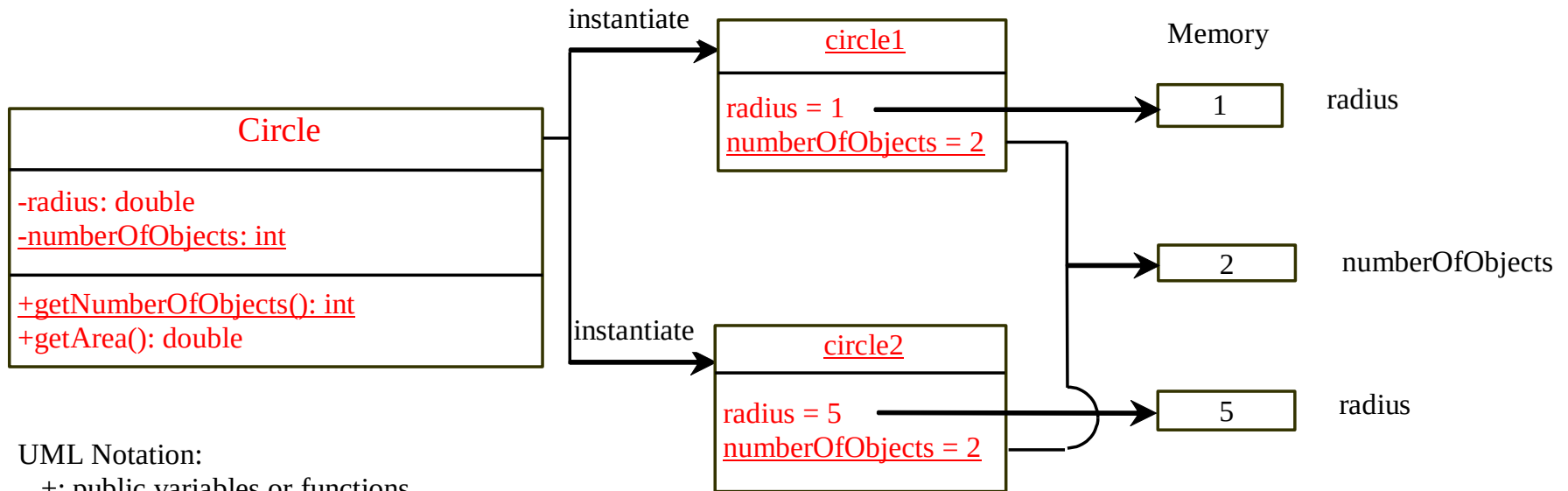
return 0;

CircleWithStaticDataFields.cpp

TestCircleWithStaticDataFields.cpp



Instance and Static Members



UML Notation:

- `+`: public variables or functions
- `-`: private variables or functions
- underline: static variables or functions

Interactive Figure

CircleWithStaticDataFields.h

CircleWithStaticDataFields.cpp

TestCircleWithStaticDataFields.cpp



Use Class Name

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

```
circle1.getNumberOfObjects()
```

```
Circle::getNumberOfObjects() // This is better
```



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.

Instance or Static?

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

class Test
{
public:
    ? int square(int n)
    {
        return n * n;
    }

    ? int getAge()
    {
        return age;
    }

private:
    int age;
};
```



Instance or Static?

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

class Test
{
public:
    static int square(int n)
    {
        return n * n;
    }

    int getAge()
    {
        return age;
    }

private:
    int age;
};
```



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.

CircleWithConstantMemberFunctions.h

CircleWithConstantMemberFunctions.cpp



Constant Member Functions

You can use the const keyword to specify a constant member function to tell the compiler that the value of any data fields in the function should not be changed in the function. You specify a constant member function by placing the const keyword at the end of the function header.

```
#ifndef CIRCLE_H
#define CIRCLE_H

class Circle
{
public:
    Circle();
    Circle(double);
    double getArea() const;
    double getRadius() const;
    void setRadius(double);
    static int getNumberOfObjects();

private:
    double radius;
    static int numberOfObjects;
};

#endif
```

CircleWithConstantMemberFunctions.h

CircleWithConstantMemberFunctions.cpp



```
#include "CircleWithConstantMemberFunctions.h"
```

```
int Circle::numberOfObjects = 0;
```

```
// Construct a circle object
```

```
Circle::Circle()  
{  
    radius = 1;  
    numberOfObjects++;  
}
```

```
// Construct a circle object
```

```
Circle::Circle(double newRadius)  
{  
    radius = newRadius;  
    numberOfObjects++;  
}
```

```
// Return the area of this circle
```

```
double Circle::getArea() const  
{  
    return radius * radius * 3.14159;  
}
```

```
// Return the radius of this circle
```

```
double Circle::getRadius() const  
{  
    return radius;  
}
```

```
// Set a new radius
```

```
void Circle::setRadius(double newRadius)  
{  
    radius = (newRadius >= 0) ? newRadius : 0;  
}
```

```
// Return the number of circle objects
```

```
int Circle::getNumberOfObjects()  
{  
    return numberOfObjects;  
}
```

t Member Functions

const keyword to specify a
er to tell the compiler that the
not be changed in the function.
you to specify a constant member
e compiler that the function
e the value of any data fields in
so, place the const keyword at the
on header.

CircleWithConstantMemberFunctions.h

CircleWithConstantMemberFunctions.cpp



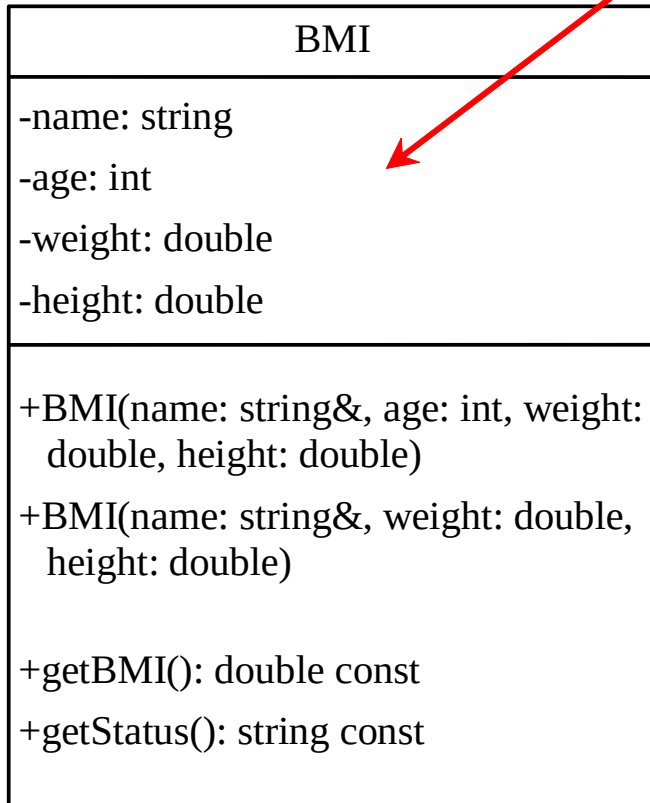
Object-Oriented Thinking

The book 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. This section improves the solution for a problem introduced in Chapter 3 using the object-oriented approach. From the improvements, you will gain insight on the differences between the procedural programming and object-oriented programming and see the benefits of developing reusable code using objects and classes.



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



The BMI Class

```
#ifndef BMI_H
#define BMI_H

#include <string>
using namespace std;

class BMI
{
public:
    BMI(const string& newName, int newAge,
        double newWeight, double newHeight);
    BMI(const string& newName, double newWeight, double newHeight);
    double getBMI() const;
    string getStatus() const;
    string getName() const;
    int getAge() const;
    double getWeight() const;
    double getHeight() const;

private:
    string name;
    int age;
    double weight;
    double height;
};

#endif

Overweight, etc.)
```

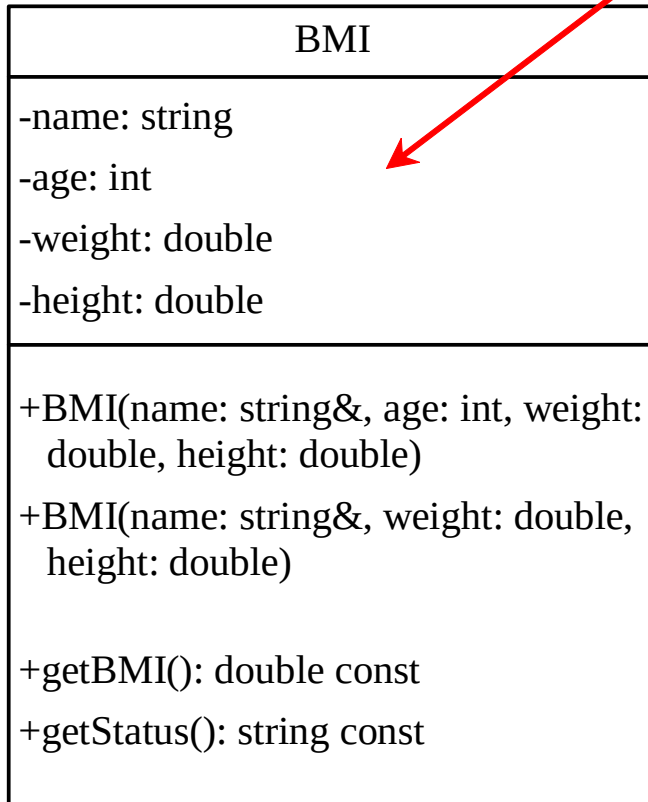
BMI
-name: string -age: int -weight: double -height: double
+BMI(name: string&, age: int, weight: double, height: double) +BMI(name: string&, weight: double, height: double) +getBMI(): double const +getStatus(): string const

BMI.h

BMI.cpp

UseBMIClass

The BMI



The get functions provided in the class UML diagram for

The name of the person

The age of the person

The weight of the person

The height of the person

Creates a BMI object with name, age, weight, and height

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

Returns the BMI.

Returns the BMI status: Underweight, Normal, Overweight, etc.

```
#include <iostream>
#include "BMI.h"
using namespace std;

BMI::BMI(const string& newName, int newAge, double newWeight, double newHeight)
{
    name = newName;
    age = newAge;
    weight = newWeight;
    height = newHeight;
}

BMI::BMI(const string& newName, double newWeight, double newHeight)
{
    name = newName;
    age = 20;
    weight = newWeight;
    height = newHeight;
}

double BMI::getBMI() const
{
    const double KILOGRAMS_PER_POUND = 0.45359237;
    const double METERS_PER_INCH = 0.0254;
    double bmi = weight * KILOGRAMS_PER_POUND / ((height * METERS_PER_INCH) * (height * METERS_PER_INCH));
    return bmi;
}

string BMI::getStatus() const
{
    double bmi = getBMI();
    if (bmi < 18.5)
        return "Underweight";
    else if (bmi < 25)
        return "Normal";
    else if (bmi < 30)
        return "Overweight";
    else
        return "Obese";
}

string BMI::getName() const
{
    return name;
}

int BMI::getAge() const
{
    return age;
}
```

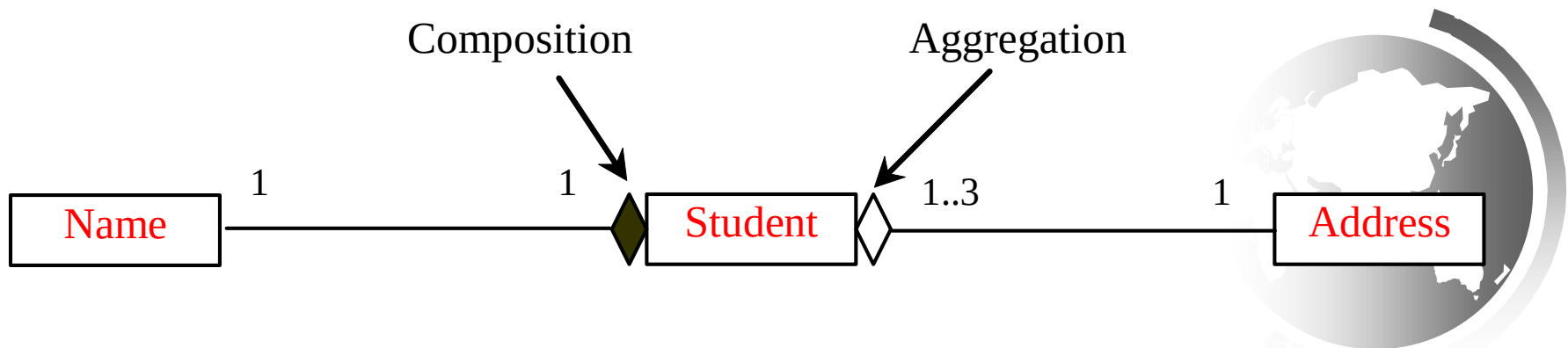
BMI.h

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
{
    ...
}
```

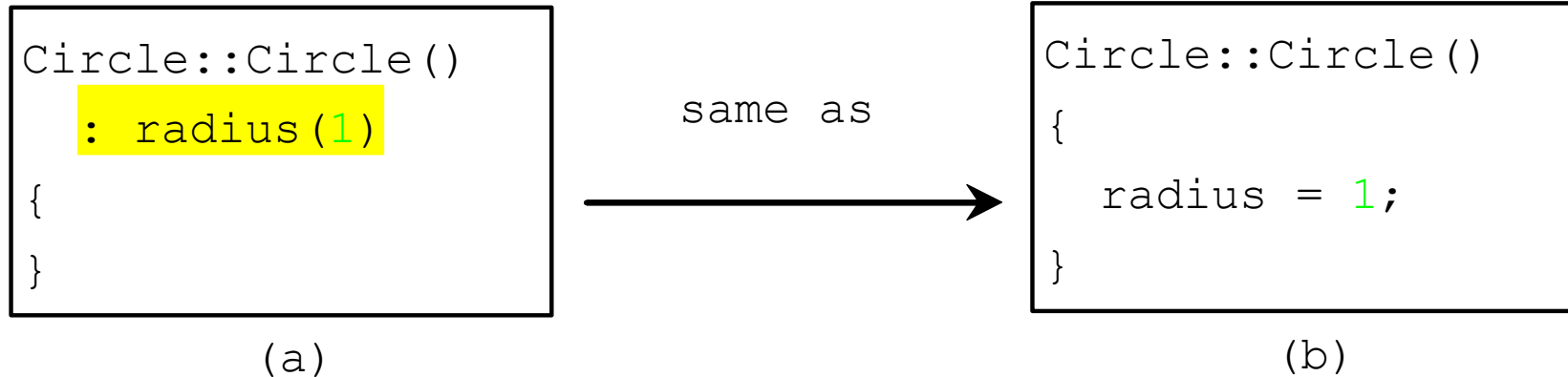
Aggregated class

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.



Constructor_INITIALIZER



You can use either constructor initializer in (a) or the assignment in (b) to initialize data in the constructor. However, using an initializer list is necessary to initialize object data fields that don't have a no-arg constructor.

Object Data Fields Are Not Initialized

```
class Student
{
public:
    Student();

private:
    int age = 5; // OK
    // Wrong, must use a no-arg constructor
    string name("Peter");
};
```

```
class Student
{
public:
    Student();

private:
    int age = 5;
    // Declare a data field of the string type
    string name;
};
```



Use Constructor Initializer

NoArgConstructorNeeded.cpp

UseConstructorInitializer.cpp



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

```
class Time
{
public:
    Time(int newHour, int newMinute, int newSecond)
    {
        hour = newHour;
        minute = newMinute;
        second = newSecond;
    }

    int getHour()
    {
        return hour;
    }

private:
    int hour;
    int minute;
    int second;
};

class Action
{
public:
    Action(const string& newActionName, int hour, int minute, int second)
    {
        actionName = newActionName;
        time = Time(hour, minute, second);
    }

    Time getTime()
    {
        return time;
    }

private:
    string actionName;
    Time time;
};

int main()
{
    Action action("Go to class", 11, 30, 0);
    cout << action.getTime().getHour() << endl;

    return 0;
}
```

ctor_INITIALIZER

NoArgConstructorNeeded.cpp

UseConstructorInitializer.cpp



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

```
class Time
{
public:
    Time(int newHour, int newMinute, int newSecond)
    {
        hour = newHour;
        minute = newMinute;
        second = newSecond;
    }

    int getHour()
    {
        return hour;
    }
};
```

```
private:
    int hour;
    int minute;
    int second;
};
```

```
class Action
{
public:
    Action(const string& newActionName, int hour, int minute, int second)
        :time(hour, minute, second)
    {
        actionName = newActionName;
    }

    Time getTime()
    {
        return time;
    }
};
```

```
private:
    string actionName;
    Time time;
};
```

```
int main()
{
    Action action("Go to class", 11, 30, 0);
    cout << action.getTime().getHour() << endl;

    return 0;
}
```

ctor_INITIALIZER

NoArgConstructorNeeded.cpp

UseConstructorInitializer.cpp



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 in Listing 10.9, 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.

Summary

The focus of this chapter is on object-oriented thinking and design. We also cover the following major topics:

- ❑ The string class
- ❑ The instance vs static members
- ❑ Object-oriented thinking and design
- ❑ Class relationship
- ❑ Class design

