Object-oriented Programming
Classes Part 2

YoungWoon Cha
CSE Department
Spring 2023

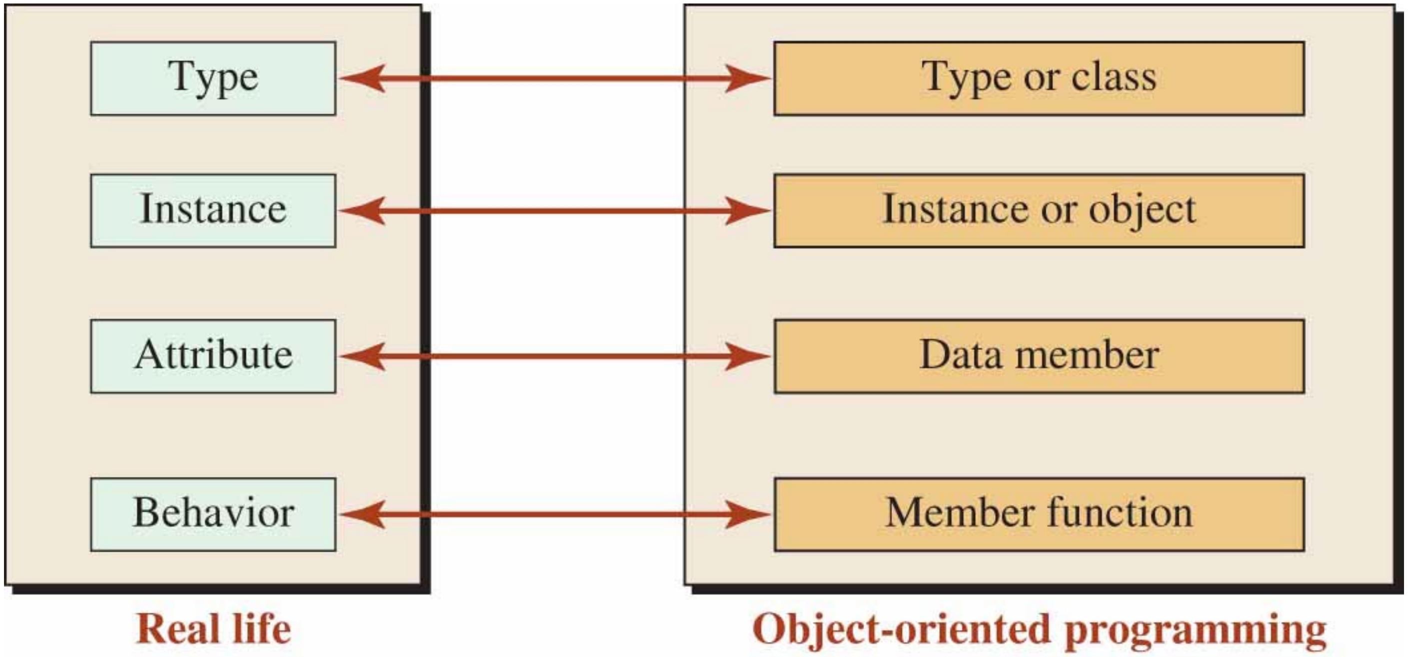
Review

Classes and Objects in Programs

In C++, a user-defined type can be created using a construct named class.

An instance of a class is referred to as an *object*.

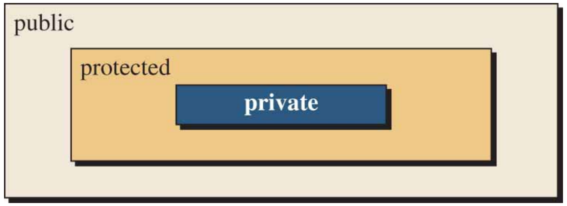
This means that we use *type* and *class*. We also use *instances* and *objects*.



Class Definition

```
class Circle // Header
{
    private:
        double radius;    // Data member declaration
    public:
        double getRadius() const; // Member function declaration
        double getArea() const; // Member function declaration
        double getPerimeter() const; // Member function declaration
        void setRadius(double value);  // Member function declaration
}; // A semicolon is needed at the end of class definition
```

Access Modifiers



Data member of a class are normally set to private.

Member functions of a class are normally set to public.

Member Function Implementation

```
class Circle
{
  ...
   public:
      double getRadius () const;
      ...
}
```

No scope resolution in the function declaration

```
double Circle :: getRadius () const;
{
   ...
}

...
```

Scope resolution in in the function definition

Inline Functions and Application

Implicit Inline Function

```
class Circle
{
      // Data Members
      private:
            double radius;
      // Member functions
      public:
            double getRadius()const {return radius };
      …
};
```

Explicit Inline Function

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

Object Instantiation

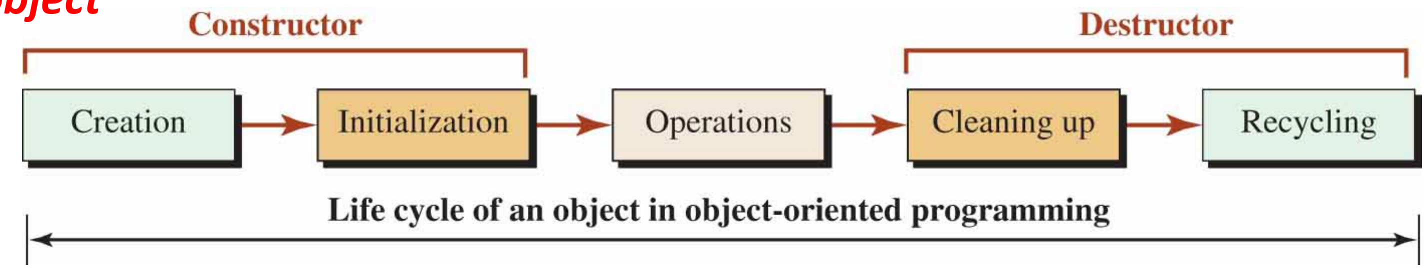
```
Circle circle1;
```

Applying Operation on Objects

```
circle1.setRadius(10.0);
cout << "Radius: " << circle1.getRadius() << endl;
cout << "Area: " << circle1.getArea() << endl;
cout << "Perimeter: " << circle1.getPerimeter() << endl << endl;
```

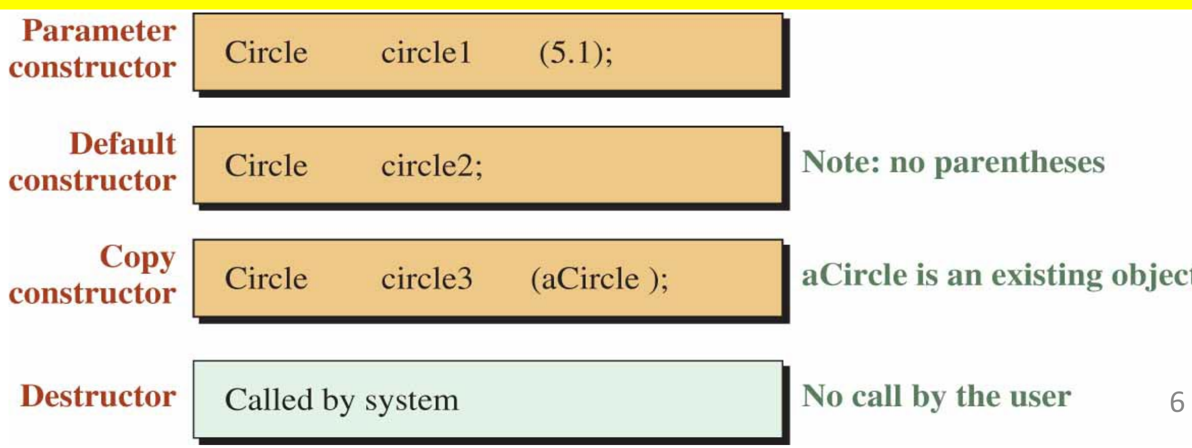
CONSTRUCTOR AND DESTRUCTOR

Life Cycle of an object




```
class Circle
{
      . . .
      public:
            Circle(double radius); // Parameter Constructor
            Circle(); // Default Constructor
            Circle(const Circle& circle); // Copy Constructor
            ~Circle();    // Destructor
      . . .
}
```

Object Creating and destroying for a class type



Classes Part 2

Instance Data Members

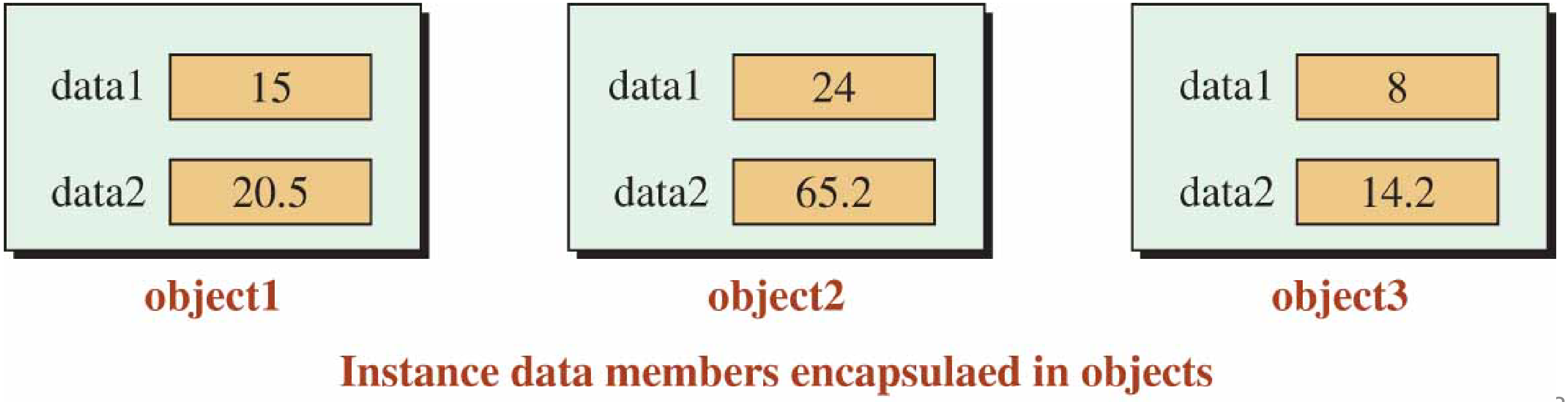
The instance data members of a class are normally private to be accessed only through instance member functions.

An instance data member defines the attributes of an instance, which means that each object needs to *encapsulate* the set of data members defined in the class.

These data members exclusively belong to the corresponding instance and cannot be accessed by other instances.

The term *encapsulation* here means that separate regions of memory are assigned for each object and each region stores possibly different values for each data member.

Figure 7.9 *Encapsulation of data members in objects*

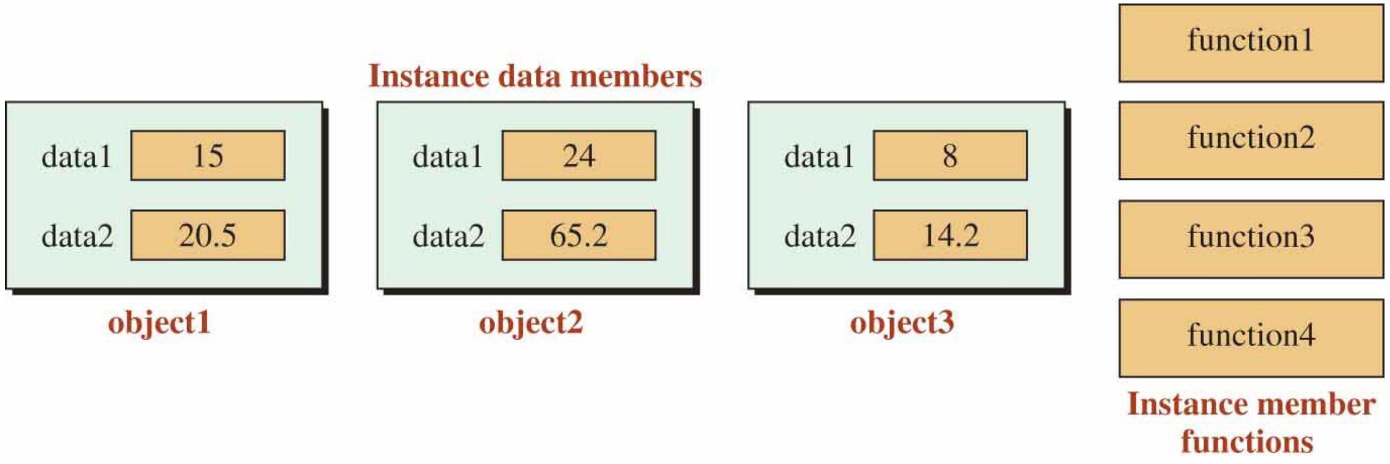


Instance Member Functions

The instance member function of a class needs to be public to be accessed from outside of the class.

Although each object has its own instance data members, there is only one copy of each instance member function in memory and it needs to be shared by all instances.

Unlike instance data members, the access modifier for an instance member function is normally public to allow access from outside the class (the application) unless the instance member function is supposed to be used only by other instance member functions within the class.



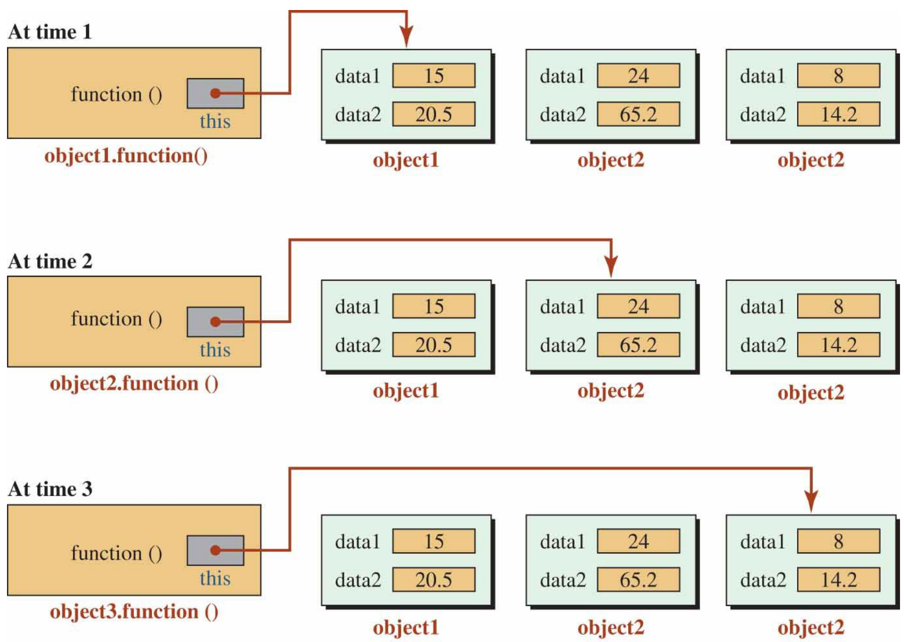
Instance Member Function Selectors

Locking and Unlocking

The question often asked is if there is only one copy of a member function, how can that function be used by one object at one time and by another object at another time.

This is done using a special pointer called the *this* pointer as shown in the next slide. The pointer is pointing to the current object.

Figure 7.12 *Locking and unlocking of a function to an object*



Instance Member Function Selectors

Hidden Parameter

How does an instance member function get a *this* pointer?

It is added as a parameter to the instance member function by the compiler as shown below:

```
// Written by the user
double getRadius() const
{
    return radius;
}
```

```
// Changed by the compiler
double getRadius(Circle* this) const
{
    return (this -> radius);
}
```

The operator (->) is a special operator that is the combination of the indirection operator and the member operator.

`this -> radius` **is the same as** `(*this).radius`

The compiler changes the call statement into two statements as shown below:

```
// Written by the user
circle1.getRadius();
```

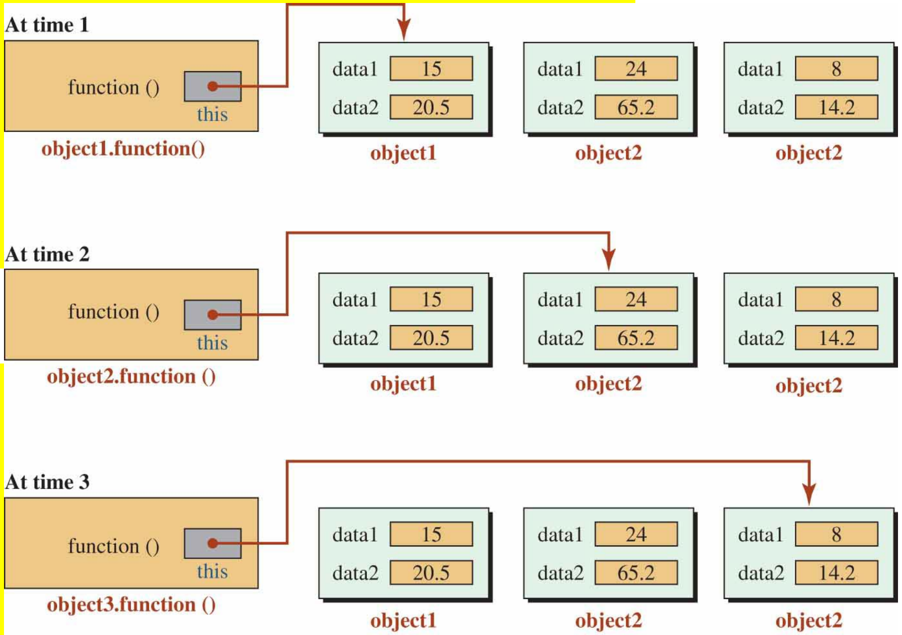
```
//Changed by the system
this = &cirlce1;
getRadius(this);
```

Instance Member Function Selectors

Explicit Use of this Pointer

We can use the *this* pointer in our program to refer to a data member instead of using the data member itself and we can use the name of the data member as a parameter.

```
// Without using this pointer
void Circle :: setRadius(double rds)
{
    radius = rds;
}
```




```
// Using this pointer
void Circle :: setRadius(double radius)
{
    this -> radius = radius;
}
```

Getter and Setter Member Functions

Accessor Member Function

An *accessor* member function (sometimes called a *getter*) gets information from the host object but does not change the state of the object.

```
double getRadius() const;       // Host object is read-only
double getPerimeter() const;    // Host object is read-only
double getArea() const;         // Host object is ready-only
```

An accessor instance function must not change the state of the host object; it needs the const modifier.

```
void Circle :: output() const
{
    cout << "Radius: " << radius << endl;
    cout << "Perimeter: " << 2 * radius * 3.14 << endl;
    cout << "Area: " << radius * radius * 3.14 << endl;
}
```

Getter and Setter Member Functions

Mutator Member Function

We may need some instance member functions that can change the state of their host objects. Such a function is called a *mutator* instance member function (sometimes called a *setter*).

```
void setRadius(double rds);// No const qualifier for a mutator
```

A mutator instance function changes the state of the host object; it cannot have the *const* modifier.

```
void Circle :: input()
{
    cout << "Enter the radius of the circle object: ";
    cin << radius;
}
```

Class Invariants

An *invariant* is one or more conditions that need to be imposed on some or all class data members.

We enforce the *invariant* of a class through instance data member functions that create objects (parameter constructors) or mutator member functions that change the value of a data member.

```
Circle :: Circle(double rds)
: radius (rds)
{
    if (radius <= 0.0))
    {
        cout << "No circle can be made!" << endl;
        cout << "The program is aborted" << endl;
        assert(false);
    }
}
```

Class Invariants Part 1

Program 7.4 *Using the class Rectangle*

```
/***************************************************************
 * A program to declare, define, and use a Rectangle class    *
 ***************************************************************/
#include <iostream>
#include <cassert>
using namespace std;
/***************************************************************
 * Class Definition (Declaration of data members and member    *
 * functions) for a Rectangle class.                           *
 ***************************************************************/
class Rectangle
{
    private:
        double length; // Data member
        double height; // Data member
    public:
        Rectangle (double length, double height); // Constructor
        Rectangle (const Rectangle& rect); // Copy constructor
        ~Rectangle (); // Destructor
        void print () const; // Accessor member
```

Class Invariants Part 2

Program 7.4 *Using the class Rectangle*

```
double getArea () const; // Accessor member
double getPerimeter () const; // Accessor member
};
/*****************************************************************
 * Definitions of constructors, destructor, and the accessor     *
 * instance member functions                                     *
 *****************************************************************/
// Parameter constructor
Rectangle :: Rectangle (double len, double hgt)
: length (len), height (hgt)
{
     if ((length <= 0.0) || (height <= 0.0 ))
     {
         cout << "No rectangle can be made!" << endl;
         assert (false);
     }
}
// Copy constructor
Rectangle :: Rectangle (const Rectangle& rect)
: length (rect.length), height (rect.height)
```

Class Invariants Part 3

Program 7.4 *Using the class Rectangle*

```
{
}
// Destructor
Rectangle :: ~Rectangle ()
{
}
// Accessor member function: Print length and heigth
void Rectangle :: print() const
{
     cout << "A rectangle of " << length << " by " << height << endl;
}
// Accessor member function: Get area
double Rectangle :: getArea () const
{
     return (length * height);
}
// Accessor member function: Get perimeter
double Rectangle :: getPerimeter () const
{
     return (2 * (length + height));
```

Class Invariants Part 4

Program 7.4 *Using the class Rectangle*

```
}
/*****************************************************************
 * Application to instantiate three objects and use them         *
 *****************************************************************/
int main ()
{
     // Instantiation of three objects
     Rectangle rect1 (3.0, 4.2); // Using parameter constructor
     Rectangle rect2 (5.1, 10.2); // Using parameter constructor
     Rectangle rect3 (rect2); // Using copy constructor
     // Operations on first rectangle
     cout << "Rectangle 1: ";
     rect1.print();
     cout << "Area: " << rect1.getArea() << endl;
     cout << "Perimeter: " << rect1.getPerimeter() << endl << endl;
     // Operations on second rectangle
     cout << "Rectangle 2: ";
     rect2.print();
     cout << "Area: " << rect2.getArea() << endl;
     cout << "Perimeter: " << rect2.getPerimeter() << endl << endl;
```

Class Invariants Part 5

Program 7.4 *Using the class Rectangle*

```
// Operations on third rectangle
     cout << "Rectangle 3: ";
     rect3.print();
     cout << "Area: " << rect3.getArea() << endl;
     cout << "Perimeter: " << rect3.getPerimeter() << endl << endl;
     return 0;
}
```

```
Run
Rectangle 1: A rectangle of 3 by 4.2
Area: 12.6
Perimeter: 14.4

Rectangle 2: A rectangle of 5.1 by 10.2
Area: 52.02
Perimeter: 30.6

Rectangle 3: A rectangle of 5.1 by 10.2
Area: 52.02
Perimeter: 30.6
```

In-class Exercise I

In-class Exercise I

- **Modify the previous example with the following code and answer the following questions.**

(a) `Rectangle rect1(-3.0, 4.2);`

- What happened?
- What caused the problem?
- What's the solution?

(b) `Rectangle rect0;`

- What happened?
- What caused the problem?
- What's the solution?

Static Members

STATIC MEMBERS

A class type can have two types of members: *instance members* and *static members*.

We discussed instance members in the previous section; we discuss static members in this section.

As with the case of instance members, we can have *static data members* and *static member functions*.

Static Data Members

A *static data member* is a data member that belongs to all instances; it also belongs to the class itself.

Declaration of a Static Data Member

Data members belong to the class and their declarations must be included in the class definition; static members need to be qualified with the keyword *static*.

The following shows how we declare a static data member named *count* inside the class definition

```
class Rectangle
{
    private:
        …
        static int count;   // Static data member
    public:
        …
}
```

Static Data Members

Initialization Of Static Data Members

A static data member belongs to no instance, which means it cannot be initialized in a constructor.

A static data member must be initialized after the class definition.

This means it must be initialized in a global area of the program.

We need to show that it belongs to the class by adding the class name and the class scope operator (::) to the definition, but the static qualifier should not be added.

```
int Rectangle :: count = 0; // initialization of static data member
```

Static Member Function Part 1

Since a static data member is normally private, we need a public member function to access it.
Although it can be accessed by an instance member function, normally we use a static member function for this purpose.

A static member function has no host object.

Declaration of a Static Member Function

Member function belong to the class and their declarations must be included in the class definition; they need to be qualified with the keyword *static*.
The following shows how we declare a static member function named *count* inside the class definition.

```
class Rectangle
{
    private:
    …
        static int count;  // Static data member
    public:
        static int getCount(); // Static member function
    …
}
```

Static Member Function Part 2

Definition of Static Member Functions

A static member function needs to be defined outside of the class, like an instance member function.

```
int Rectangle :: getCount()
{
    return count;
}
```

Calling Static Member Functions

A static member function can be called either through an instance or through the class.

```
rect.getCount();             // Through an instance
Rectangle :: getCount();     // Through the class
```

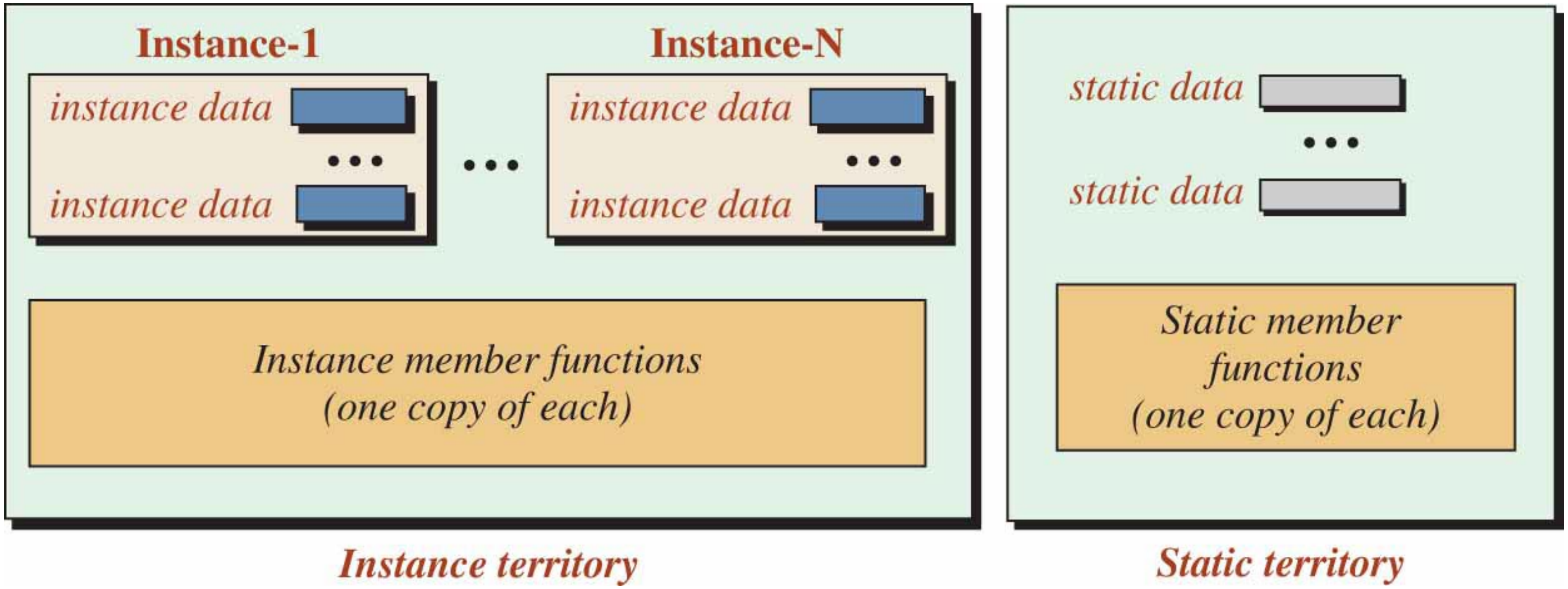
A static member function cannot be used to access instance data members because it has no *this* pointer parameter.

Static Member Function Part 3

An instance member function can be used to access static data members (the *this* pointer is not used), but we usually avoid this.

A good practice is to use instance member functions to access instance data members and static member functions to access static data members.

Figure 7.13 *Separation of instance and static territory*



In-class Exercise II

In-class Exercise II Part 1

- Implement the following Rectangle Class so that you can get the same result in the output.

```
/*************************************************************
 * A program to create objects and count them.              *
 *************************************************************/
#include <iostream>
using namespace std;
/*************************************************************
 * Definitions of the class Rectangle                       *
 *************************************************************/
class Rectangle
{
    private:
        double length;
        double height;
        static int count; //Static data member
    public:
        Rectangle (double length, double height);
        Rectangle ();
        ~Rectangle ();
        Rectangle (const Rectangle& rect);
        static int getCount (); // Static member function
```

In-class Exercise II Part 2

- Implement the following Rectangle Class so that you can get the same result in the output.

```
};
```

?

In-class Exercise II Part 3

- Implement the following Rectangle Class so that you can get the same result in the output.

```
?
```

In-class Exercise II Part 4

- Implement the following Rectangle Class so that you can get the same result in the output.

```
int main ( )
{
     {
          Rectangle rect1 (3.2, 1.2);
          Rectangle rect2 (1.5, 2.1);
          Rectangle rect3;
          Rectangle rect4 (rect1);
          Rectangle rect5 (rect2);
          cout << "Count of objects: " << rect5.getCount() << endl;
     }
     cout << "Count of objects: " << Rectangle :: getCount();
     return 0;
}
```

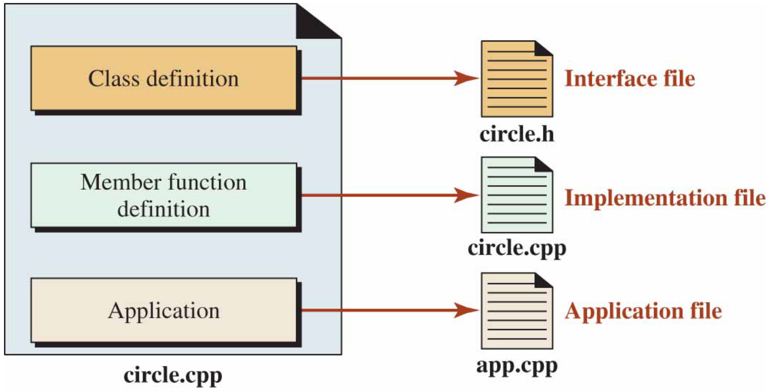
```
Run:
Count of objects: 5
Count of objects: 0
```

Separate Files

Separate Files

Interface File

Figure 7.14 *Three files created in C++ for a class*



The interface file is a file that contains the class definition The name of this file is normally the name of the class with an *h* extension, such as *circle.h*. The letter *h* designates it as a header file.

Implementation File

The implementation file contains the definition of member functions. The name of this file is normally the name of the class with a *cpp* extension, such as *circle.cpp*, although the extension may vary in different C++ environments.

Application File

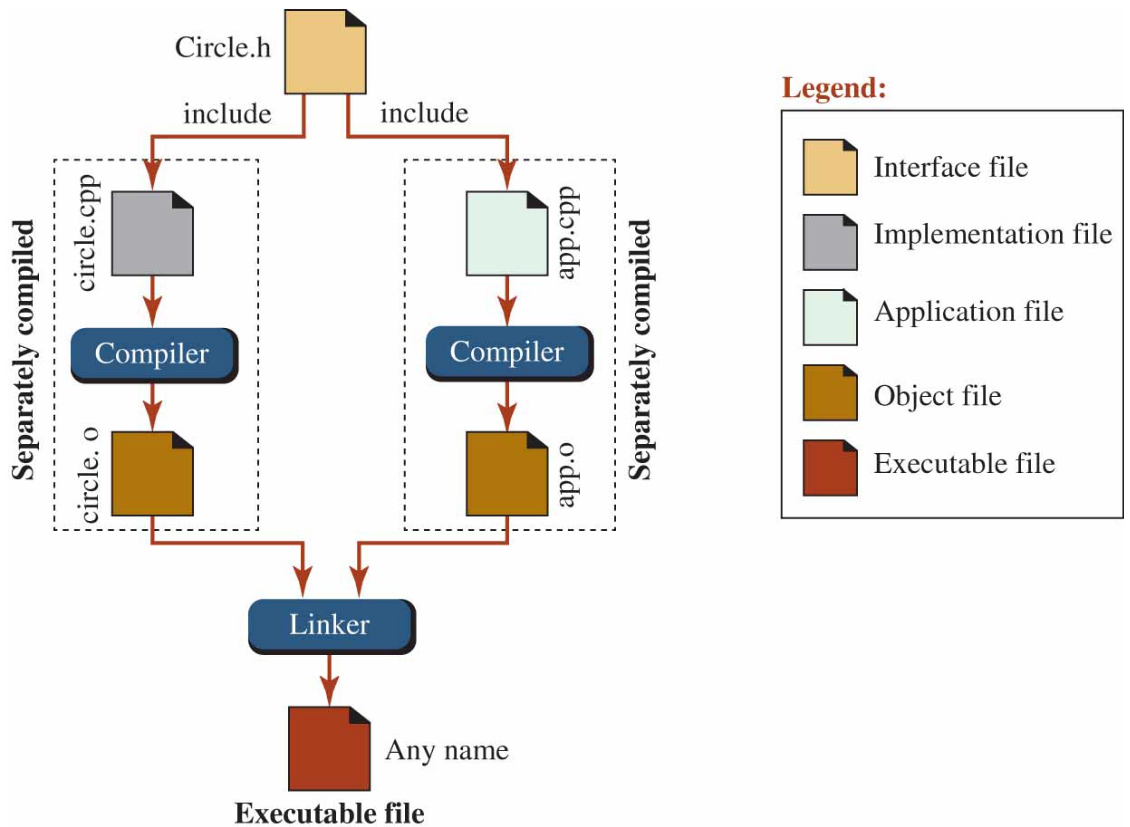
The application file includes the *main* function that is used to instantiate objects and let each object perform operations on themselves.
The application file needs also to have the extension *cpp*, but the name of the file is usually chosen by the user.
The name we use is *app.cpp* although the extension may vary in different C++ environments.

Separate Compilation Part 1

After creating three separate files, we need to compile them to create an executable file.

In C++, the process is referred to as *separate compilation*.

Figure 7.15 ***Process of separate compilation***



Separate Compilation Part 2

Step-by-Step Process

a. The *interface file* is created containing only the class definition. This file needs to be included in the *implementation file* and the *application file*.

b. The *implementation* file is created with the interface file included using an *include directive*. The option -c indicates that we want only compilation.

```
c++ -c circle.cpp
```

c. The application file is created in which the interface file is also added to the beginning of the file using the include directive.

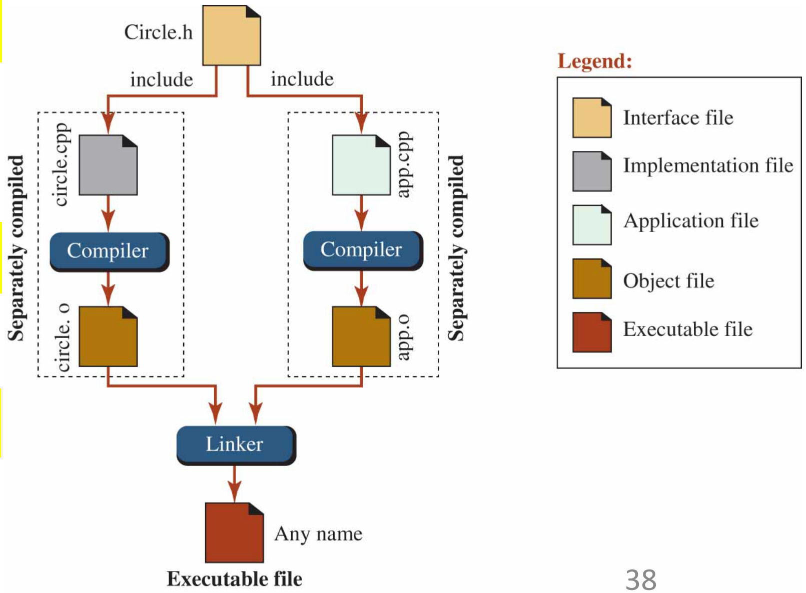
```
c++ -c app.cpp
```

d. We link the two object files together with the -o option to create an executable file.

```
c++ -o application circle.o app.o
```

e. The result is an executable file.

```
c++ application
```



Interface File Part 1

Program 7.7 *The interface file*

```
/***********************************************************************
 * This is the interface file that defines the class Circle.    *
 * It gives declaration of data members and member functions.   *
 * This file will be included at the top of the implementation *
 * and application files.                                       *
 ***********************************************************************/
#ifndef CIRCLE_H
#define CIRCLE_H
#include <iostream>
#include <cassert>
#include "circle.h"
using namespace std;
// Class Definition
class Circle
{
     private:
          double radius;
     public:
          Circle (double radius); // Parameter constructor
          Circle (); // Default constructor
```

Interface File Part 2

Program 7.7 *The interface file*

```
    Circle (const Circle& circle); // Copy constructor
    ~Circle (); // Destructor
    void setRadius (double radius); // Mutator function
    double getRadius () const; // Accessor function
    double getArea () const; // Accessor function
    double getPerimeter () const; // Accessor function
};
#endif
```

Interface File Part 3

Program 7.8 *The implementation file*

```
/**************************************************************
 * This is the implementation file that defines the definition *
 * of all member functions. A copy of the interface file is    *
 * included at the top to allow compilation of this file.      *
 **************************************************************/
# include "circle.h"
/****************************************************************
 * The parameter constructor with one argument that initializes  *
 * a circle with the given value. It uses the assert function to *
 * validate that the radius is a positive double value. If not,  *
 * the program is aborted.                                       *
 ****************************************************************/
Circle :: Circle (double rds)
: radius (rds)
{
     if (radius < 0.0)
     {
         assert (false);
     }
}
```

Interface File Part 4

Program 7.8 *The implementation file*

```
/*****************************************************************
 * The default constructor that initializes a circle set to 0.0. *
 * It does not need an assertion.                                *
 *****************************************************************/
Circle :: Circle ()
: radius (0.0)
{
}
/*****************************************************************
 * The copy constructor that copies the radius of another circle *
 * to create a new one. The source circle is already validated,  *
 * which means that we do not need validation.                   *
 *****************************************************************/
Circle :: Circle (const Circle& circle)
: radius (circle.radius)
{
}
/*****************************************************************
 * A destructor that cleans up an object when the application is *
 * terminated.                                                   *
```

Interface File Part 5

Program 7.8 *The implementation file*

```
 ****************************************************************/
Circle :: ~Circle ()
{
}
/****************************************************************
 * The setRadius function is defined to change the circle       *
 * by decreasing or increasing the size of the radius. It needs *
 * validation because the new size of must be a positive value  *
 ****************************************************************/
void Circle :: setRadius (double value)
{
     radius = value;
     if (radius < 0.0)
     {
         assert (false);
     }
}
/***************************************************************
 * The getRadius is a function that returns the radius         *
 * of an object. It needs the const modifier to prevent the    *
```

Interface File Part 6

Program 7.8 *The implementation file*

```
 * accidental change of the host object.                    *
 ************************************************************/
double Circle :: getRadius () const
{
     return radius;
}
/**************************************************************
 * The getArea accessor function returns the area of the host    *
 * object. It needs the const modifier to prevent the accidental *
 * change of the host object.                                    *
 **************************************************************/
double Circle :: getArea () const
{
     const double PI = 3.14;
     return (PI * radius * radius);
}
/************************************************************
 * The getPerimeter accessor function returns the perimeter of *
 * the host object. It needs the const modifier to prevent the *
 * accidental change of the host object.                       *
```

Interface File Part 7

Program 7.8 *The implementation file*

```
 ******************************************************************/
double Circle :: getPerimeter () const
{
    const double PI = 3.14;
    return (2 * PI * radius);
}
```

Interface File Part 7

Program 7.9 *The application file*

```
/********************************************************************
 * This is the application file that instantiates objects and     *
 * lets the object operate on themselves using member functions. *
 * To be to compiled, it needs a copy of the interface file       *
/********************************************************************
# include "circle.h"
int main ( )
{
    // Instantiation of first object and applying operations
    Circle circle1 (5.2);
    cout << "Radius: " << circle1.getRadius() << endl;
    cout << "Area: " << circle1.getArea() << endl;
    cout << "Perimeter: " << circle1.getPerimeter() << endl;
    cout << endl;
    // Instantiation of second object and applying operations
    Circle circle2 (circle1);
    cout << "Radius: " << circle2.getRadius() << endl;
    cout << "Area: " << circle2.getArea() << endl;
    cout << "Perimeter: " << circle2.getPerimeter() << endl;
    cout << endl;
```

Interface File Part 8

Program 7.9 *The application file*

```
    // Instantiation of third object and applying operations
    Circle circle3;
    cout << "Radius: " << circle3.getRadius() << endl;
    cout << "Area: " << circle3.getArea() << endl;
    cout << "Perimeter: " << circle3.getPerimeter() << endl;
    cout << endl;
    return 0;
}
```

Interface File Output

Program 7.9 *The application file*

```
c++ -c circle.cpp  // Compilation of implementation file
c++ -c app.cpp     // Compilation of application file
c++ -o application circle.o app.o  // Linking of two compiled files
application         // Running the executable fil
```

```
Run:
Radius: 5.2
Area: 84.9056
Perimeter: 32.656

Radius: 5.2
Area: 84.9056
Perimeter: 32.656

Radius: 0
Area: 0
Perimeter: 0
```

Preventing Multiple Inclusion Part 1

If we include the contents of the same header file more than once in a compilation file, the compiler issues an error and the compilation is aborted. To prevent this, we use the following *preprocessor directives*: *define*, *ifndef* (if not defined), and *endif*.

Figure 7.16 *Contents of header file with three directives*

```
#ifndef   CIRCLE_H
#define   CIRCLE_H

     // contents of the circle.h file

 #endif
```

circle.h

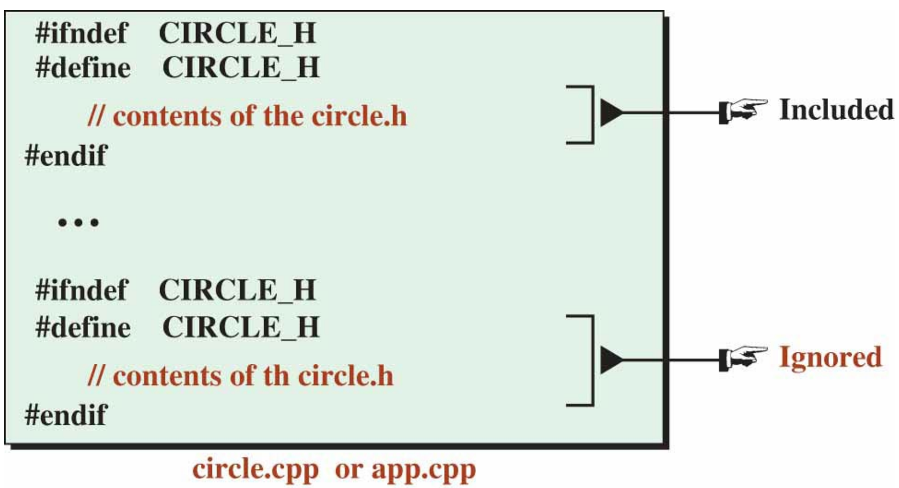
These three directives work with a flag (a constant). The flag we have used in the figure is the name of the file with an underscore followed by the letter H, all in uppercase. This is a convention; any name can be used if consistent.

Preventing Multiple Inclusion Part 2

When the preprocessor encounters the first *ifndef* directive, since the flag is not defined yet, it defines it (next line) and adds the rest of code until it encounter the *endif* directive, which means that the contents of the header file are added to the source file.

When the preprocessor encounters the second *ifndef* directive, since the flag is already defined, it immediately jumps to the *endif* directive and does not include the contents of the header file again.

Figure 7.17 ***How conditional directives ignore duplicate inclusion***



Encapsulation Part 1

Why do we need separate compilations. The reason is that it allows us to achieve one of the goals of object-oriented programming, encapsulation.

Design of the Class

The designer creates the *interface file* and the *implementation file*. The designer makes the *interface file* public.

The *implementation file* is compiled, but only the compiled version is made public; the source code remains private.

The designer can change the implementation file at any time, re-compile it, and re-announce it.

Use of the Class

The user receives a copy of the interface file and the compiled version of the implementation file. The user adds the interface to her application file and compiles it. She then links her own compiled file and the compiled file received from the designer to create an executable file.

Encapsulation Part 2

Effect

The effective result is that the designer protects both the interface file and the implementation file from any changes by the user as shown below:

- **The interface file is protected from change because there are two copies of it used in the process. The designer uses one copy and the user uses another copy. If the user changes the copy she received publicly, the separate-compilation process does not work.**
- **The implementation file is protected from change because the designer just sends the compiled version of the file to the user. Compilation is a one-way process. The user cannot get the original file from the compiled file to change it.**

Encapsulation Part 3

The public interface is a text file based on the functions declarations that tells the user of the class how to use it.

Table 7.5 ***The public Interface for the Circle class***

Constructors and Destructor
Circle :: Circle()
A default constructor to build a circle with length = 0.0 and height = 0.0.
Circle :: Circle(double radius)
A parameter constructor to build a circle with the given radius.
Circle :: Circle(const Circle& circle)
A copy constructor to build a circle the same as an existing circle.
Circle :: ~Circle() .
The destructor to cleanup the circle object that goes out of scope.
Accessor Functions
Circle :: double getRadius() const
An accessor function that returns the radius of the host object.
Circle :: double getArea() const
An accessor function that returns the area of the host object.
Circle:: double getPerimeter() const
An accessor function that returns the perimeter of the host object.
Mutator Functions
Circle:: void setRadius(double radius)
A mutator function that changes the radius of the host object.

In-class Exercise III

In-class Exercise III

- **Modify the “Interface file code” with “#pragma once”.**

“*#pragma once*” is a preprocessor directive used in C++ to avoid the multiple inclusion of header files (Windows only).

Using “#pragma once” instead of traditional include guards (*#ifndef, #define, and #endif*) is often preferred as it is more concise and can improve compilation time. However, #pragma once is not part of the C++ standard, and some compilers may not support it (Linux). In such cases, using traditional include guards is a reliable alternative.

- **Modify the “Interface file code” to avoid “#include” in .h files.**

When *a header file* contains “*#include*” directives, and that header file is included in multiple source files, the contents of the included files are duplicated in each source file, leading to code bloat and longer compile times. This can also cause naming conflicts, redefinition errors, and other issues, especially if the included files define classes or functions.

To avoid these issues, it is recommended to include only the necessary declarations (such as class declarations, function prototypes, constants, etc.) in header files, and avoid including any implementation details or other headers that are not required. Instead, the implementation details and necessary headers should be included in the corresponding source (.cpp) files.

What’s Next?

Reading Assignment

- **Read Chap. 8. Arrays**
- **Read Chap. 9. References, pointers, and Memory Management**
- **Read Chap. 10. Strings**

End of Class

Thank you

E-mail: youngcha@konkuk.ac.kr