Object-oriented Programming

YoungWoon Cha
CSE Department
Spring 2023

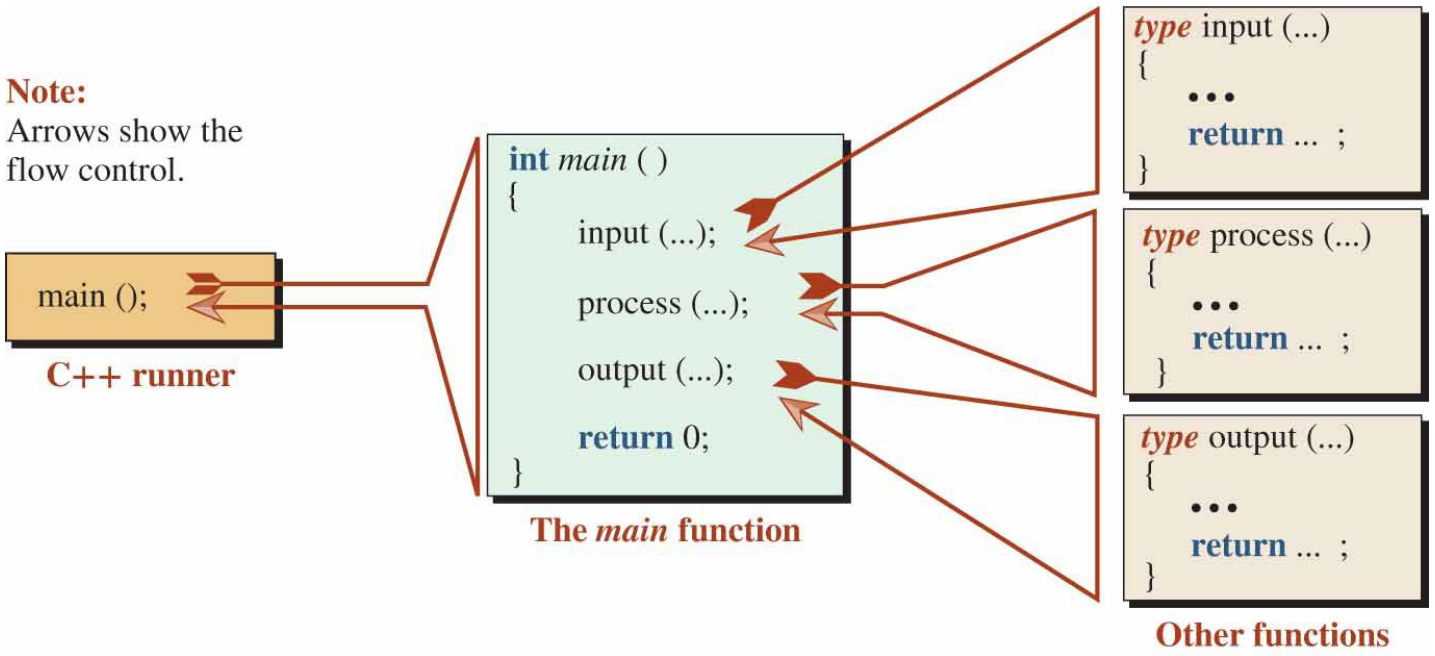
KU KONKUK UNIVERSITY

Review

Functions

A function is an entity designed to do a task; it the benefit of dividing a task into several small tasks.

Figure 6.1 *A program made of several functions*



Library and User-Defined Functions

To be able to use functions in a program, we need to have a *function definition* and a *function call*.

However, there are two cases: library functions and user-defined functions.

Library Functions

There are predefined functions in the C++ library.

We need only the declaration of the functions to be able to call them.

We study some of these functions in the next sections.

User-Defined Functions

There are a lot of functions that we need, but there are no predefined functions for them.

We need to define these functions first and then call them. We learn how to do that later in the chapter.

Passing Data

We can use three mechanisms for passing data from an argument to a parameter: *pass-by-value*, *pass-by-reference*, and *pass-by-pointer*.

Data can be returned in three ways: *return-by-value*, *return-by-reference*, and *return-by-pointer*.

return-by-pointer is seldom used.

Function Overloading

Can we have two functions with the same name?

The answer is positive if their parameter lists are different (in type, in number, or in order).

In C++, the practice is called overloading.

The criteria the compiler uses to allow two functions with the same name in a program is referred to as the function *signature*.

```
int max(int a, int b)
{
    ...
}
```

```
double max(double a, double b)
{
    ...
}
```

Scope

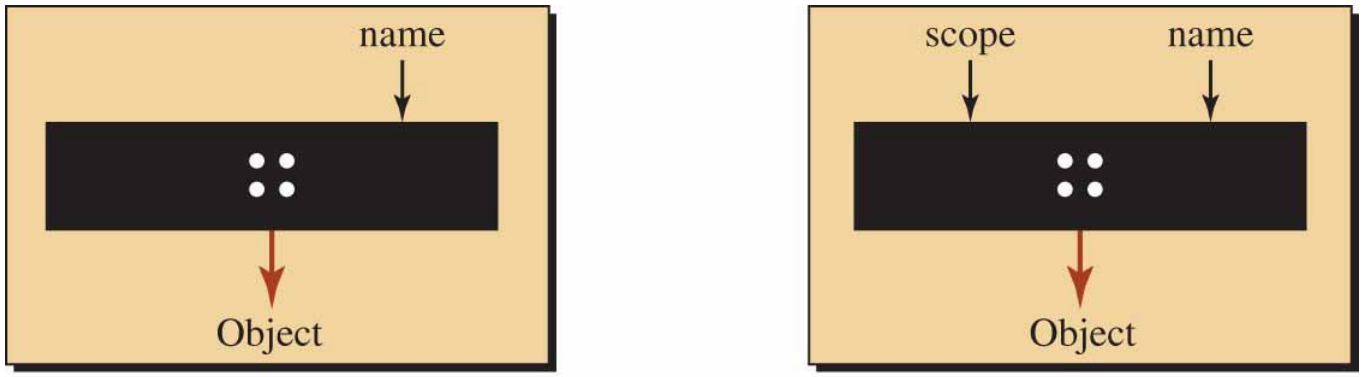
Local Scope, Overlapped scope, Global scope of variables.

- static variables and initialization.

C++ provides an operator (::) that can explicitly or implicitly define the scope of the entity.

Scope of function names and parameters.

Figure 6.25 *Two versions of scope resolution operator*



Version with one operand

Version with two operands

KU KONKUK UNIVERSITY

Classes Part 1

Objectives

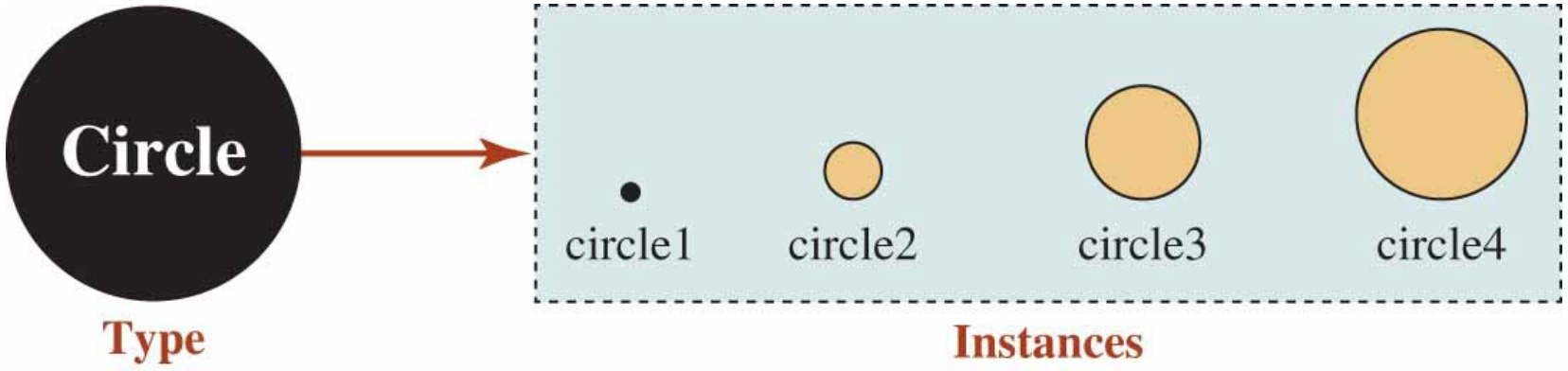
- Classes to introduce object-oriented concepts
  - ✓ Definition, data members, member functions
  - ✓ Constructors, Destructors
- To discuss how we divide the three sections of a program into three separate files
  - ✓ interface file, implementation file, and application file.
  - ✓ encapsulation.
- Object-oriented approach
  - ✓ C++ is a combination of a procedural and an object-oriented language.
  - ✓ In the previous classes, we mostly used as a procedural language.
  - ✓ Start using it as an object-oriented language.

Types And Instances in Real Life

A type is a concept from which instances are created. In other words, a type is an abstraction; an instance of that type is a concrete entity.

The relationship between a type and its instances is a one-to-many relation. We can have many instances from one single type.

Figure 7.1 *A type and its instances*

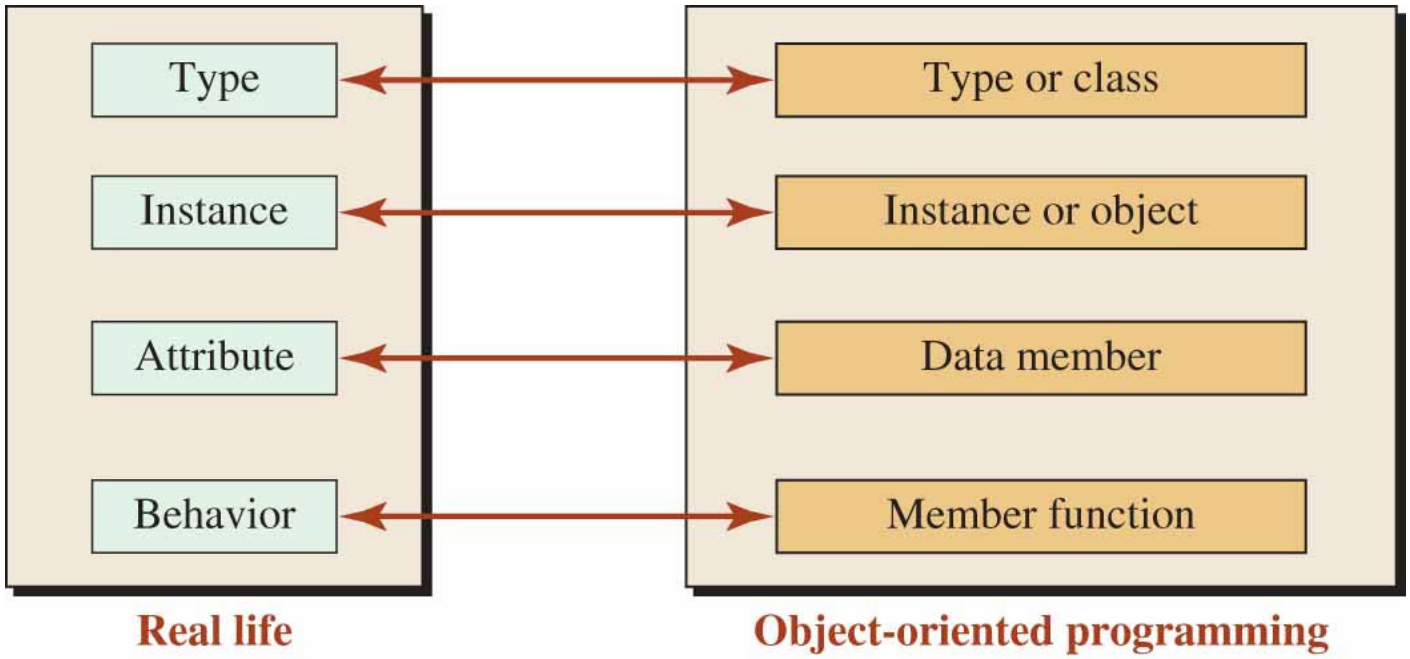


Classes and Objects in Programs

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*.



Data Members and Member Functions

Data Members

A data member of an object is a variable whose value represents an attribute.

Attributes of an object in object-oriented programming are simulated using data members.

Member Functions

When we think about instances, we also think about their behaviors.

In this sense, a behavior is a operation that an instance can perform on itself.

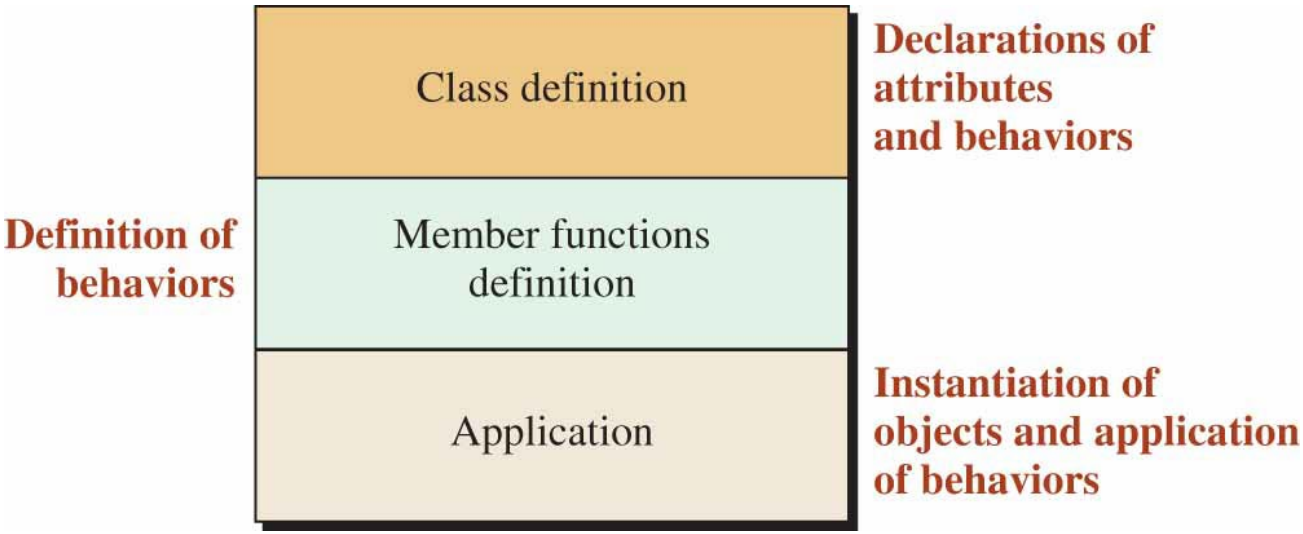
Behaviors of an object in object-oriented programming are simulated using member functions.

CLASSES

In C++, new types are mostly created using a *class*.

To be able to write object-oriented programs, we need three sections:

Figure 7.3 ***Three sections of a C++ program***



Class Definition

A class definition is made of three parts:

header
body
semicolon

A class header is made of the reserved word *class* followed by the name given by the designer.

The class body is a block that holds the declaration of data members and member functions in the block.

The third element of a class declaration is a semicolon that terminates the 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
```

Class Definition

Declaration of Data Members

One part of the class definition declares data members of the class, variables or constants of built-in types or other previously defined class types.

The data members of a class actually simulate the attributes of the objects that are instantiated from the class.

```
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
```

Class Definition

Declaration of Member Functions

The second part of the class definition declares the member functions of the class; that is it declares all functions that are used to simulate the behavior of the class.

This section is similar to the prototype declaration we used when we were working with global functions in the programs we wrote previously.

```
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
```

Class Definition

Access Modifier

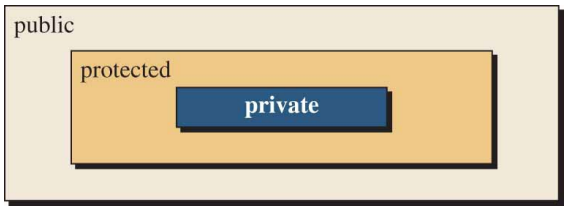
The declaration of data members and member functions in a class are by default *private*.

This means that they cannot be accessed for retrieving or changing.

When there is no access modifier for a member, it is private by default.

```
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 Modifier



Modifier	Access from same class	Access from subclass	Access from anywhere
private	Yes	No	No
protected	Yes	Yes	No
public	Yes	Yes	Yes

Access Modifier for Data Members

The modifiers for data members are normally set to *private* for emphasis. This means that the data members are not accessible directly. They must be accessed through the member functions.

Data member of a class are normally set to private.

Access Modifier for Member Functions

To operate on the data members, the application needs to use member functions, which means that the declaration of member functions usually needs to be set to *public*.

Member functions of a class are normally set to public.

Member Functions Definition

The definition of each member function is similar to the definition of regular function, but with two differences.

The first is the qualifier (*const*) that is applied to some member function. (these functions ensure not to modify the object)

The second is the name of the function that needs to be qualified with the name of the class.

In C++, we need to mention the class name first followed by a class scope (::) symbol to achieve this goal.

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

No scope resolution in the function declaration

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

•••
```

Scope resolution in in the function definition

Member Functions Definition Part 1

The declaration of a member function just gives its prototype; each member function also needs definition.

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

```
void Circle :: setRadius(double value)
{
    radius = value;
}
```

Creation and Handling Two Circle Objects Part 1

Program 7.1 *Creating and handling two circle objects*

```
1  /*************************************************************
2   * A program to use a class in object-oriented programming     *
3   *************************************************************/
4  #include <iostream>
5  using namespace std;
6
7  /*************************************************************
8   * Class definition: the declaration of data members and member *
9   * functions of the class                                      *
10  *************************************************************/
11 class Circle
12 {
13      private:
14          double radius;
15      public:
16          double getRadius () const;
17          double getArea () const;
18          double getPerimeter () const;
19          void setRadius (double value);
20 };
```

Creation and Handling Two Circle Objects Part 2

Program 7.1 *Creating and handling two circle objects*

```
21  /*************************************************************
22   * Members function definition. Each function declared in the  *
23   * class definition section is defined in this section.        *
24   *************************************************************/
25  // Definition of getRadius member function
26  double Circle :: getRadius () const
27  {
28       return radius;
29  }
30  // Definition of getArea member function
31  double Circle :: getArea () const
32  {
33       const double PI = 3.14;
34       return (PI * radius * radius);
35  }
36  // Definition of getPerimeter member function
37  double Circle :: getPerimeter () const
38  {
39       const double PI = 3.14;
40       return (2 * PI * radius);
```

Creation and Handling Two Circle Objects Part 3

Program 7.1 *Creating and handling two circle objects*

```
41  }
42  // Definition of setRadius member function
43  void Circle :: setRadius (double value)
44  {
45       radius = value;
46  }
47  /*************************************************************
48   * Application section: Objects are instantiated in this section. *
49   * Object use member functions to get or set their attributes.    *
50   *************************************************************/
51  int main ( )
52  {
53       // Creating first circle and applying member functions
54       cout << "Circle 1: " << endl;
55       Circle circle1;
56       circle1.setRadius (10.0);
57       cout << "Radius: " << circle1.getRadius() << endl;
58       cout << "Area: " << circle1.getArea() << endl;
59       cout << "Perimeter: " << circle1.getPerimeter() << endl << endl;
60       // Creating second circle and applying member functions
```

Creation and Handling Two Circle Objects Part 4

Program 7.1 *Creating and handling two circle objects*

```
61      cout << "Circle 2: " << endl;
62      Circle circle2;
63      circle2.setRadius (20.0);
64      cout << "Radius: " << circle2.getRadius() << endl;
65      cout << "Area: " << circle2.getArea() << endl;
66      cout << "Perimeter: " << circle2.getPerimeter();
67      return 0;
68  }
```

```
Run:
Circle 1:
Radius: 10
Area: 314
Perimeter: 62.8
Circle 2:
Radius: 20
Area: 1256
Perimeter: 125.6
```

Inline Functions

A function can be designated as *inline* to indicate that the compiler can replace the function call with the actual code in the function.

Implicit Inline Function

A function can be considered implicitly *inline* if we replace its declaration (in the class definition) with its definition.

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

Explicit Inline Function

We can also designate a function to be inline by adding the keyword *inline* in front of the function definition.In this case, the definition remained unchanged with the exception of adding the *inline* keyword.

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

Application

We need to have an application section, the *main* function for example, to instantiate objects of the class and apply the member functions on the those objects.

Object Instantiation

Before using any member function, we need to instantiate an object of the class as shown below:

```
Circle circle1;
```

Applying Operation on Objects

After instantiation, we can let the object to apply one or more operations defined in the member function definition on itself.

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

Member Selection

We are using a dot between the object name and the member function that is suppose to operate on the object. This is called the member select operator that we discuss later in the chapter. In other words, we can apply the same function on different objects using this operator as shown below:

```
circle1.getRadius();      // circle1 is supposed to get its radius
circle2.getRadius();      // circle2 is supposed to get its radius
```

Struct

A *struct* in the C++ language is actually a class with one difference: in a *struct*, all members are public by default; in a class all members are private by default.

We can always create a class to simulate a *struct* as shown below:

```
struct
{

    string first;
    char middle;
    string last;
};
```

```
class
{
    public:
        string first;
        char middle;
        string last;
};
```

KU KONKUK UNIVERSITY

Constructors and Destructors

CONSTRUCTOR AND DESTRUCTOR

If we want an object to perform some operations on itself, we should first create the object and initialize its data members.

Creation is done when a special member function named a *constructor* is called in the application; initialization is when the body of a constructor is executed.

On the other hand, when we do not need an object anymore, the object should be cleaned up and the memory occupied by the object should be recycled.

Cleanup is automatically done when another special member function named a *destructor* is called when the object goes out of scope and the body of the destructor is executed; recycling is the done when the program is terminated.

A constructor is a special member function that creates and initializes an object.
A destructor is a special member function that cleans and destroys an object.

CONSTRUCTOR AND DESTRUCTOR

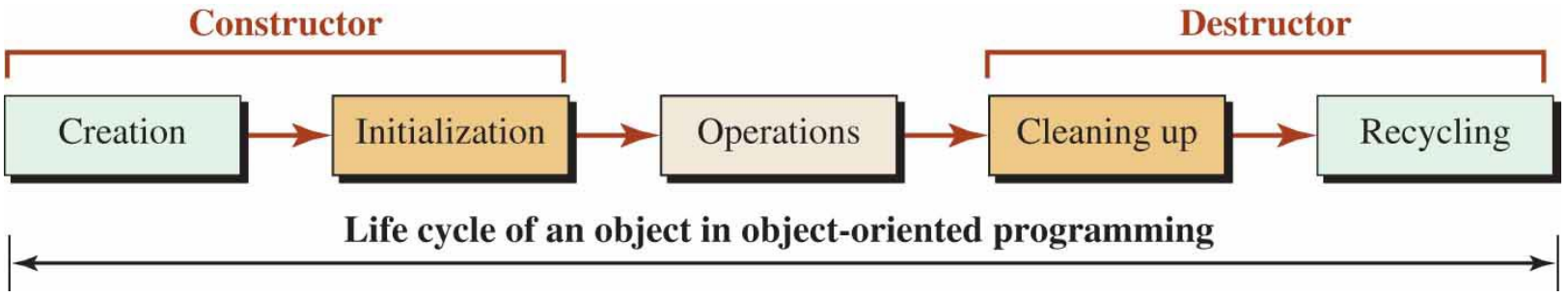
An object goes through five steps.

It is *created* and *initialized* by a special member function called a constructor (steps 1 and 2).

It applies some *operations* requested by the application on itself (step 3).

It is *cleaned up* and *recycled* by another special member function called a destructor (steps 4 and 5).

Figure 7.6 ***Life cycle of an object***



Constructors

A *constructor* is a member function that creates an object when it is called and can initialize the data members of an object when it is executed.

Constructor Declaration

A constructor has no return value, its name is the same as the name of the class, and it cannot have the const qualifier because it initializes the value of the data members.

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

Constructors

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

Parameter Constructor

Normally we have a *parameter constructor* that initializes the data members of each instance with specified values.

The parameter constructor can be overloaded for a class.

Default Constructor

The default constructor is a constructor with no parameters.

The default constructor cannot be overloaded for a class.

Copy Constructor

A copy constructor copies the data member values of the given object to create a new object.

The copy constructor cannot be overloaded for a class.

Constructors

Constructor Definition

A constructor is a member function, but a special one. It cannot have a return value and its name is the same as the name of the class.

```
// Definition of a parameter constructor
Circle :: Circle (double rds)
: radius(rds) // Initialization list
{
     // Any other statements
}
```

```
// Definition of a default constructor
Circle :: Circle()
: radius(0.0)  // Initialization list.
{
     // Any other statements
}
```

```
// Definition of a copy constructor
Circle :: Circle(const Circle& cr)
: radius(cr.radius)  // Initialization list
{
     // Any other statements
}
```

Destructors

A destructor is guaranteed to be automatically called and executed by the system when the object instantiated from the class goes out of scope.

Like a constructor, a *destructor* has two special characteristics.

The name of the destructor is the name of the class preceded by a tilde symbol (~).

Like a constructor, a destructor cannot have a return value (not even void) because it returns nothing.

Destructor Definition

The definition of a destructor is similar to the other three member functions, but

```
class Circle
{
      ...
      public:
            ...
            ~Circle();    // Destructor
}
```

```
// Definition of a destructor
Circle :: ~Circle()
{
      // Any statements as needed
}
```

Creating and Destroying Objects

Figure 7.7 *Object creating and destroying for a class type*

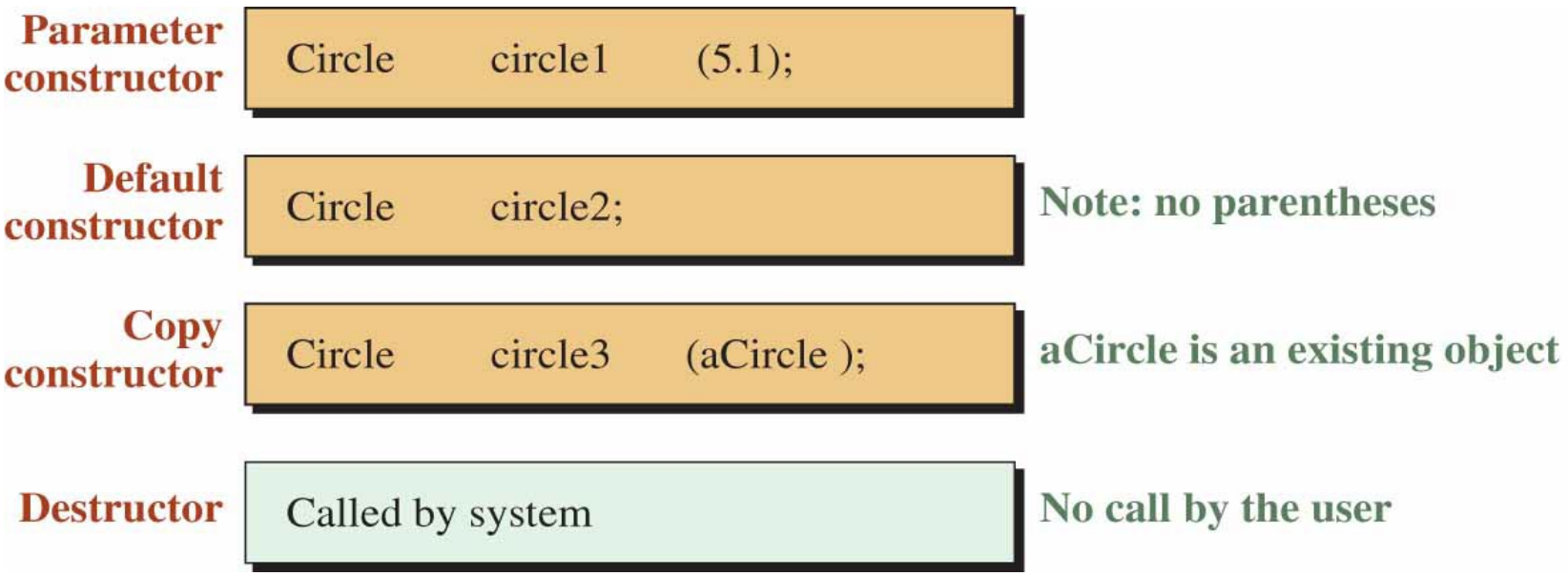


Table 7.3 *Comparison between variables and objects of classes*

Member	Class type	Built-in type
Parameter constructor	Circle circle1 (10.0);	double x1 = 10.0;
Default constructor	Circle circle2;	double x2;
Copy constructor	Circle circle3 (circle1);	None
Destructor	No call	No call

Circle Class with Constructors and Destructor Part 1

Program 7.2 A complete Circle class

```
1  /*************************************************************
2   * A program to use a class in object-oriented programming     *
3   *************************************************************/
4  #include <iostream>
5  using namespace std;
6
7  /*************************************************************
8   * Class Definition:                                           *
9   * declaration of parameter constructor, default constructor,  *
10  * copy constructor, destructor, and other member functions    *
11  *************************************************************/
12 class Circle
13 {
14      private:
15          double radius;
16      public:
17          Circle (double radius); // Parameter Constructor
18          Circle (); // Default Constructor
19          ~Circle (); // Destructor
20          Circle (const Circle& circle); // Copy Constructor
```

Circle Class with Constructors and Destructor Part 2

Program 7.2 *A complete Circle class*

```
21          void setRadius (double radius); // Mutator
22          double getRadius () const; // Accessor
23          double getArea () const; // Accessor
24          double getPerimeter () const; // Accessor
25  };
26  /**************************************************************
27   * Member Function Definition:                                *
28   * Definition of parameter constructor, default constructor,  *
29   * copy constructor, destructor, and other member functions   *
30   **************************************************************/
31  // Definition of parameter constructor
32  Circle :: Circle (double rds)
33  : radius (rds)
34  {
35      cout << "The parameter constructor was called. " << endl;
36  }
37  // Definition of default constructor
38  Circle :: Circle ()
39  : radius (0.0)
40  {
```

Circle Class with Constructors and Destructor Part 3

Program 7.2 *A complete Circle class*

```
41        cout << "The default constructor was called. " << endl;
42  }
43  // Definition of copy constructor
44  Circle :: Circle (const Circle& circle)
45  : radius (circle.radius)
46  {
47       cout << "The copy constructor was called. " << endl;
48  }
49  // Definition of destructor
50  Circle :: ~Circle ()
51  {
52       cout << "The destructor was called for circle with radius " ;
53       cout << endl;
54  }
55  // Definition of setRadius member function
56  void Circle :: setRadius (double value)
57  {
58       radius = value;
59  }
60  // Definition of getRadius member function
```

Circle Class with Constructors and Destructor Part 4

Program 7.2 *A complete Circle class*

```
61  double Circle :: getRadius () const
62  {
63       return radius;
64  }
65  // Definition of getArea member function
66  double Circle :: getArea () const
67  {
68       const double PI = 3.14;
69       return (PI * radius * radius);
70  }
71  // Definition of getPerimeter member function
72  double Circle :: getPerimeter () const
73  {
74       const double PI = 3.14;
75       return (2 * PI * radius);
76  }
77  /*************************************************************
78   * Application :                                             *
79   * Creating three objects of class Circle (circle1, circle2,  *
80   * and circle3) and applying some operation on each object    *
```

Circle Class with Constructors and Destructor Part 5

Program 7.2 *A complete Circle class*

```
 ********************************************************/
int main ()
{
// Instantiation of circle1 and applying operations on it
Circle circle1 (5.2);
cout << "Radius: " << circle1.getRadius() << endl;
cout << "Area: " << circle1.getArea() << endl;
cout << "Perimeter: " << circle1.getPerimeter() << endl << endl;
// Instantiation of circle2 and applying operations on it
Circle circle2 (circle1);
cout << "Radius: " << circle2.getRadius() << endl;
cout << "Area: " << circle2.getArea() << endl;
cout << "Perimeter: " << circle2.getPerimeter() << endl << endl;
// Instantiation of circle3 and applying operations on it
Circle circle3;
cout << "Radius: " << circle3.getRadius() << endl;
cout << "Area: " << circle3.getArea() << endl;
cout << "Perimeter: " << circle3.getPerimeter() << endl << endl;
// Calls to destructors occur here
return 0;
```

Circle Class with Constructors and Destructor Output

Program 7.2 *A complete Circle class*

```
101 }
```

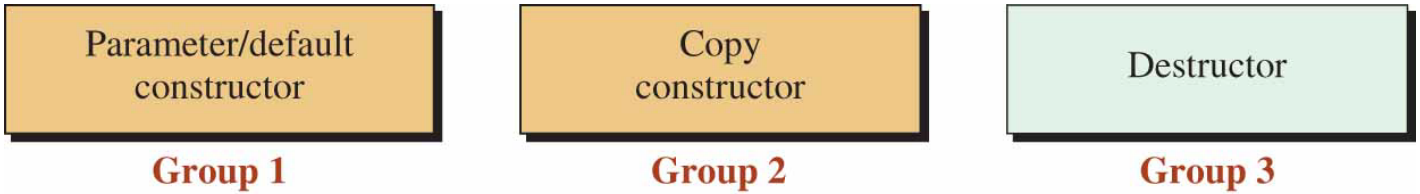
```
Run:
The parameter constructor was called.
Radius: 5.2
Area: 84.9056
Perimeter: 32.656

The copy constructor was called.
Radius: 5.2
Area: 84.9056
Perimeter: 32.656

The default constructor was called.
Radius: 0
Area: 0
Perimeter: 0

The destructor was called for circle with radius: 0
The destructor was called for circle with radius: 5.2
The destructor was called for circle with radius: 5.2
```

Required Member Functions



Note:
We need at least one member from each group. If we do not define at least one member from each group, the system provides one.

Group 1 consists of parameter constructor and the default constructor.
We need to have at least one of these constructors; we may sometimes need both.
If we provide either of them, the system does not provide any for us.
If we provide none of them, the system provides a default constructor, referred to as a *synthesized default constructor*, that initializes each member to what is left over as garbage in the system.

The second group is the copy constructor.
A class needs to have one and only one copy constructor, but if we do not provide one, the system provides one for us, which is referred to as the *synthesized copy constructor*.
Most of the time it is better to create our own copy constructor.

The third group is the destructor.
A class needs to have one and only one destructor but if we do not provide one, the system provides one for us, which is referred to as the *synthetized destructor*.
Most of the time, the *synthesized destructor* is not what we want. It is better to create our own destructor.

Declare, Define, and Use a Class Part 1

Program 7.3 *Defining and creating three random numbers*

```
1  /*************************************************************
2   * A program to declare, define, and use a class that generates *
3   * a random-number integer between any given range defined in   *
4   * the constructor of the class.                                *
5   *************************************************************/
6  #include <iostream>
7  #include <cstdlib>
8  #include <ctime>
9  using namespace std;
10
11 /*************************************************************
12  * Class Definition (Declaration of data members and member    *
13  * functions) for a Random-number generator.                   *
14  *************************************************************/
15 class RandomInteger
16 {
17      private:
18          int low; // Data member
19          int high; // Data member
20          int value; // Data member
```

Declare, Define, and Use a Class Part 2

Program 7.3 *Defining and creating three random numbers*

```
21      public:
22          RandomInteger (int low, int high); // Constructor
23          ~RandomInteger (); // Destructor
24          // Preventing a synthesized copy constructor
25          RandomInteger (const RandomInteger& random) = delete;
26          void print () const; // Accessor member function
27  };
28  /*************************************************************
29   * Definitions of constructor, destructor, and accessor member *
30   * functions for the random number generator class             *
31   *************************************************************/
32  // Constructor
33  RandomInteger :: RandomInteger (int lw, int hh)
34  :low (lw), high (hh)
35  {
36      srand (time (0));
37      int temp = rand ();
38      value = temp % (high - low + 1) + low;
39  }
40  // Destructor
```

- The new C++11 allows us to declare a copy constructor and set it to the keyword *delete*, which prevents the system from providing a synthesized copy constructor.

Declare, Define, and Use a Class Part 3

Program 7.3 *Defining and creating three random numbers*

```
41  RandomInteger :: ~RandomInteger ()
42  {
43  }
44  // Accessor member function
45  void RandomInteger :: print () const
46  {
47       cout << value << endl;
48  }
49  /*************************************************************
50   * Application to instantiate random number objects and print  *
51   * the value of the random number                              *
52   *************************************************************/
53  int main ( )
54  {
55       // Generating a random integer between 100 and 200
56       RandomInteger r1 (100, 200);
57       cout << "Random number between 100 and 200: ";
58       r1.print ();
59       // Generating a random integer between 400 and 600
60       RandomInteger r2 (400, 600);
```

Declare, Define, and Use a Class Part 4

Program 7.3 *Defining and creating three random numbers*

```
61      cout << "Random number between 400 and 600: ";
62      r2.print ();
63      // Generating a random integer between 400 and 600 ;
64      RandomInteger r3 (1500, 2000);
65      cout << "Random number between 1500 and 2000: ";
66      r3.print ();
67      return 0;
68  }
```

```
Run:
Random number between 100 and 200: 142

Random number between 400 and 600: 517

Random number between 1500 and 2000: 1662
```

In-class Exercise

- **Given an array with <u>all distinct elements</u>, find the largest <u>two distinct</u> elements in an array.**
  - Define a class: “CLargest”
    - ✓ Clargest.h, Clargest.cpp, Main.cpp
  - Define a member function: “findLargest ()”
    - ✓ to find the largest element
  - Define a member function: “findLargestTwo()”
    - ✓ To find the largest two distinct element
  - *Call the member function in main()*
  - *Print the largest two distinct elements in main()*

  - Test set:

```
Input: arr[] = {10, 4, 3, 50, 23, 90}
Output: 90, 50
```

```
Input: arr[] = {99, 77, 11, 15, 88, 1}
```

```
Input: arr[] = {10,9636, 2401, 777, 2080, 1, 50}
```

KU KONKUK UNIVERSITY

Encapsulation

Instance Data Members

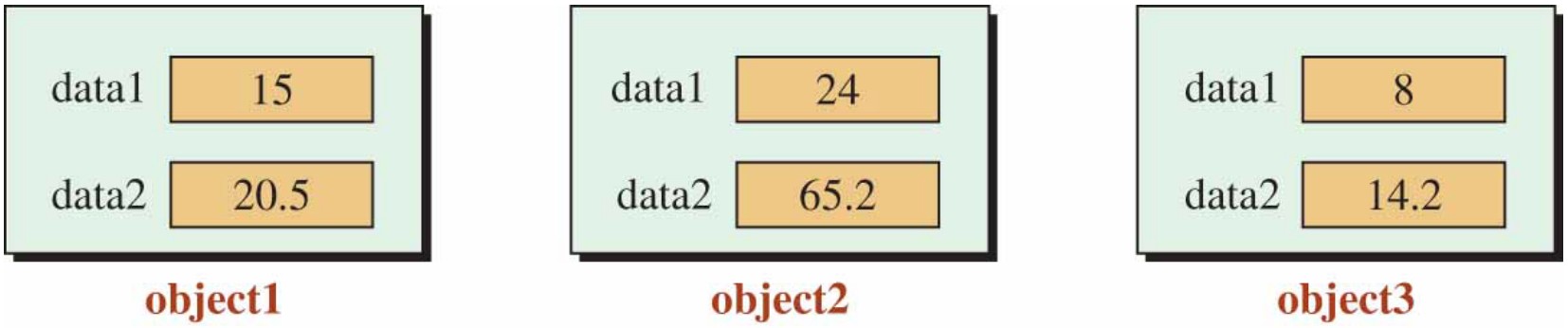
The instance data members of a class are normally private to be accessed only through instance member functions.

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***



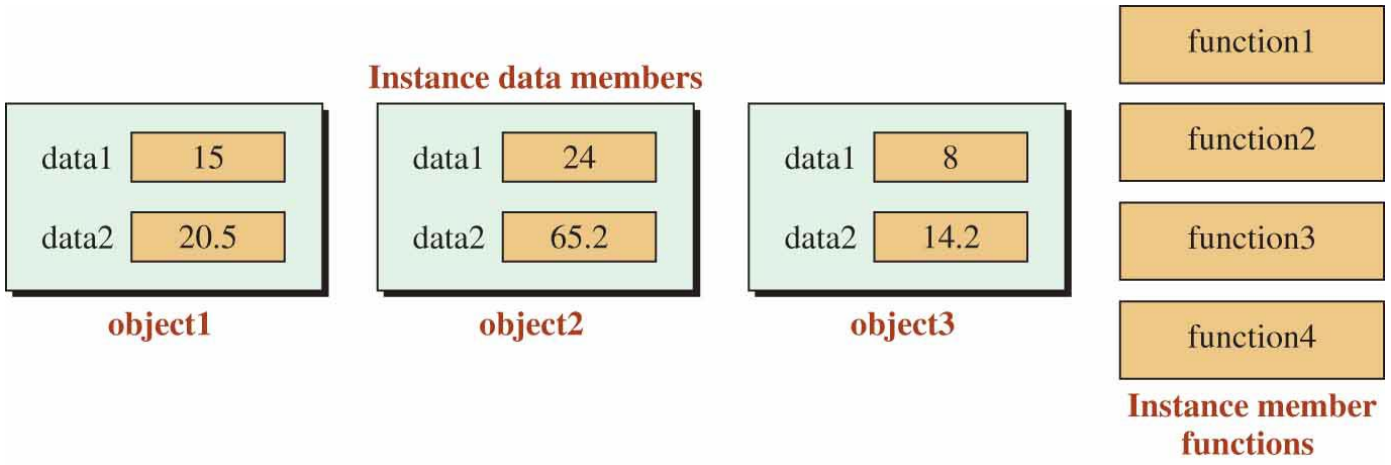
Instance data members encapsulaed in objects

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.



End of Class

Thank you

E-mail: youngcha@konkuk.ac.kr