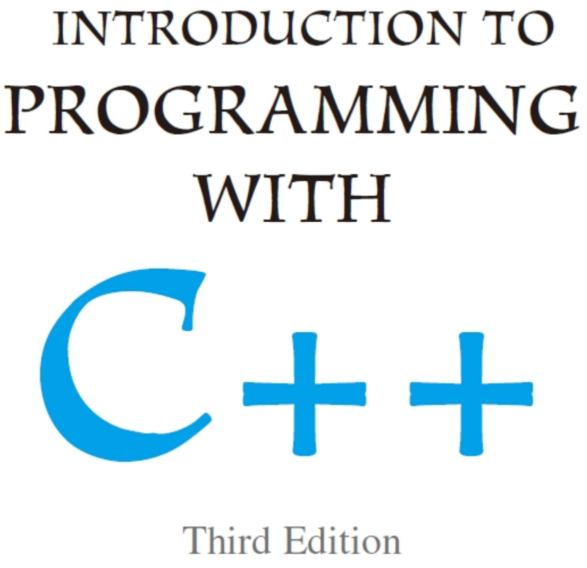
Introduction to Programming with C++

Chieh-Sen (Jason) Huang

Department of Applied Mathematics

National Sun Yat-sen University



INTRODUCTION TO PROGRAMMING WITH C++

Third Edition

Contents are based on book by Y. Daniel Liang

Classes for Objects

- Object-oriented programming (OOP) enable you to develop large-scale software systems effectively.

- Object-oriented programming involves programming using objects.

- An object represents an entity in the real world that can be distinctly identified.

- For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects.

- An object has a unique identity, **state**, and **behavior**.

Classes for Objects

- An object has a unique identity, **state**, and **behavior**.

  1. The **state** of an object (also known as **properties** or **attributes**) is represented by **data fields** with their **current values**.

     - A circle object, for example, has a data field, coordinate of center and radius, which are the properties that characterize a circle.

     - A rectangle object, for example, has data fields, width and height, which are the properties that characterize a rectangle.

  2. The **behavior** of an object (also known as **actions**) is defined by **functions**. To invoke a function on an object is to ask the object to **perform an action**.

     - For example, you may define a function named **getArea()** for circle objects. A circle object may invoke getArea() to return its area.

Classes for Objects

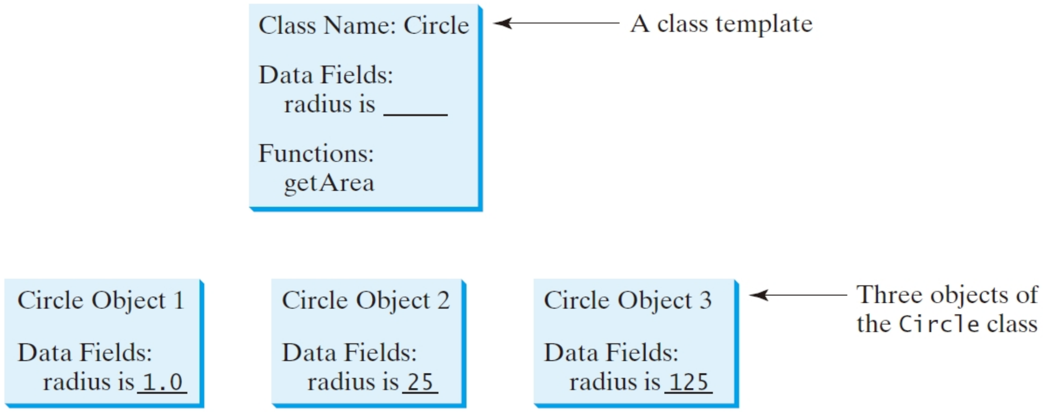
- A class is a template, blueprint, or contract that defines what an object's data fields and functions will be.

- An object is an **instance** of a class. You can create many instances of a class.

- Creating an instance is referred to as **instantiation**.

- The terms object and instance are often interchangeable.

- The relationship between classes and objects is analogous to the relationship between apple pie recipes and apple pies.



Classes for Objects

- A C++ class uses **variables** to define data fields and **functions** to define behaviors.

- A class provides functions of a special type, known as **constructors**, which are invoked when a new object is created.

- A **constructor** is a special kind of function. They are designed to perform initializing actions, such as initializing the data fields of objects.

```
class Circle
{
public:
    // The radius of this circle
  double radius;  <------------------------------------ Data field
    // Construct a circle object with default radius
  Circle(){       <----------------------------------
    radius = 1;                                      |
  }                                                  |
    // Construct a circle object                     |-- Constructors
  Circle(double newRadius){                          |
    radius = newRadius;                              |
  }                 <----------------------------------
    // Return the area of this circle
  double getArea(){ <--------------- ------------------- Function
    return radius * radius * 3.14159;
  }
}; // Must place a semicolon here
```

Classes for Objects

```
int main()
{
  Circle circle1; \\ List9_1.cpp
  Circle circle2(25);
  Circle circle3(125);

  cout << "The area of the circle of radius "
       << circle1.radius << " is " << circle1.getArea() << endl;
  cout << "The area of the circle of radius "
       << circle2.radius << " is " << circle2.getArea() << endl;
  cout << "The area of the circle of radius "
       << circle3.radius << " is " << circle3.getArea() << endl;

  // Modify circle radius
  circle2.radius = 100;
  cout << "The area of the circle of radius "
       << circle2.radius << " is " << circle2.getArea() << endl;

  return 0;
}
------------------------------------------------------------------
The area of the circle of radius 1 is 3.14159
The area of the circle of radius 25 is 1963.49
The area of the circle of radius 125 is 49087.3
The area of the circle of radius 100 is 31415.9
```

- The **public** keyword denotes that all data fields, constructors, and functions can be accessed from the objects of the class.

- If you don’t use the public keyword, the visibility is private by default. Private visibility will be introduced later.

Constructors

- A constructor is invoked to create an object.

- Constructors are a special kind of function, with three peculiarities:

  1. Constructors must have the same name as the class itself.
  2. Constructors do not have a return type-not even void.
  3. Constructors are invoked when an object is created. Constructors play the role of initializing objects.

- Like regular functions, constructors can be overloaded (i.e., multiple constructors with the same name but different signatures)

- Note that a variable (local or global) can be declared and initialized in one statement, **but** as a class member, **a data field** cannot be initialized when it is declared.

- For example, it would be wrong to do

  ```
  double radius = 5; // Wrong for data field declaration
  ```

Constructors

- A constructor is invoked when an object is created. The syntax to create an object using the no-arg constructor is
`ClassName objectName;`
e.g. `Circle circle1;`

- In OOP term, an object's **member refers** to its data fields and functions.

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

1. objectName.dataField references a data field in the object. `circle1.radius`
2. objectName.function(arguments) invokes a function on the object. `circle1.getArea()`

instance member variable/function

- The data field radius is referred to as an **instance member variable** or simply **instance variable**.

- For the same reason, the function getArea is referred to as an **instance member function** or **instance function**.

- Note When you define a custom class, capitalize the first letter of each word in a class name.

- In C++, you can use the assignment operator = to copy the contents from one object to the other.

- An object contains data and may invoke functions. This may lead you to think that an object is quite large.

- It isn't, though. Data are physically stored in an object, but functions are not. Since functions are shared by all objects of the same class, the compiler creates just one copy for sharing.

  ```
  cout << sizeof(circle1) << endl; \\ 8 bytes
  ```

anonymous objects

- Occasionally you may create an object and use it only once. In this case, you don't have to name it. Such objects are called anonymous objects.

  ```
  cout << "Area is " << Circle().getArea() << endl;
  ```

- To create a named object using the no-arg constructor, you can NOT use the parentheses after the constructor name (e.g., you use `Circle circle1` rather than ~~Circle circle1()~~).

  Homework: Add coordinate of center (x, y) to the Circle calss.

Separating Class Definition from Implementation

- C++ allows you to separate class definition from implementation.

- The class definition describes the contract of the class and the class implementation carries out the contract.

- The class definition simply lists all the data fields, constructor prototypes, and function prototypes.

- The class implementation implements the constructors and functions.

- The class definition and implementation may be in two separate files. Both files should have the same name but different extension names.

- The class definition file has an extension name .h (h means header) and the class implementation file an extension name .cpp.

Separating Class Definition from Implementation

```
//Circle.h
class Circle
{
  public:
    // The radius of this circle
    double radius;
    // Construct a default circle object
    Circle();
    // Construct a circle object
    Circle(double);
    // Return the area of this circle
    double getArea();
};
```

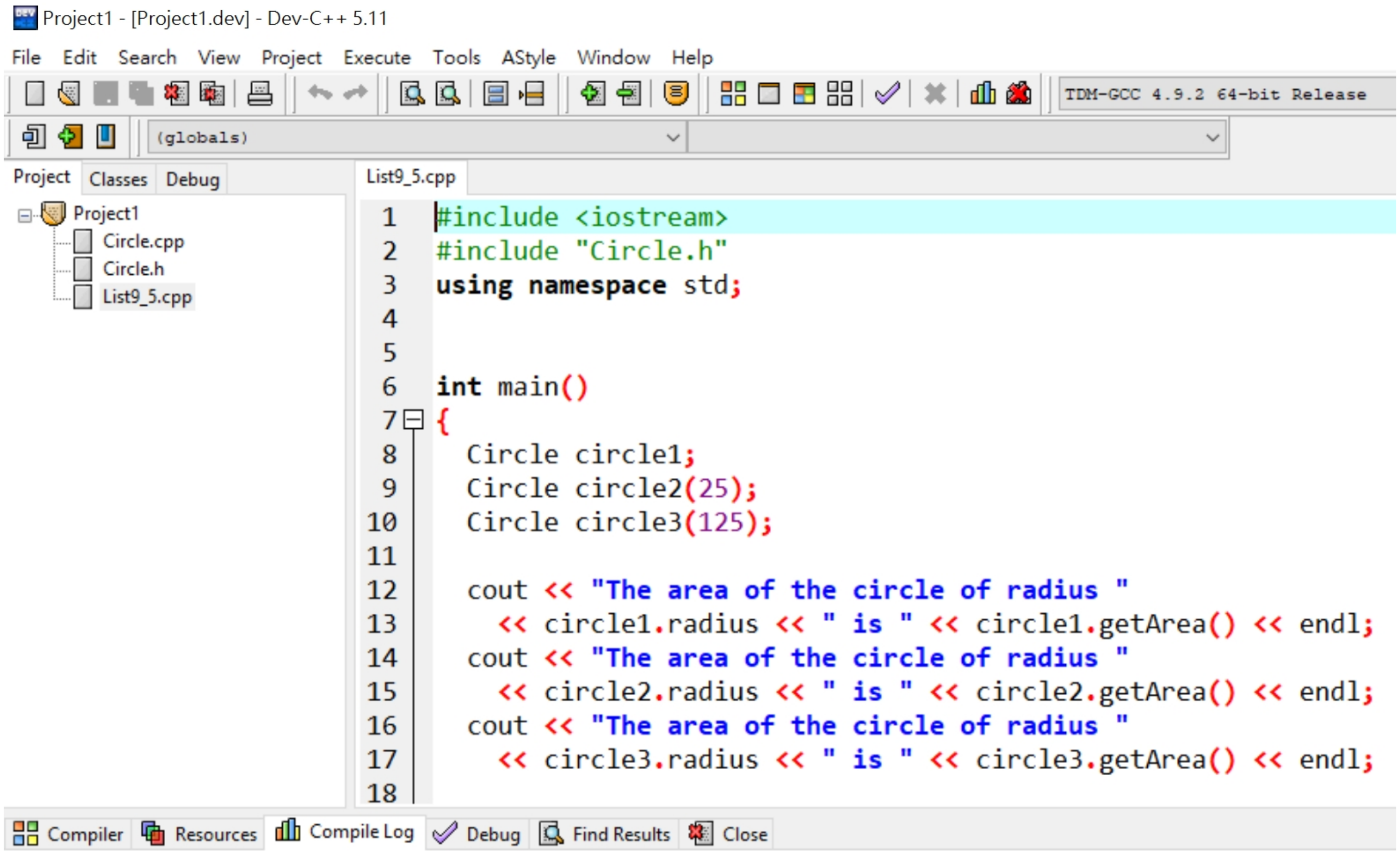
```
//Circle.cpp
#include "Circle.h"
  // Construct a default circle object
Circle::Circle()
{
  radius = 1;
}
  // Construct a circle object
Circle::Circle(double newRadius)
{
  radius = newRadius;
}
  // Return the area of this circle
double Circle::getArea()
{
  return radius * radius * 3.14159;
}
```

- The `::` symbol, known as the **binary scope resolution operator**, specifies the scope of a class member in a class.

- Circle:: preceding each constructor and function in the Circle class tells the compiler that these constructors and functions are defined in the Circle class

Separating Class Definition from Implementation

- Open a new project.
- add all files to the project.



Project1 - [Project1.dev] - Dev-C++ 5.11
File Edit Search View Project Execute Tools AStyle Window Help
TDM-GCC 4.9.2 64-bit Release
(globals)
Project Classes Debug
Project1
Circle.cpp
Circle.h
List9_5.cpp
List9_5.cpp
Compiler Resources Compile Log Debug Find Results Close

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


int main()
{
  Circle circle1;
  Circle circle2(25);
  Circle circle3(125);

  cout << "The area of the circle of radius "
    << circle1.radius << " is " << circle1.getArea() << endl;
  cout << "The area of the circle of radius "
    << circle2.radius << " is " << circle2.getArea() << endl;
  cout << "The area of the circle of radius "
    << circle3.radius << " is " << circle3.getArea() << endl;

```

Inline Functions in Classes

- When a function is implemented inside a class definition, it automatically becomes an inline function. This is also known as **inline definition**.

- There is another way to define inline functions for classes. You may define inline functions in the class's implementation file.

```
// class definition
class A
{
public:
double f2();
};
// Implement function as inline
inline double A::f2()
{
 // Return a number
}
```

Data Field Encapsulation

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

1. Data may be tampered with.
2. It makes the class difficult to maintain and vulnerable to bugs. Suppose you want to modify the Circle class to ensure that the radius is nonnegative after other programs have already used the class. You have to change not only the Circle class, but also the programs that use the Circle class.

Data Field Encapsulation

- To prevent direct modifications of properties, you should declare the data field **private**, using the private keyword. This is known as **data field encapsulation**.

  ```
  class Circle
  {
    public:
     Circle();
     Circle(double);
     double getArea();
    private:
     double radius;
  };
  ```

- A private data field cannot be accessed by an object through a direct reference outside the class.

- To make a private data field accessible, provide a get function to return the field's value.

- To enable a private data field to be updated, provide a set function to set a new value.

Data Field Encapsulation

- Colloquially, a get function is referred to as an accessor, and a set function is referred to as a mutator.

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

```
double Circle::getRadius()
{
  return radius;
}
    // Set a new radius
void Circle::setRadius(double newRadius)
{
  radius = (newRadius >= 0) ? newRadius : 0;
}
```

```
\\within main List9_11.cpp
cout << "The area of the circle of radius "
     << circle1.getRadius() << " is "
     << circle1.getArea() << endl;
cout << "The area of the circle of radius "
     << circle2.getRadius() << " is "
     << circle2.getArea() << endl;
  // Modify circle radius
circle2.setRadius(100);
cout << "The area of the circle of radius "
     << circle2.getRadius() << " is "
     << circle2.getArea() << endl;
------------------------------------------------
The area of the circle of radius 1 is 3.14159
The area of the circle of radius 5 is 78.5397
The area of the circle of radius 100 is 31415.9
```

Data Field Encapsulation

- Private data can be accessed only within their defining class.
- You cannot use `circle1.radius` in the client program.

The Scope of Variables

- The data fields are declared as variables and are accessible to all constructors and functions in the class.

- Data fields and functions can be in any order in a class.

- Though the class members can be in any order, the common style in C++ is to place public members first and then private members.

The Scope of Variables

- If a local variable has the same name as a data field, the local variable takes precedence, and the data field with the same name is hidden.

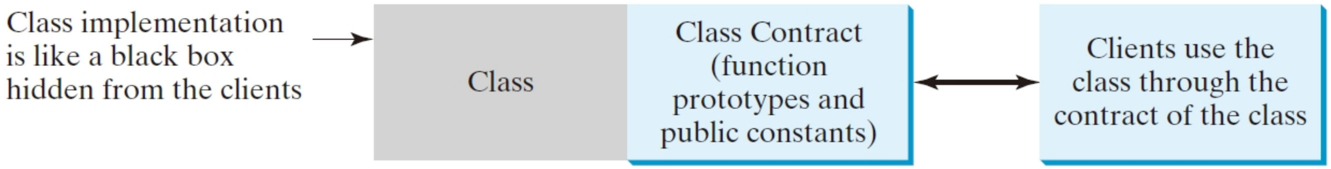
```
class Foo
{
  public:
  int x; // Data field
  int y; // Data field

  Foo()
  {
   x = 10;
   y = 10;
  }

  void p()
  {
   int x = 20; // Local variable
   cout << "x is " << x << endl;
   cout << "y is " << y << endl;
  }
};
```

```
int main()
{
  Foo foo;
  foo.p();
}
-------------------------------------
x is 20
y is 10
```

- x is declared as a data field in the Foo class, but is also defined as a local variable in the function p() with an initial value of 20. The latter x is displayed to the console.

Class Abstraction and Encapsulation

- Class abstraction is the separation of class implementation from the use of a class. The details of implementation are encapsulated and hidden from the user. This is known as class encapsulation.


Class implementation is like a black box hidden from the clients
Class
Class Contract (function prototypes and public constants)
Clients use the class through the contract of the class

- Your personal computer is made up of many components, such as a CPU, CD-ROM, floppy disk, motherboard, fan, and so on.

- Each component can be viewed as an object that has properties and functions.

- You don't need to know how those parts work internally. The internal implementation is encapsulated and hidden from you.

- You can build a computer without knowing how a component is implemented.