

C/C++ Program Design

LAB 11

CONTENTS

- ❑ Learn how to define and implement a **class**
- ❑ Learn how to create and use class **objects**
- ❑ Class **constructors** and **destructors**
- ❑ Master the difference between **private** and **public**
- ❑ Const member variables and const member functions
- ❑ Static member variables and static member functions
- ❑ Learn how to use **this** pointer
- ❑ Learn to use an **array of objects**

2 Knowledge Points

2.1 **Class** and its definition

2.2 Access specifier: **private** and **public**

2.3 Class **constructors** and **destructors**

2.4 **const** members and **static** members

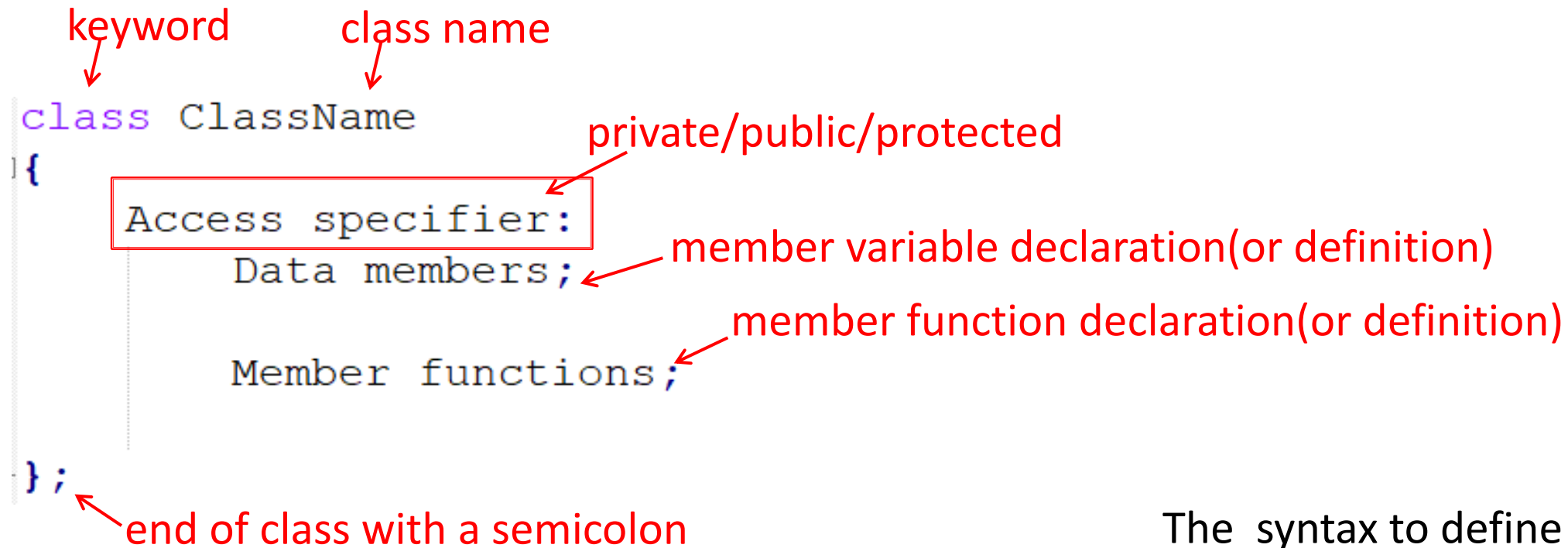
2.5 ***this*** pointer

2.6 An Array of Objects

2.1 Class

A class that uses data abstraction and encapsulation defines an abstract data type. Encapsulation enforces the separation of a class's interface and implementation.

The general syntax for a class declaration(or definition):



```
class ClassName
{
    Access specifier:
        Data members;
        Member functions;
};
```

The diagram illustrates the general syntax for a class declaration or definition in C++. It shows the following components with red arrows pointing to them:

- keyword**: Points to the `class` keyword.
- class name**: Points to `ClassName`.
- private/public/protected**: Points to the **Access specifier:** box.
- member variable declaration(or definition)**: Points to `Data members;`.
- member function declaration(or definition)**: Points to `Member functions;`.
- end of class with a semicolon**: Points to the closing brace and semicolon `};`.

The syntax to define an object:
ClassName **objectName**;

A class is a blueprint for an object!

Example: Define a class

```
rectangle.cpp > ...
1  #include <iostream>
2  using namespace std;
3
4  class Rectangle
5  {
6  public:
7      double width;
8      double height;
9
10     double getArea()
11     {
12         return width * height;
13     }
14
15 };
16
17 int main()
18 {
19     Rectangle r;
20     r.width = 1;
21     r.height = 2;
22
23     cout << "The width is:" << r.width << ", the height is:" << r.height << endl;
24     cout << "The area is:" << r.getArea() << endl;
25
26     return 0;
27 }
```

Define a class named Rectangle

The member variables of a class

The member function of a class. It is an inline function.

Create an object of Rectangle

Accessing data member by . operator

Accessing member function by .operator

Defining a class, creating an object and accessing the members of an object, these operations are as the same as we use in structure. The only difference is that we call a class **instance** as an object. You must use an object to access the attributes(data member) and member functions of a class if their access specifier is **public**.

In general, **private** specifier is used for data members and **public** specifier for member functions when defining a class.

2.2 Access specifier: **private** and **public**

The **private** keyword makes members private. Private members **can be accessed only inside the class**. We usually make **data private** to prevent them from being modified outside the class. This is known as **data encapsulation (data hiding)**.

The **public** keyword makes members public. Public members **can be accessed out of the class**. We usually make **functions public** for accessing outside the class.

```
rectangle2.cpp > ...
1  #include <iostream>
2  using namespace std;
3
4  class Rectangle
5  {
6  private:           private members
7      double width;
8      double height;
9  public:           public member
10     double getArea()
11     {
12         return width * height;
13     }
14
15 };
```

```
17 int main()
18 {
19     Rectangle r;
20     r.width = 1;
21     r.height = 2;
22
23     cout << "The area is:" << r.getArea() << endl;
24
25     return 0;
26 }
```

Private members cannot be accessed outside the class

Note: Private is the default access specifier for a class in C++. This means that if no access specifier is specified for the members in a class, it is considered private.

```

4  class Rectangle
5  {
6  private:
7      double width;
8      double height;
9  public:
10     double getArea()
11     {
12         return width * height;
13     }
14
15     double getWidth()
16     {
17         return width;
18     }
19
20     double getHeight()
21     {
22         return height;
23     }
24
25     void setWidth(double w)
26     {
27         width = w;
28     }
29
30     void setHeight(double h)
31     {
32         height = h;
33     }
34
35 };

```

A **private** data field cannot be accessed by an object from outside the class. To make a private data field accessible, provide a **getter** method to return its value. To enable a private data field to be updated, provide a **setter** method to set a new value.

```

38  int main()
39  {
40      Rectangle r;
41
42      cout << "The width is:" << r.getWidth()
43          << ", the height is:" << r.getHeight() << endl;
44      cout << "The area is:" << r.getArea() << endl;
45
46      return 0;
47  }

```

Access the private data by getters

```

The width is:6.93909e-310, the height is:4.65933e-310
The area is:0

```

The values of data members are not initialized, these values are invalid.

2.3 Class Constructors and Destructors

2.3.1 Constructors

Each class defines how objects of its type can be initialized. Classes control object initialization by defining one or more special member functions known as **constructors**. The job of a constructor is to initialize the data members of a class object. A constructor is run whenever an object of a class type is created.

A class constructor is a special member function:

1. Has exact the same name as the class
2. No return value
3. It is a public member function of the class
4. Invoked whenever you create objects of that class

If you do not provide any constructor, C++ compiler will implicitly **generates** a **synthetic default constructor** (has no parameters and an empty body) for you. This synthesized constructor initializes each data member of the class as follows:

- If there is an in-class initializer, use it to initialize the member.
- Otherwise, default-initialize the member.


```

4  class Rectangle
5  {      in-class initializer
6  private:
7      double width = 1;
8      double height = 2;
9  public:
10     double getArea()
11     {
12         return width * height;
13     }
14
15     double getWidth()
16     {
17         return width;
18     }
19
20     double getHeight()
21     {
22         return height;
23     }
24
25     void setWidth(double w)
26     {
27         width = w;
28     }
29
30     void setHeight(double h)
31     {
32         height = h;
33     }
34
35 };

```

```

38 int main()
39 {
40     Rectangle r;
41
42     cout << "The width is:" << r.getWidth()
43         << ", the height is:" << r.getHeight() << endl;
44     cout << "The area is:" << r.getArea() << endl;
45
46     return 0;
47 }

```

Defining an object will invoke the default constructor, using in-class initializer to initialize the data members.

```

The width is:1, the height is:2
The area is:2

```

The values of data members are the in-class initializers.

Note: If your compiler does not support in-class initializers, you must provide a constructor explicitly to initialize every member of built-in type.

The declaration of a class is in a .h file

```
C rectangle.h > ...
1  #ifndef RECTANGLE_H
2  #define RECTANGLE_H
3  class Rectangle
4  {
5  private:
6      double width;
7      double height;
8
9  public:
10     //default constructor
11     Rectangle();
12
13     // parameterized constructor
14     Rectangle(double w, double h);
15
16     double getArea();
17     double getWidth();
18     double getHeight();
19     void setWidth(double w);
20     void setHeight(double h);
21
22 };
23 #endif
```

The implementations of functions are in a .cpp file

```
rectangleclass.cpp > ...
1  #include "rectangle.h"
2
3  Rectangle::Rectangle()
4  {
5      width = 1;
6      height = 2;
7  }
8
9  Rectangle::Rectangle(double w, double h)
10 {
11     width = w;
12     height = h;
13 }
14
15 double Rectangle::getArea()
16 {
17     return width * height;
18 }
19
20 double Rectangle::getWidth()
21 {
22     return width;
23 }
```

Defining member functions outside the class must use class name and class resolution specifier ::.

Note: Once you define a constructor, the compiler no longer provides you a default constructor.

```
rectanglemain.cpp > ...
1  #include <iostream>
2  #include "rectangle.h"
3  using namespace std;
4
5  int main()
6  {
7      Rectangle r1;
8      Rectangle r2(3,5);
9
10     cout << "The width of r1 is:" << r1.getWidth() << ", the height of r1 is:" << r1.getHeight() << endl;
11     cout << "The width of r2 is:" << r2.getWidth() << ", the height of r2 is:" << r2.getHeight() << endl;
12
13     cout << "The area of r1 is:" << r1.getArea() << endl;
14     cout << "The area of r2 is:" << r2.getArea() << endl;
15
16
17     return 0;
18 }
```

Invoke default constructor

Invoke parameterized constructor

```
The width of r1 is:1, the height of r1 is:2
The width of r2 is:3, the height of r2 is:5
The area of r1 is:2
The area of r2 is:15
```

Invoking constructors to initialize the data members.

Default constructor

A **default constructor** is a constructor that is used to create an object when you don't provide explicit initialization values. If you don't provide any constructor, the compiler will automatically supply a default constructor. It's an implicit version of a default constructor. There is **only one default constructor** in a class.

If you define any constructor for a class, the compiler will not provide default constructor.
You must define your own default constructor that takes no arguments.

You can define a default constructor in **two ways**. **One** is to provide default values for all the arguments to the existing constructor:

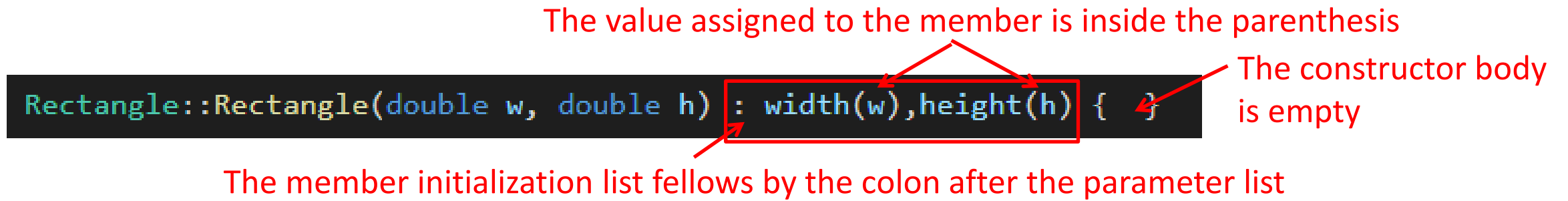
```
Rectangle(double w = 1, double h = 2);
```

The **second** is to use function overloading to define a second constructor that has no arguments:

```
Rectangle( );
```

NOTE: You can have **only one default constructor**, so be sure that you don't do both.

Another way to initialize the data members in a constructor is to use member initialization list.



The value assigned to the member is inside the parenthesis

```
Rectangle::Rectangle(double w, double h) : width(w), height(h) { }
```

The constructor body is empty

The member initialization list follows by the colon after the parameter list

The diagram shows a C++ constructor definition for a class named Rectangle. The code is: `Rectangle::Rectangle(double w, double h) : width(w), height(h) { }`. Red arrows point from text annotations to specific parts of the code: one points to the parentheses in `width(w)` and `height(h)` with the text 'The value assigned to the member is inside the parenthesis'; another points to the curly braces `{ }` with the text 'The constructor body is empty'; and a third points to the colon `:` with the text 'The member initialization list follows by the colon after the parameter list'. A red box highlights the member initialization list `: width(w), height(h)`.

Under the following circumstance, you may use the member initialization list.

- There is an object as a data member in your class definition
- There is a const data member
- There is a reference data member

Using constructor

After you define default constructor, you can create object variables without initializing them explicitly.

Invoking default constructor

```
Rectangle r1; // call the default constructor implicitly
Rectangle r2 = Rectangle(); // call the default constructor explicitly
Rectangle *pr = new Rectangle; // call the default constructor implicitly

Rectangle r3(3,5); // call the non-default constructor

Rectangle r4(); // declare a function
```

r4() is a function that returns a Rectangle object.

When you implicitly call the default constructor, do not use parentheses.

Destructors

A destructor is applied automatically to a class object before the end of its lifetime. The primary use of a destructor is to free resources acquired within the constructor or during the lifetime of the object.

A class destructor is also a special member function:

1. A destructor name is the same as the classname but begins with tilde(~) sign.
2. Destructor has no return value.
3. A destructor has no arguments.
4. There can be only one destructor in a class (Destructor can not be overloaded).
5. The compiler always creates a default destructor if you fail to provide one for a class.
6. Invoke when an object goes out of scope or the delete is applied to a pointer to the object.

```

C rectangle.h > ...
1  #ifndef RECTANGLE_H
2  #define RECTANGLE_H
3  class Rectangle
4  {
5  private:
6      double width;
7      double height;
8
9  public:
10     //default constructor
11     Rectangle();
12
13     // parameterized constructor
14     Rectangle(double w, double h);
15
16     // destructor
17     ~Rectangle();
18
19     double getArea();
20     double getWidth();
21     double getHeight();
22     void setWidth(double w);
23     void setHeight(double h);
24
25 };
26 #endif

```

```

Rectangle::~~Rectangle()
{
    std::cout << "Destructor is called." << std::endl;
}

```

```

rectanglemain.cpp > ...
1  #include <iostream>
2  #include "rectangle.h"
3  using namespace std;
4
5  int main()
6  {
7      Rectangle r1; // call the default constructor implicitly
8      Rectangle r2 = Rectangle(); // call the default constructor explicitly
9
10     Rectangle r3(3,5); // call the non-default constructor
11
12     cout << "The area of r1 is:" << r1.getArea() << endl;
13     cout << "The area of r2 is:" << r2.getArea() << endl;
14     cout << "The area of r3 is:" << r3.getArea() << endl;
15
16     return 0;
17
18 }

```

Creating three objects

```

The area of r1 is:2
The area of r2 is:2
The area of r3 is:15
Destructor is called.
Destructor is called.
Destructor is called.

```

When the object is out of its scope, the destructor is called automatically.

Memberwise initialization

By default, when we initialize one class object with another, the data members of the class are copied in turn. This is called **default memberwise initialization**.

```
rectanglemain.cpp > main()
1  #include <iostream>
2  #include "rectangle.h"
3  using namespace std;
4
5  int main()
6  {
7      Rectangle r1; // call the default constructor implicitly
8      Rectangle r2 = Rectangle(); // call the default constructor explicitly
9
10     Rectangle r3(3,5); // call the non-default constructor
11
12     Rectangle r4 = r3; ← width and height are copied in turn from r3 to r4.
13
14     cout << "The width of r3 is:" << r3.getWidth() << ", the height of r3 is:" << r3.getHeight() << endl;
15     cout << "The width of r4 is:" << r4.getWidth() << ", the height of r4 is:" << r4.getHeight() << endl;
16
17     cout << "The area of r3 is:" << r3.getArea() << endl;
18     cout << "The area of r4 is:" << r4.getArea() << endl;
19
20
21     return 0;
22 }
```

```
The width of r3 is:3, the height of r3 is:5
The width of r4 is:3, the height of r4 is:5
The area of r3 is:15
The area of r4 is:15
Destructor is called.
Destructor is called.
Destructor is called.
Destructor is called.
```

In the case of the Rectangle class, default memberwise initialization correctly copies the class data members and there is nothing we need to do explicitly.

```

int main()
{
    using std::cout;
    cout << "Using constructors to create new objects\n";
    Stock stock1("NanoSmart", 12, 20.0); //syntax 1
    stock1.show();
    cout << '\n';

    Stock stock2 = Stock("Boffo Objects", 2, 2.0); //syntax 2
    stock2.show();
    cout << '\n';

    cout << "Assigning stock1 to stock2:\n";
    stock2 = stock1;

    cout << "Listing stock1 and stock2:\n";
    cout << "stock1:";
    stock1.show();
    cout << '\n';
    cout << "stock2:";
    stock2.show();
    cout << '\n';

    cout << "Using a constructor to reset an object\n";
    stock1 = Stock("Nifty Foods", 10, 50.0); // temp object
    cout << "Revised stock1:\n";
    stock1.show();
    cout << "Done\n";
    return 0;
}

```

Creates an object

Assigns one object to another

Using a constructor to reset an object

The last object created is the first deleted.
stock2 is the last object, stock1 is the first object.

```

Using constructors to create new objects
Constructor using NanoSmart called
Company: NanoSmart Shares: 12
Share Price: $20.000 Total Worth: $240.00

Constructor using Boffo Objects called
Company: Boffo Objects Shares: 2
Share Price: $2.000 Total Worth: $4.00

Assigning stock1 to stock2:
Listing stock1 and stock2:
stock1:Company: NanoSmart Shares: 12
Share Price: $20.000 Total Worth: $240.00

stock2:Company: NanoSmart Shares: 12
Share Price: $20.000 Total Worth: $240.00

Using a constructor to reset an object
Constructor using Nifty Foods called
Bye, Nifty Foods!
Revised stock1:
Company: Nifty Foods Shares: 10
Share Price: $50.000 Total Worth: $500.00
Done
Bye, NanoSmart!
Bye, Nifty Foods!

```

Deleted the temporary object

2.4 *const members and static members*

2.4.1 const member variables and member functions

1. const member variables

If some member variables need not be modified, these variables can be defined as **const**.

These const member variables can be initialized by **in-class initializers** or **initialization list**.

```
C person.h > ...
1  #include <iostream>
2
3  #ifndef PERSON_H
4  #define PERSON_H
5  class Person
6  {
7  private:
8      const int SIZE = 10;
9      int age = 20;
10
11 public:
12
13     Person() { } // define an empty default constructor
14     Person(int a) { age = a;}
15
16     void Show()
17     {
18         std::cout << "The size is:" << SIZE << ", the age is:" << age << std::endl;
19     }
20
21 };
22
23 #endif
```

```
G+ personmain.cpp > main()
1  #include <iostream>
2  #include "person.h"
3
4  int main()
5  {
6      Person p1;
7      Person p2(18);
8
9      p1.Show();
10     p2.Show();
11
12     return 0;
13 }
```

Invoke the default constructor and use the in-class initializers to initialize the members

```
The size is:10, the age is:20
The size is:10, the age is:18
```

```

C person.h > ...
1  #include <iostream>
2
3  #ifndef PERSON_H
4  #define PERSON_H
5  class Person
6  {
7  private:
8      const int SIZE;
9      int age;
10
11 public:
12
13     Person() { SIZE = 10; age = 20; } // define an empty default constructor
14     Person(int a) { age = a; }
15
16     void Show()
17     {
18         std::cout << "The size is:" << SIZE << ", the age is:" << age << std::endl;
19     }
20
21 };
22
23 #endif

```

no in-class initializers

Inside constructor, assign a value to a const member is not allowed.

```

private:
    const int SIZE;
    int age;

public:
    Person() : SIZE(10), age(20) { } // define an empty default constructor
    Person(int a) : SIZE(10) { age = a; }

```

Using member initialization list to initialize the const member.

```

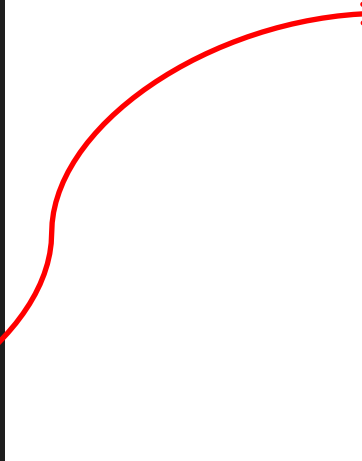
person.h: In constructor 'Person::Person()':
person.h:13:5: error: uninitialized const member in 'const int' [-fpermissive]
13 |     Person() {SIZE = 10; age = 20; }
    |     ~~~~~
person.h:8:15: note: 'const int Person::SIZE' should be initialized
8 |     const int SIZE;
    |     ~~~~~
person.h:13:21: error: assignment of read-only member 'Person::SIZE'
13 |     Person() {SIZE = 10; age = 20; }
    |     ~~~~~

```

2. const member functions

If some member functions need not modify the data members, define them as const member functions. The **const** modifier follows the parameter list of the function. A const member function defined outside the class body must specify the **const** modifier in both its declaration and definition.

```
C complex.h > ...
1  #ifndef COMPLEX_H
2  #define COMPLEX_H
3  class Complex
4  {
5  private:
6      double real;
7      double imag;
8
9  public:
10     Complex() : real(1), imag(1) { }
11     Complex(double re, double im)
12     {   real = re;
13         imag = im;
14     }
15
16     Complex Add(const Complex &rhs);
17     void Show() const;
18 };
19
20 #endif
```



```
void Complex::Show() const
{
    std::cout << real << (imag >= 0? "+":"" ) << imag << "i";
}
```

2.4.2 static member variables and member functions

1. static member variables

```
class Person
{
private:
    const int SIZE = 10;
    char name[SIZE];
    int age;
```

If we want to use char array to store name, the const SIZE can not be allowed to represent the length of the array. Because SIZE must be a static member.

```
const int Person::SIZE
a nonstatic member reference must be relative to a specific object
```

Another way to solve this problem is to use enumeration.

```
private:
//    static const int SIZE = 10;
    enum {SIZE = 10};
    char name[SIZE];
    int age;
```

Define an enumerator

```
1  #include <iostream>
2  #include <cstring>
3
4  #ifndef PERSON_H
5  #define PERSON_H
6  class Person
7  {
8  private:
9      static const int SIZE = 10;
10     char name[SIZE];
11     int age;
12
13 public:
14
15     Person() : name("Peter"),age(20) { }
16     Person( const char *n , int a)
17     {
18         strcpy(name,n);
19         age = a;
20     }
21
22     void Show() const
23     {
24         std::cout << "The size is:" << SIZE << ", the age is:" << age << std::endl;
25     }
26
27 };
```

Define SIZE as a static const variable and initialize it with a const value.

A **static** data member represents a single, shared instance of that member that is accessible to all the objects of that class. The type of a static data member can be const, reference, array, class type, and so forth.

```
person.h > count
1  #include <iostream>
2  #include "cstring"
3
4  #ifndef PERSON_H
5  #define PERSON_H
6  class Person
7  {
8  private:
9  //    static const int SIZE = 10;
10     enum {SIZE = 10};
11     char name[SIZE];
12     int age;
13     static int count;
14
15 public:
16
17     Person() : name("Peter"),age(20) { count ++; } // define an empty default constructor
18     Person(const char *n, int a)
19     {
20         strcpy(name,n);
21         age = a;
22         count ++;
23     }
24
25     void Show() const
26     {
27         std::cout << "The name is:" << name << ", the age is:" << age << std::endl;
28     }
29
30     static void ShowCount()
31     {
32         std::cout << count << " objects are created." << std::endl;
33     }
34
35 };
36
37 int Person :: count = 0;
38
39 #endif
```

static member variable can not be initialized when it is defined

static member variable must be initialized outside the class definition with class scope operator ::

If there is an implementation file of a class, put this initialization statement in it. Otherwise, it'll cause a link error.

```
personmain.cpp > ...
1  #include <iostream>
2  #include "person.h"
3
4  int main()
5  {
6      Person p1;
7      Person p2("ALICE",18);
8
9      p1.Show();
10     p2.Show();
11
12     p1.ShowCount();
13     Person::ShowCount();
14
15     return 0;
16 }
```

```
The name is:Peter, the age is:20
The name is:ALICE, the age is:18
2 objects are created.
2 objects are created.
```

Because static data members are not part of individual objects of the class type, they are not defined when we create objects of the class. As a result, they are not initialized by the class's constructors. Moreover, in general, we may not initialize a static member inside the class.

2. static member functions

A static member function can be invoked independently of a class object in exactly this way. A member function can be declared as static only if it does not access any nonstatic class members. We make it static by prefacing its declaration within the class definition with the keyword **static**.

Define a static member function with **static** keyword

```
static void ShowCount()  
{  
    std::cout << count << " objects are created." << std::endl;  
}
```

A static member function can not be a const member function.

You can use object or class to access the static members.

```
personmain.cpp > ...  
1  #include <iostream>  
2  #include "person.h"  
3  
4  int main()  
5  {  
6      Person p1;  
7      Person p2("ALICE",18);  
8  
9      p1.Show();  
10     p2.Show();  
11  
12     p1.ShowCount();  
13     Person::ShowCount();  
14  
15     return 0;  
16 }
```

When defined a static member function outside the class body, the **static** keyword is not needed.

2.5 *this* pointer

There's only one copy of each class's functionality, but there can be many objects of a class. Every object has access to its own address through a pointer called *this*. The *this* pointer is passed(by the compiler) as an implicit argument to each of the object's **non-static** member functions.

```
complexclass.cpp > ...
1  #include <iostream>
2  #include "complex.h"
3
4  Complex Complex::Add(const Complex &rhs)
5  {
6      this->real += rhs.real;
7      this->imag += rhs.imag;
8
9      return *this;
10 }
11
12 void Complex::Show() const
13 {
14     std::cout << real << (imag >= 0? "+" : "") << imag << "i";
15 }
```

```
complexmain.cpp > main()
1  #include <iostream>
2  #include "complex.h"
3
4  int main()
5  {
6      Complex c1;
7      Complex c2(2,-4);
8
9      c1.Show();
10     std::cout << " + ";
11     c2.Show();
12
13     Complex c = c1.Add(c2);
14
15     std::cout << " = ";
16     c.Show();
17     std::cout << std::endl;
18
19     return 0;
20 }
```

Inside a class member function, the *this* pointer provides access to the class object through which the member function is invoked. To return **Complex type object**, we simply **dereference the *this* pointer**.

A pointer to const and a const pointer

```
double dval = 2;
const double pi = 3.14; //pi is a const, its value may not be changed
double *ptrd = &dval;    // ok, ptrd is plain pointer, it can point to the double variable
ptrd = &pi; // error, ptrd is a plain pointer, it can not point to the const variable
const double *ptrc = &pi; // ok, ptrc may point to a const double variable
ptrc = &dval; // ok, ptrc can point to the no-const double variable
ptrc is a pointer to const

dval = 3; // ok
*ptrd = 5; // ok
*ptrc = 10; // error, ptrc is a pointer to const, we can not change the value by this pointer
```

cptr is a const pointer

```
double *const cptr = &dval; // cptr is a const pointer, it must be initialized.
// Its value can not be change which means it can
//not be pointed to another variable
cptr = &pi; // error, a const pointer can not be assigned to another variable
const double *const cptrc = &dval; // cptrc is a const pointer, and it points to a const variable

cptrc = &pi; // error, a const pointer can not be assigned to another variable
*cptr = 4; // ok, we can change the value of the variable to which the const pointer points
*cptrc = 9; // error, cptrc is a pointer to const, we can not change the value by this pointer
```

The **this** pointer is a **const pointer**, because **this** is intended to always refer to “this” object. We can not change the address that **this** holds.

When a member function is invoked, **this** pointer is bound to “**this**” object. By default, it is bound to a non-const object. If the member function doesn’t change the object to which **this** points, it should be a pointer to const. The **const** at the end of the member function header is for that purpose. Such **const member functions** are more flexible because **this** pointer can points both a const object and a non-const object. Moreover, whether a member function is const or not will constitute overloading.

A **static** member function has no **this** pointer, so it can not be defined as a const member function.

```

1
2 class Point
3 {
4     private:
5         double x;
6         double y;
7
8     public:
9         Point():x(0),y(0) {}
10        Point(double a, double b): x(a),y(b) {}
11
12        Point& setPoint(double a, double b);
13        void display();
14
15 };

```

```

1 #include <iostream>
2 #include "point.h"
3
4 Point& Point::setPoint(double a, double b)
5 {
6     this->x = a;
7     this->y = b;
8     return *this;
9 }
10
11 void Point::display()
12 {
13     std::cout << "(" << x << "," << y << ")" << std::endl;
14 }

```

```

1 #include <iostream>
2 #include "point.h"
3 using namespace std;
4
5 int main()
6 {
7     Point a;
8     const Point b(1,5);
9
10    a.setPoint(3,6).display();
11
12    b.setPoint(10,20);
13
14    b.display();
15
16    return 0;
17 }

```

For a const object, it is reasonable that the setpoint function can not be called.

For a const object, it is unreasonable that the display function can not be called.

Define the display as a const member function that makes the function more flexible.

2.6 An Array of Objects

You can declare an array of objects the same way you declare an array of any standard types.

Stock mystuff[4]; // creates an array of 4 Stock objects

You can use a constructor to initialize the array elements by calling the constructor for each individual element:

```
const int STKS = 4;  
Stock stocks[STKS] = {  
    Stock("NanoSmart", 12.5, 20),  
    Stock("Boffo Objects", 200, 2.0),  
    Stock("Monolithic Obelisks", 130, 3.25),  
    Stock("Fleep Enterprises", 60, 6.5)  
};
```

If the class has more than one constructor, you can use different constructors for different elements:

```
const int STKS = 10;  
Stock stocks[STKS] = {  
    Stock("NanoSmart", 12.5, 20),  
    Stock(),  
    Stock("Monolithic Obelisks", 130, 3.25),  
};
```

The remaining seven members are initialized using the default constructor.

```

const int STKS = 4;

int main()
{
    //create an array of initialized objects
    Stock stocks[STKS] = {
        Stock("NanoSmart", 12, 20.0),
        Stock("Boffo Objects", 200, 2.0),
        Stock("Monolithic Obelisks", 130, 3.25),
        Stock("Fleep Enterprises", 60, 6.5)
    };

    std::cout << "Stock holdings:\n";
    int st;
    for(st = 0; st < STKS; st++)
        stocks[st].show();

    //set pointer to first element
    const Stock * top = &stocks[0];
    for(st = 1; st < STKS; st++)
        top = &top->topval(stocks[st]);

    //now top points to the most valuable holding
    std::cout << "\nMost valuable holding:\n";
    top->show();

    return 0;
}

```

```

Constructor using NanoSmart called
Constructor using Boffo Objects called
Constructor using Monolithic Obelisks called
Constructor using Fleep Enterprises called

```

Stock holdings:

```

Company: NanoSmart Shares: 12
Share Price: $20.000 Total Worth: $240.00
Company: Boffo Objects Shares: 200
Share Price: $2.000 Total Worth: $400.00
Company: Monolithic Obelisks Shares: 130
Share Price: $3.250 Total Worth: $422.50
Company: Fleep Enterprises Shares: 60
Share Price: $6.500 Total Worth: $390.00

```

Most valuable holding:

```

Company: Monolithic Obelisks Shares: 130
Share Price: $3.250 Total Worth: $422.50

```

```

Bye, Fleep Enterprises!
Bye, Monolithic Obelisks!
Bye, Boffo Objects!
Bye, NanoSmart!

```

The destructor is called in the reverse order that the constructor is called.

3 Exercises

Designs a class named **Rectangle** to represent a rectangle. The class contains:

- **Two double data fields** named **width** and **height** that specify the width and height of the rectangle. The default values are 1 for both width and height.
- **A static data member** named **countOfObject**, which stores the numbers of rectangle objects.
- **A no-arg constructor** that creates a default rectangle.
- **A constructor** that creates a rectangle with the specified width and height.
- **Two getters and two setters.**
- A method named **getArea()** that returns the area of this rectangle.
- A method named **getPerimeter()** that returns the perimeter.
- A method named **display()** that print out the information of rectangle.
- A static method named **getCount()** that returns the number of the rectangle objects.

Write a test program that creates two Rectangle objects, one with the default width and height, and the other with width 4 and height 35.2. Display the width, height, area and perimeter of each rectangle in this order and then the numbers of the rectangle objects.

Using a three-file way, one .h for class declaration and other two .cpps for the member functions' definitions and the test program respectively.

A sample runs might look like this:

```
      Rectangle 1
-----
Width:      1
Height:     1
Perimeter:  4
Area:       1

      Rectangle 2
-----
Width:      4
Height:    35.2
Perimeter:  78.4
Area:     140.8
The numbers of the rectangles are:2
```