类和对象

1. 类

面向对象方法中的类，是对具有相同属性和行为的同一类对象的抽象描述，其内部包括属性（本类的数据成员）和行为（本类的成员函数）两个主要部分，即是说，类以数据为中心，把相关的一批函数组成为一体。

2. 对象

如果将类看做数据类型，那么该类的对象就是相应类型的变量
如果将类看做某类事物的概括，那么该类的对象是类的实例

3. 面向对象程序编程的特点

封装性

- 将同类事物的共同属性封装为一类

继承性

- 从其它事物中继承某些属性

多态性

- 函数、运算符重载
- 虚函数

类的主要组成
成员变量、成员函数

4. 面向对象程序的结构

面向对象程序的结构

•类定义文件（以h为扩展名）

•类的成员函数定义文件（以cpp为扩展名）

•主函数文件（以cpp为扩展名）

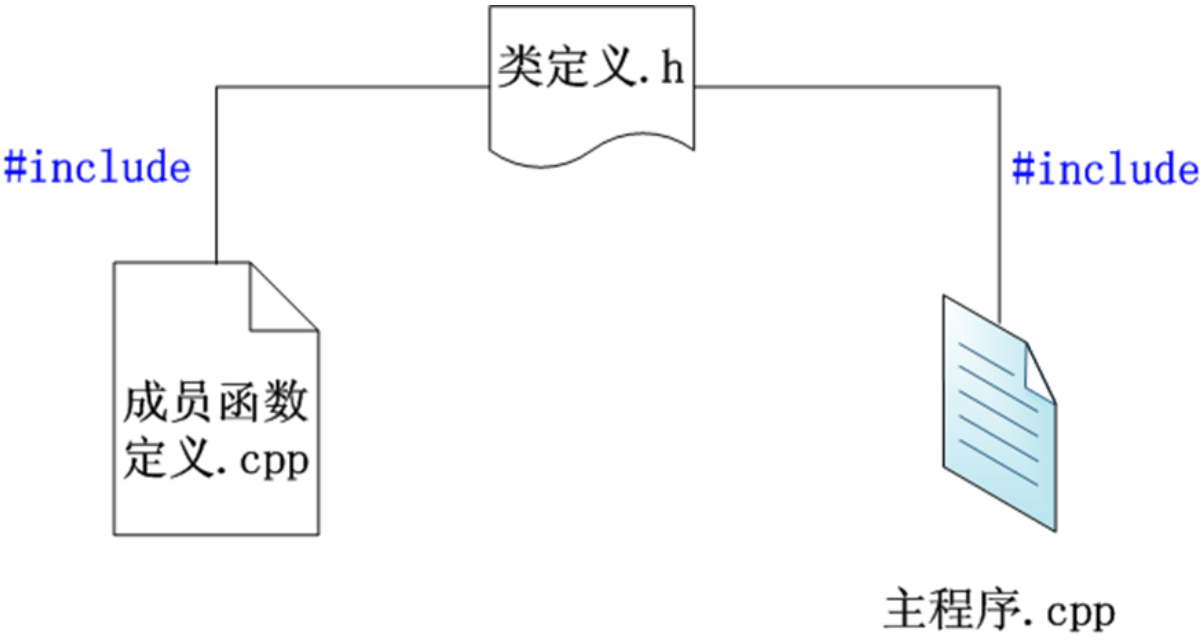


Figure 1

5. 类的成员

5.1 成员变量

普通变量（包括结构类型变量）

数组（包括结构类型数组）

指针（包括结构类型指针变量）

类对象

- 普通对象
- 对象数组
- 对象指针

可以在类成员变量说明时给出默认值，如果说明对象时未对成员变量进行初始化，可以使用默认值

5.2 成员函数

- 即函数成员，对该类对象所含数据进行操作的方法
- 既可放于类定义的花括号之中，也可按类外定义方式放于之外（但要求类体内必须有其函数原型，且类定义外函数说明的前面必须用“<类名>::”来限定）

在类的定义外，定义成员函数的格式

<返回值类型> <类名>::<函数名>(<参数表>)

{<函数体>}

6. 类的封装性

类把数据（事物的属性）和函数（事物的行为——操作）封装为一个整体。

•成员函数可以直接使用类定义中的任何成员，可以处理数据成员，也可调用函数成员。

•类是一种数据类型，定义类时系统不为类分配存储空间，所以不能对类的数据成员初始化。类的数据成员在类定义中通常不能直接初始化，数据成员的初始化需要在对象创建时进行。类中的任何数据成员也不能使用关键字extern限定其存储类型。（因为它们的存储由对象管理，与全局变量的外部链接机制不兼容，**静态数据成员例外，因为它们本质上是全局变量**）
extern 关键字的作用：extern 通常用于声明变量或函数，表示它们在其他地方定义，目的是告诉编译器这个变量或函数的存储空间在其他地方分配，当前只是声明。例如：

```
extern int globalVar//声明全局变量，实际存储在别处
```

Fence 1

为什么数据成员不能用 extern 限定：

- 类的数据成员是对象的组成部分，每个对象都有自己的数据成员副本，存储空间在对象实例化时分配。
- extern 关键字用于全局变量或函数的跨文件链接，而类的数据成员是对象内部的成员，与全局变量的存储方式和生命周期不同。
- 如果允许数据成员使用 extern，会导致语义上的混乱，因为数据成员的存储和生命周期由对象管理，而 extern 意味着外部链接，这与类的封装和对象实例化的机制冲突。
- 另外，extern 通常用于静态存储（全局或静态变量），而类的数据成员（非静态）是每个对象独有的，存储在对象的内存区域中，因此无法使用 extern。

例外：如果是**静态数据成员**，它不属于某个对象，而是属于整个类，存储在全局/静态存储区，因此可以用 extern 声明（但需要在类外部定义）。例如：

```
class MyClass {
public:
    static int x; // 静态数据成员声明
};
int MyClass::x = 10; // 静态数据成员定义
```

Fence 2

7. 类对象的存储

在类说明中定义函数，系统为每一个对象分配了全套的内存。数据区安放成员数据，代码区安放成员函数。

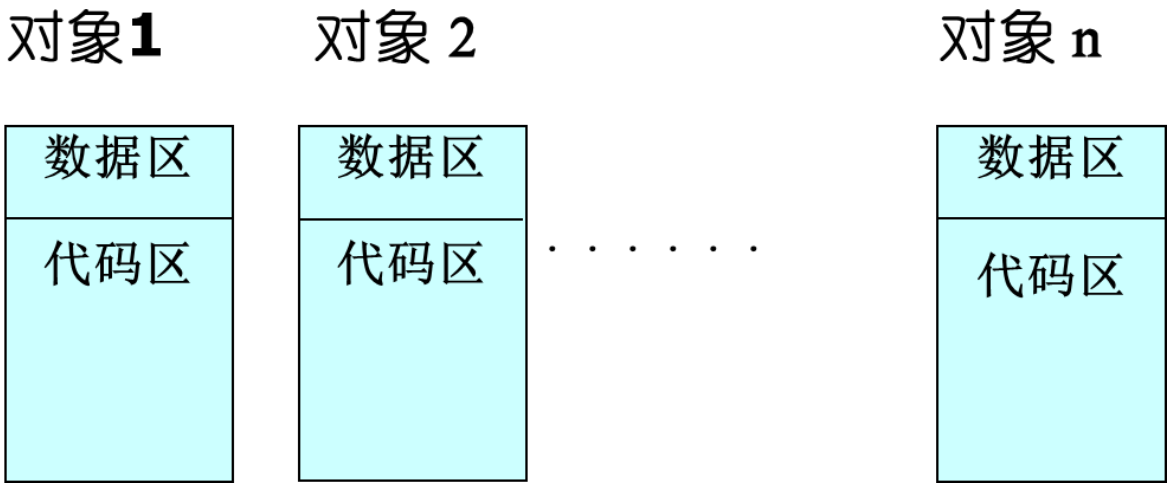
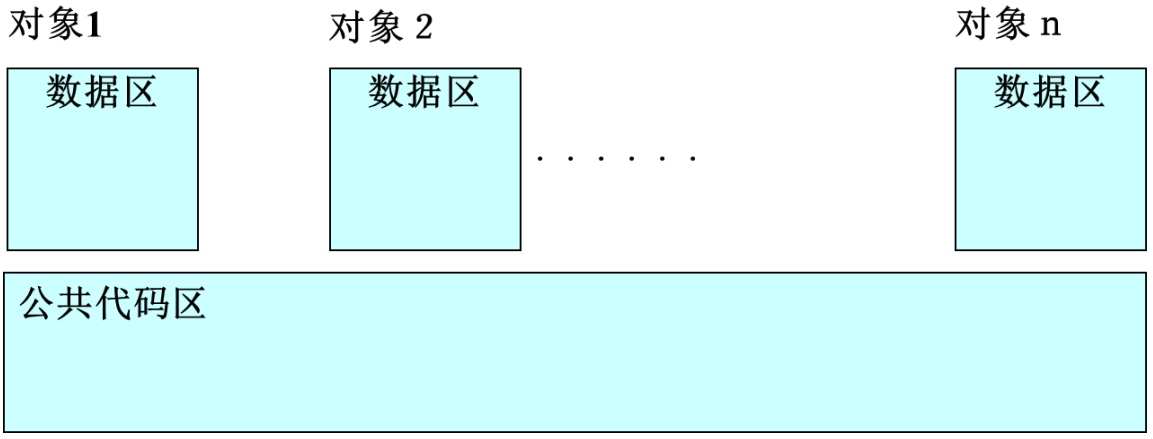


Figure 2

在类说明外部定义函数,为每个对象分配一个数据区，代码区（放成员函数的区域）为各对象类共用。



区别同一个类的各个不同的对象的属性是由数据成员决定的，不同对象的数据成员的内容是不一样的；而行为（操作）是用函数来描述的，这些操作的代码对所有对象都是一样的。

8. 指向成员的指针

说明格式：<类型名> <类名>::*<指针变量名>;eg:

```
s.*p=5;
```

Fence 3

<类型名> (<类名>::*<函数指针名>)(形参表);eg: int A::fun()

•对于已创建的对象，可将其包含的类成员（包括非静态成员变量和成员函数）地址赋给指向成员的指针

```
class X
{
public:
   void f(int);
   int a;
};
void X::f(int x)
{
   a = x;
}
void main()
{
   int X::* pmi = &X::a;
   void (X:: * pmf)(int) = &X::f;
   X objx;
   objx.*pmi = 10;
   cout << objx.a << endl;
   (objx.*pmf)(5);
   cout << objx.a << endl;
}


```

Fence 4

9. 构造函数与析构函数

9.1 构造函数

构造函数在创建对象时被执行

构造函数在**对象创建时**自动执行，用于初始化**非静态成员变量**或执行其他初始化逻辑。

静态成员变量（类变量）在**程序启动时**初始化，与构造函数的执行无关。

用来对类对象进行初始化
函数名与类名相同，无函数返回类型说明（不可以带有返回值）
可有多个构造函数（构造函数可以重载，各自的参数表不相同）
构造函数无任何函数类型
若某个类定义中没有给出一个显示的构造函数的话，则系统自动给出一个
缺省的（隐式的）如下形式的构造函数

<类名>（）{}//此函数无参，且什么事情也不做

对象初始化的方式

9.1.1 直接初始化

采用初始化列表的方式，类似于对数组、结构类型变量进行初始化的方法

9.1.2 使用构造函数

默认构造函数

关于默认构造函数

不在类定义中进行说明，也不给出函数的定义
当类定义中不包含显示的构造函数时，由系统自动给出，并在说明类对象时自动调用

自定义构造函数

必须在类定义内进行说明，在类定义内或外给出函数的定义
函数名和返回值类型与默认构造函数相同
函数可以有参数，也可以无参数

类中包含自定义构造函数时，默认构造函数失效。如果仍想让对象被默认构造，可以采用以下方式

1.显示自定义默认构造函数

```
address (){}
```

Fence 5

2.使用**default**关键字，表示默认构造函数

```
address ()= default;
```

Fence 6

重载构造函数

不同的参数列表

9.1.3 进一步讨论构造函数

使用explicit关键字
C++标准支持带有一个参数的构造函数，将参数类型隐式转换为相应的类类型
去掉隐式转换机制，在构造函数说明时，可以使用explicit关键字explicit关键字用于防止构造函数或类型转换函数的隐式调用。
在构造函数中使用explicit可以避免意外的类型转换，提高代码的可读性和安全性。
建议在单参数构造函数或可能引发隐式转换的场景中使用explicit，除非你明确需要隐式转换。

委托构造函数

•类中包含多个构造函数

•一个构造函数在其初始化列表中调用了另一个构造函数，即将构造工作委托给另一个构造函数

```
class Box
{
double length;
double width;
double height;
public:
Box(double lv,double wv,double hv):length{lv},width{wv}, height{hv}
{}
Box(double side):Box(side,side,side)
{}
};
```

Fence 7

9.2 析构函数

在撤销对象占用的内存前完成一些清理工作
当对象的生存期结束时使用
即**当对象退出其说明区域，或使用delete释放动态对象时，系统自动调用其析构函数**

格式：~<类名>（）{}
无函数返回类型且无参数
可以由用户自定义

一个类只能有一个析构函数，也可以缺省
没有重载的析构函数
若某个类定义中没有用户自定义的析构函数，则系统使用默认的析构函数，形式为~<类名>（）{}，此函数什么也不做

9.3 构造函数与析构函数的执行顺序

先构造的后析构，后构造的先析构

构造函数：基类 → 成员对象（声明顺序） → 派生类。
析构函数：派生类 → 成员对象（声明相反顺序） → 基类。

10. 拷贝构造函数

一种特殊的构造函数，具有一般构造函数的特性，只含有一个形参，且为本类对象的引用
原型为：<类名>（<类名>&）
作用是使用一个已存在的对象去初始化另一个正在创建的对象
类对象间的一般赋值，由拷贝构造函数实现（通过深拷贝或浅拷贝的方式）

10.1 浅拷贝

系统会自动生成缺省的拷贝构造函数，实现对象间的拷贝（浅拷贝）

什么时候会自动调用拷贝构造函数？（三类情况）
1.使用以下形式的说明语句
<类名><对象名2>（<对象名1>）；
<类名><对象名2>=<对象名1>;
2.对象作为函数的赋值参数
3.函数的返回值为类的对象

赋值语句并不自动调用

10.2 深拷贝（显示拷贝构造函数）

在某些情况下，必须在类定义中给出显示拷贝构造函数，以实现用户指定的拷贝功能（深拷贝）

假设在某类的普通构造函数中分配并使用了某些系统资源，而且在该类的析构函数中释放了这些资源。如果执行"浅拷贝"，**使两个对象使用相同的系统资源**，调用析构函数将会两次释放相同的资源而导致错误。

给出显式的拷贝构造函数，可以在实现拷贝的过程中，**为"被拷贝"的对象分配新的系统资源，避免了重复释放资源的错误**

11. 常对象与常量成员

11.1 常对象

说明格式：const<类名><类对象名>(<实参表>);
构成常对象的任何成员变量（不包括由mutable修饰的）都不能被修改，但是可以读取成员变量值

常对象不能够调用任何成员函数（成员函数中能够访问成员变量，有可能间接改变成员变量的值）

11.2 类的常量成员

由关键字const修饰的类成员说明称为类的常量成员
常量数据成员、常量函数成员

11.2.1 常量数据成员

类的常量数据成员必须进行初始化，而且**只能通过构造函数的成员初始化列表**的方式来进行
在对象被创建以后，其常量数据成员的值就不允许被修改（只可读取其值，但不可进行改变）
一个类的常量数据成员并非由该类的所有对象共享，而是对于该类的不同对象可以取不同的值

11.2.2 常量函数成员

常量函数成员只有权读取相应对象的内容，但无权修改它们。
说明格式：<类型说明符><函数名>(<参数表>)const;

定义格式：<类型说明符><函数名>(<参数表>)const{<函数体>}
常对象可以调用常量函数成员（并非只能使用常量数据成员）

12. 静态成员

12.1 类的静态成员：

由关键字**static**修饰的类成员说明称为类的静态成员
包括：静态数据成员、静态函数成员
类的静态成员为其所有对象所共享，不管有多少对象，静态成员都只有一份存于公用内存中

12.2 静态数据成员

类的静态数据成员为该类的所有对象所共享。
在类定义中对静态数据成员进行说明
必须在**类外文件作用域**中的某个地方对静态数据成员赋初值（按以下格式）
<类型><类名>:<静态数据成员>=<初值>;
访问静态数据成员有三种方式
1.<类名>::<静态数据成员名>
2.<对象名>.<静态数据成员名>
3.<对象指针>-><静态数据成员名>

12.3 静态函数成员

类的静态函数**没有this指针**（非静态成员函数拥有this指针），从而无法处理不同的调用者对象的各自数据成员值。通常情况下，类的静态函数只处理类的静态数据成员。
静态成员函数不依赖类的对象实例，可以直接通过类名调用。非静态成员函数需要通过对象来调用。
故可通过类名::函数名（）来调用函数的话，说明该函数是类的公有静态成员函数
对类的静态成员的访问通常为
<**类名**>::<静态函数成员调用>
也可以为
<对象名>.<静态函数成员调用>
<对象指针>-><静态函数成员调用>

13. 友元

概念：

一种类成员访问权限

- 能够访问类的任何数据成员

在类的定义中，以关键字friend标识

- 出现在函数说明语句前，表示该函数为类的友元函数（一个函数可以同时说明为多个类的友元函数）
- 出现在类名之前，表示该类为类的友元类

13.1 友元函数

将普通函数说明为某类的友元函数

- 该函数定义在类外
- 说明格式为
  friend+返回值类型+函数名——函数表

```
friend void Func();
```

Fence 8

将其他类的成员函数说明为某类的友元函数

- 前提是已有其他类的定义
- 说明格式为
  friend+返回值类型+类名：：函数名+参数表

```
friend void B::Func();
```

Fence 9

前提是对B已经进行了定义或说明

友元函数**不是**类的成员函数，在函数体中访问对象的成员，必须用对象名加运算符“.”加对象成员名。但友元函数可以访问类中的**所有成员**，一般函数只能访问类中的公有成员。

友元函数不受类中的访问关键字限制，可以把它放在类的公有、私有、保护位置，但结果一样

某类的友元函数的作用域并非该类域。如果该友元函数是另一类的成员函数，则其作用域为另一类的类域，否则与一般函数相同。

作用

提高程序的运行效率（并不能加强类的封装性和实现数据的隐藏性，反而降低了）

13.2 友元类

将一个类B说明为另一个类A的友元类，类B中的所有函数都是A的友元函数，可以访问类A中的所有成员。

说明方式：
friend+类名；

友元类的关系是**单向的**；
友元类的**关系不能传递**：

除非确有必要，一般不把整个类说明为友元类，而把成员函数说明为友元函数

友元的概念**破坏了**类的封装性，但有助于数据共享，能够提高程序的效率

14. 类与类之间的关系

C++语言为类和对象之间的联系提供了许多方式，主要有：

•一个类的对象作为另一个类的成员

•一个类的成员函数作为另一个类的友元

•一个类定义在另一个类的说明中，即类的嵌套

•一个类作为另一个类的派生类

14.1 包含对象成员的类对象构造

在定义（生成）一个含有对象成员的类对象时，它的构造函数被系统调用，这时将首先按照初始化符表来依次执行各对象成员的构造函数，完成各对象成员的初始化工作，而后执行本类的构造函数体。析构函数的调用顺序恰好与之相反。

即：**构造函数：基类 → 成员对象（声明顺序） → 派生类。**
析构函数：派生类 → 成员对象（声明相反顺序） → 基类。

如果初始化符表中没有对象成员的显式初始化，则调用无参构造函数初始化类对象成员

14.2 类的嵌套

分为公有嵌套类和私有嵌套类
使用时CC::C2 a,b;
（CC为原本的类，C2为公有嵌套类，私有嵌套类用不了，公有嵌套类的使用也必须要加定义域限制）

使用并不方便，不宜多用

15. 类中的运算符重载

允许以下两种方式来定义运算符重载函数

- 以类的**友元函数**定义

  所有运算分量必须显式地列在本友元函数的参数表中，而且这些参数类型中至少要有一个应该是说明该友元的类类型或是对该类的引用

- 以类的**公有成员函数**方式定义

  总以当前调用者对象(*this)作为该成员函数的隐式第一运算分量，若所定义的运算多于一个运算对象时，才将其余运算对象显式地列在该成员函数的参数表中

对重载运算符的限制：

- 被用户重载的运算符，其**优先级、结合性、以及运算分量个数**都必须与系统中的本原运算符相一致。**不可以改变语法结构**
- 如下5个运算符不可重载：

  . :: ?: .* sizeof

- 只能以类成员而不能以友元身份重载的运算符:

  = ( ) [ ] ->

- 不可自创新的运算符。

16. 简单的数据结构设计

16.1 链表（List）

16.2 栈（Stack）

16.3 队列（Queue）

17. 附录

在C++中，explicit关键字用于修饰类的构造函数或类型转换函数，防止隐式类型转换或意外的构造函数调用。它确保只有显 式的类型转换或构造调用才被允许，从而提高代码的安全性和可读性。

17.1 explicit关键字的作用

1. **防止隐式类型转换**： 当构造函数被声明为explicit时，它不能被用于隐式类型转换。也就是说，编译器不会自动将某种类型转换为该类的对象，除非程序员明确指定。
2. **提高代码清晰度**： 使用explicit可以明确表达程序员的意图，避免因隐式转换导致的意外行为或难以调试的错误。
3. **只适用于构造函数和类型转换函数**： explicit通常用于单参数构造函数（或带有默认参数的多参数构造函数），以及operator类型转换函数。

17.2 explicit在构造函数中的具体作用

当一个构造函数被标记为explicit时，它只能通过显式的构造调用，而不能用于隐式转换。例如：

17.3 示例1：无explicit的构造函数（允许隐式转换）

```
#include <iostream>
class MyClass {
public:
    MyClass(int x) : value(x) {} // 没有explicit
    void display() const { std::cout << value << std::endl; }
private:
    int value;
};

void func(MyClass obj) {
    obj.display();
}

int main() {
    MyClass obj = 42; // 隐式转换：int -> MyClass
    func(42);         // 隐式转换：int -> MyClass
    return 0;
}
```

Fence 10

输出 42 42

在上面的代码中，MyClass的构造函数允许将int类型隐式转换为MyClass对象。这种隐式转换可能导致代码行为不符合预期，尤其是在大型项目中。

17.4 示例2：使用explicit构造函数（禁止隐式转换）

```
#include <iostream>
class MyClass {
public:
    explicit MyClass(int x) : value(x) {} // 使用explicit
    void display() const { std::cout << value << std::endl; }
private:
    int value;
};

void func(MyClass obj) {
    obj.display();
}

int main() {
    // MyClass obj = 42; // 错误：explicit构造函数禁止隐式转换
    MyClass obj(42);     // 正确：显式构造
    // func(42);         // 错误：不能隐式转换int到MyClass
    func(MyClass(42));   // 正确：显式构造
    return 0;
}
```

Fence 11

输出 42

在上述代码中，explicit修饰的构造函数禁止了从int到MyClass的隐式转换，必须显式地调用构造函数。

17.5 explicit的适用场景

1. **单参数构造函数**：如果一个类有单参数构造函数（或多参数构造函数中除第一个参数外其他参数有默认值），通常建议使用explicit，除非你明确希望允许隐式转换。
2. **类型转换函数**：explicit也可以用于类型转换函数，例如：

```
class MyClass {
public:
    explicit operator int() const { return value; } // 显式类型转换
private:
    int value = 42;
};

int main() {
    MyClass obj;
    // int x = obj;         // 错误：需要显式转换
    int x = static_cast<int>(obj); // 正确：显式转换
    return 0;
}
```

Fence 12

17.6 注意事项

1. C++11及以后
    - 在C++11之前，explicit仅用于构造函数。
    - C++11开始，explicit也可以用于类型转换操作符（如operator int()）。
    - C++11还引入了explicit与constexpr的结合，用于编译期构造。
2. 默认构造函数和多参数构造函数
    - 如果构造函数有多个参数且无默认值，隐式转换通常不会发生，因此explicit在这种情况下影响较小，但仍可提高代码意图的清晰度。
3. **复制构造函数**： 复制构造函数通常不需要explicit，因为它们本身不会引发隐式转换问题。

17.7 总结

- explicit关键字用于防止构造函数或类型转换函数的隐式调用。
- 在构造函数中使用explicit可以避免意外的类型转换，提高代码的可读性和安全性。
- 建议在单参数构造函数或可能引发隐式转换的场景中使用explicit，除非你明确需要隐式转换。