中國人民大學
RENMIN UNIVERSITY OF CHINA

面向对象编程

授课教师：孙亚辉

教学团队

- 教师：孙亚辉
  - 邮箱：yahuisun@ruc.edu.cn
- 助教：冯逍遥

	章节	章节名称	教学内容与教学要求
教学进度安排	第 1 章	类和对象	理论与实践课内容： 1. C++中的类与对象 2. 类的成员函数
	第 2 章	函数进阶	理论与实践课内容： 1. 函数重载 2. 操作符重载 3. 头等函数
	第 3 章	数据的共享与保护	理论与实践课内容： 1. 友元 2. 共享数据的保护 3. 命名空间
	第 4 章	继承	理论与实践课内容： 1. 继承 2. 派生类构造函数 3. 继承与成员函数 4. 继承与成员变量
	第 5 章	（选讲）多态	理论与实践课内容： 1. 虚函数 2. 虚函数表 3. 抽象类 4. 虚析构函数
	第 6 章	字符串与输入输出	理论与实践课内容： 1. 字符串 2. 输入与输出
	第 7 章	模板	理论与实践课内容： 1. 函数模板 2. 类模板
	第 8 章	标准模板库	理论与实践课内容： 1. 容器 2. 迭代器 3. 算法

面向对象程序设计的基本概念——对象（object）

- 一般意义上的对象：现实世界中一个实际存在的（有形或无形的）事物，是构成世界的一个独立单位。具有静态特征和动态特征。
  - **静态特征**：可以用某种数据来描述
  - **动态特征**：对象所表现的行为或具有的功能

- 面向对象方法中的对象：是系统中用来描述客观事物的一个实体，它是构成系统的一个基本单位。对象由一组属性和一组行为构成。
  - **属性**：用来描述对象静态特征的数据项
  - **行为**：用来描述对象动态特征的操作序列

C语言中的结构体struct可以描述一个对象吗？

面向对象程序设计的基本概念——类（class）

- 分类是人类通常的思维方法，依据的原则是**抽象**
  - 注意那些与当前目标有关的本质特征，从而找出事物的共性，把具有共同性质的事物划分为一类
  - 例如，石头、树木、汽车、房屋等都是人们在长期的生产和生活实践中抽象出的概念

- 面向对象方法中的"类"
  - 具有相同属性和行为的一组对象的集合
  - 一个类为属于该类的全部对象提供了抽象的描述，包括属性和行为两个主要部分
  - 类与对象的关系：犹如模具与铸件之间的关系，一个属于某类的对象称为该类的一个实例

面向过程 VS 面向对象

面向过程（Procedure Oriented）：把事情拆分成几个步骤，然后按照一定的顺序执行。

面向对象（Object Oriented）：把事物抽象成对象的概念，然后给对象赋属性和方法，再让每个对象去执行自己的方法。

举例：用洗衣机洗衣服。

面向过程：

- 放衣服（方法）-->加洗衣粉（方法）--> 加水（方法）--> 漂洗（方法）--> 清洗（方法）--> 甩干（方法）

面向对象：

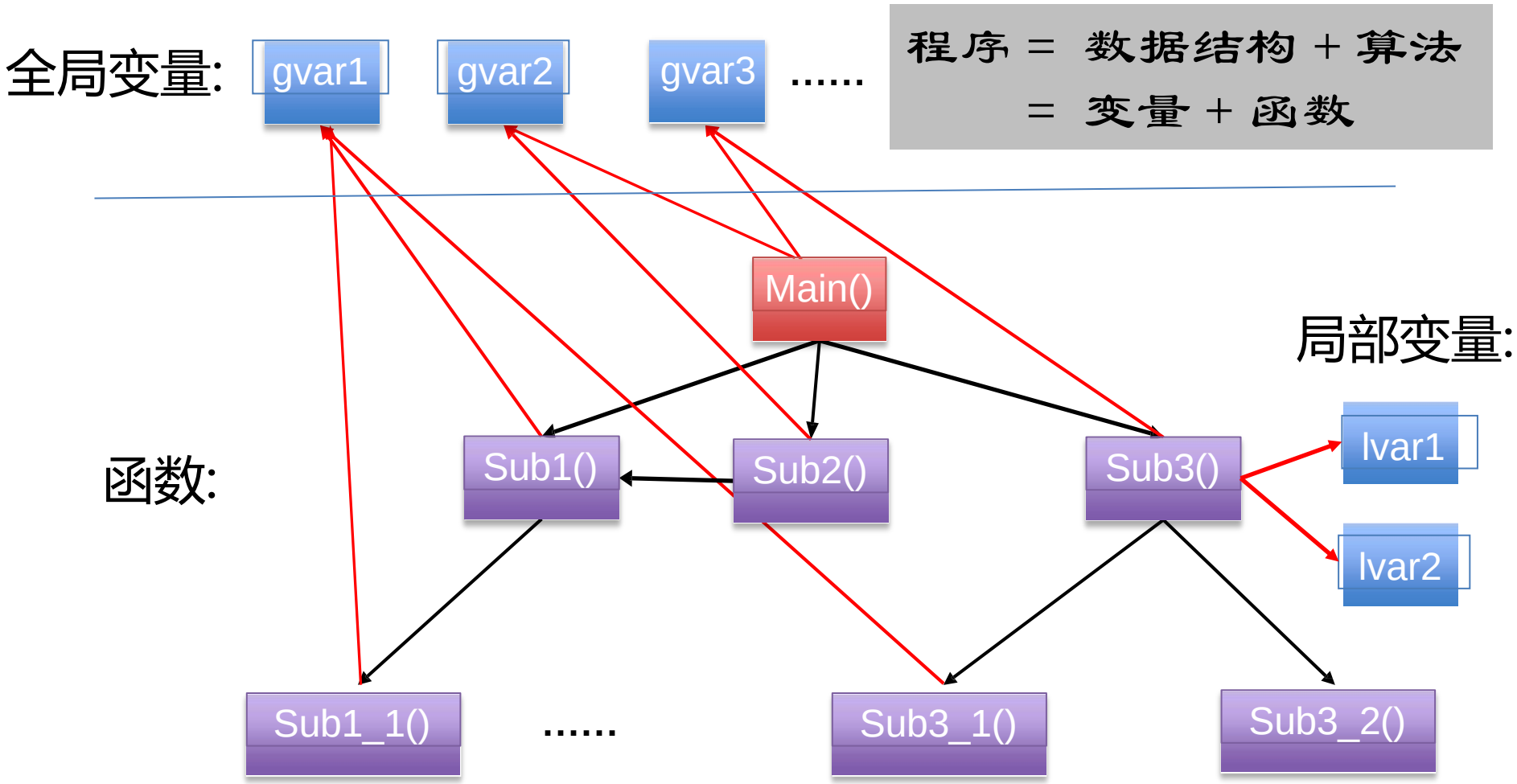
- 构造出两个对象：人和洗衣机
- 人加入属性和方法：放衣服（方法）、加洗衣粉（方法）、加水（方法）、人的各种属性。
- 洗衣机加入属性和方法：漂洗（方法）、清洗（方法）、甩干（方法）、洗衣机的各种属性。
- 人.放衣服（方法）-> 人.加洗衣粉（方法）-> 人.加水（方法）-> 洗衣机.漂洗（方法）-> 洗衣机.清洗（方法）-> 洗衣机.甩干（方法）

面向过程的程序设计方法

- 设计思路
  - 自顶向下、逐步求精（细化）
  - 采用模块分解与功能抽象，分而治之

- 程序结构
  - 按功能划分为若干个基本模块
  - 各模块间的关系尽可能简单，功能上相对独立
  - 模块内部由顺序、选择和循环三种基本结构组成
  - 其模块化实现的具体方法是使用子程序

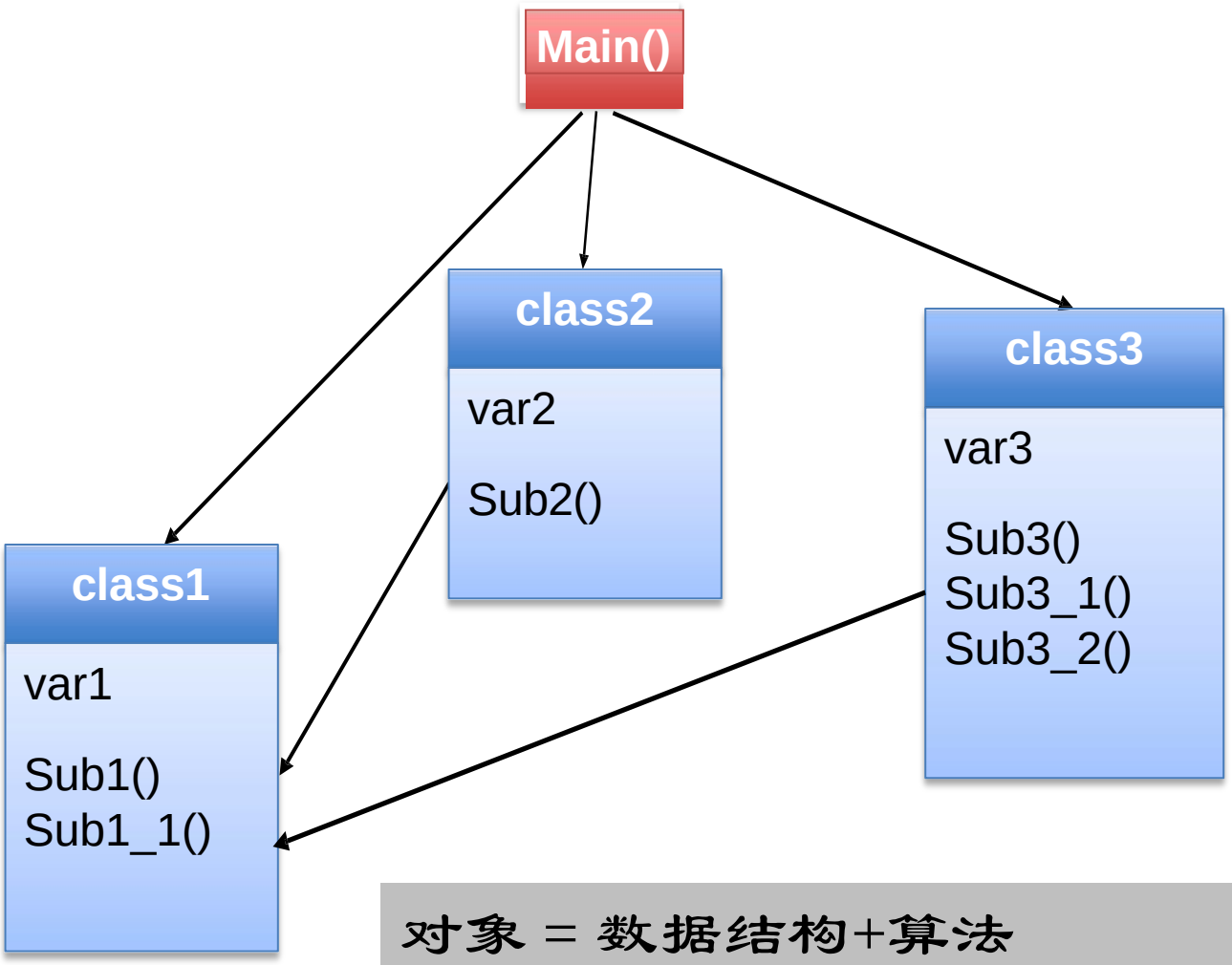
面向过程的程序模式


全局变量:
gvar1
gvar2
gvar3
.....
程序 = 数据结构 + 算法
= 变量 + 函数
Main()
局部变量:
函数:
Sub1()
Sub2()
Sub3()
lvar1
lvar2
Sub1_1()
.....
Sub3_1()
Sub3_2()

面向对象的程序设计方法

- 将数据及对数据的操作方法封装在一起，作为一个相互依存、不可分离的整体——对象
- 对同类型对象抽象出其共性，形成类
- 类通过一个简单的外部接口，与外界发生关系
- 对象与对象之间通过消息进行通信

面向对象的程序模式



对象 = 数据结构+算法

程序 = （对象+对象+...+对象）+ 消息

面向过程 VS 面向对象

面向过程

- 优点：有效地将一个较复杂的程序系统设计任务分解成许多易于控制和处理的子任务，便于开发和维护。
- 缺点：可重用性差、数据安全性差、难以开发大型软件和图形界面的应用软件，因为
  - 把数据和处理数据的过程分离为相互独立的实体。
  - 当数据结构改变时，所有相关的处理过程都要进行相应的修改。
  - 图形用户界面的应用程序，很难用过程来描述和实现，开发和维护也都很困难。

面向对象

- 优点：
  - 程序模块间的关系更为简单，程序模块的独立性、数据的安全性就有了良好的保障。
  - 通过继承与多态性，可以大大提高程序的可重用性，使得软件的开发和维护都更为方便
- 缺点：性能比面向过程低

面向对象程序设计的三大特性

- 封装性
- 继承性
- 多态性

面向对象程序设计的三大特性——封装性

- 定义：把对象的属性和行为组成一个独立的系统单元
- 例如：将与教师相关的属性（姓名、年龄、职称等）和行为（开课、立项）封装成一个Teacher类
- 意义：尽可能隐蔽对象的内部细节，对外形成一个边界，只保留有限的对外接口使之与外部发生联系

面向对象程序设计的三大特性——继承性

- 定义：特殊类的对象拥有其一般类的全部属性与行为，称作特殊类对一般类的继承。
- 例如：将轮船作为一个一般类，客轮便是一个特殊类
- 意义：实现软件复用，是面向对象技术能够提高软件开发效率的重要原因之一

面向对象程序设计的三大特性——多态性

- 定义：在一般类中定义的属性或行为，被特殊类继承之后，可以具有不同的数据类型或表现出不同的行为
- 例如：数的加法 -> 实数的加法
  -> 复数的加法
- 意义：使得同一个属性或行为在一般类及其各个特殊类中具有不同的语义

面向对象程序设计的三大特性：

- 封装性
- 继承性
- 多态性

从语言方面出发：

- C语言：完全面向过程。
- C++语言：一半面向过程、一半面向对象。
- Java语言：完全面向对象。

C++程序示例

test.cpp

```cpp
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello!" << endl;
    cout << "Welcome to c++!" << endl;
    return 0;
}
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
Hello!
Welcome to c++!
```

1. 标准C++规定main函数必须声明为int类型，如果程序正常运行，向操作系统返回一个零值，否则返回非零值，通常是-1。
2. C++程序中可以用/*…*/做注释，可以用//做注释。前者可以做多行注释，后者只做单行注释。
3. C++程序中常用cout、cin进行输出输入，cout是C++定义的输出流对象，<<是插入运算符。
4. 使用cout、cin需要用头文件iostream，在程序开始要用#include声明包含的头文件。
5. using namespace std; 意思是使用命名空间。C++标准库中的类和函数是在命名空间std中声明的，因此程序中如用C++标准库中的有关内容（此时需要用#include命令行），就要用using namespace std; 语句声明

中國人民大學
RENMIN UNIVERSITY OF CHINA

1 类和对象

授课教师：孙亚辉

本章内容

1. C++中的类
2. 成员函数
3. 类的静态成员
4. 构造函数
5. 析构函数
6. 类的组合
7. 结构体、联合体、位域

1. C++中的类

- **封装**的基本单元，将数据和函数封装在一起

```
class  Clock {
  public:
   void setTime(int newH, int newM, int newS);//成员函数
  protected:
   void showTime(); //成员函数
  private:
   int hour, minute, second; //成员数据
};
```

类的基本概念

- 结构：class 类名 {};
- 类名：通常首字母大写，名词
- 成员函数：同函数的声明
- 成员数据：同变量的声明
- 存储控制：public, private, protected

```
class 类名 {
    public:
      公有成员（外部接口）
    protected:
      保护型成员
    private:
      私有成员
};
```

```
class Clock
{
public:
    void setTime(int newH, int newM, int newS);
protected:
    void showTime()
    {
        cout << "hour= " << hour << endl;
    }
private:
    int hour, minute, second;
};
```

类的存储控制

- public: 可以被任意实体访问
  - 在关键字public后面声明，它们是类与外部的接口
  - 任何外部函数都可以访问公有类型数据和函数
- private: 只允许本类的成员函数访问
  - 在关键字private后面声明，只允许本类中的函数访问
  - 类外部的任何函数都不能访问
  - 类的所有成员的默认访问权限是private
- protected: 只允许本类及子类的成员函数访问
  - 与private类似
  - 其差别表现在继承与派生时对派生类的影响不同

对象

- 类的对象是该类的某一特定实体，即该类型的变量
- 声明形式：①类名 对象名； ②类名 *对象指针名；
- 类中成员互访：直接使用成员名
- 类外访问public成员： ①“对象名.成员名” ②“对象指针名->成员名”

```
class  Clock {
  public:
   void setTime(int newH, int newM, int newS) {
     hour = newH, minute = newM, second = newS;
   }
  protected:
    void showTime();
  private:
   int hour, minute, second;
};
```

```
int main() {
  Clock myClock1;
  myClock1.setTime(10, 0, 0);

  Clock *myClock2 = new Clock();
  myClock2->setTime(11, 30, 0);
}
```

类的用途

- 类是具有相同属性和行为的一组对象的集合，它为属于该类的全部对象提供了统一抽象描述，其内部包括属性和行为两个主要部分
- 利用类可实现数据的封装、隐藏、继承与派生
- 利用类易于编写大型复杂程序，其模块化程度比C中采用函数更高

前向引用声明

- 类应先声明，后使用。若需在某个类的声明之前引用该类，则应进行前向引用声明
- 前向引用声明只为程序引入一个标识符，但具体声明在其他地方

```
class B;  //前向引用声明

class A {
public:
  void f(B b);
};

class B {
public:
  void g(A a);
};
```

前向引用声明

- 注意：在提供一个完整的类定义之前，不能声明该类的对象，也不能在内联成员函数中使用该类的对象

```
class B; //前向引用声明

class A {
   B x; //错误：类B的定义尚不完整
};

class B {
   A y; //正确，类A在类B之前已定义
};
```

应该记住：当你使用前向引用声明时，你只能使用被声明的符号，不能涉及类的任何细节

```
class B;  //前向引用声明

class A {
public:
  void method() {
    x->f();  //错误：类B的对象在定义之前被使用
  }
 private:
  B *x;  //正确，经过前向引用声明，可以声明类B的对象指针
};

class B {
public:
  void f();
private:
  A y;  //正确，类A在类B之前定义
};
```

test.cpp

C: > Users > Yahui > Documents > Drive > VM >

```cpp
#include <stdio.h>

class B; //前向引用声明

class A {
    B x; //错误：类B的定义尚不完整
};

class B {
    A y; //正确，类A在类B之前已定义
};


int main()
{
}
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp:6:6: error: field ‘x’ has incomplete type ‘B’
    6 |     B x; //错误：类B的定义尚不完整
      |       ^
test.cpp:3:7: note: forward declaration of ‘class B’
    3 | class B; //前向引用声明
      |       ^
```

?

test.cpp

C: > Users > Yahui > Documents > Drive > VM > test.cpp > ...

```cpp
#include <stdio.h>

class B; //前向引用声明

class A {
public:
  void method() {
    x->f();  //错误：类B的对象在定义之前被使用
  }
 private:
  B *x;  //正确，经过前向引用声明，可以声明类B的对象指针
};

class B {
public:
  void f();
private:
  A y;  //正确，类A在类B之前定义
};

int main()
{
}
```

?

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp: In member function ‘void A::method()’:
test.cpp:8:6: error: invalid use of incomplete type ‘class B’
    8 |     x->f();  //错误：类B的对象在定义之前被使用
      |      ^~
test.cpp:3:7: note: forward declaration of ‘class B’
    3 | class B; //前向引用声明
      |       ^
```

本章内容

1. C++中的类
2. 成员函数
3. 类的静态成员
4. 构造函数
5. 析构函数
6. 类的组合
7. 结构体、联合体、位域

2. 成员函数

- 在类中说明成员函数原型，可以直接在类中给出函数体，也可以在类外给出函数体实现
- 定义在类中（成员函数前面不加类名和::）
  - 一般规模较小
- 定义在类之后（成员函数前面需加类名和::）
  - 对于较大的成员函数，通常放在类之后，其他的文件中定义
  - 将类的定义（在.h文件中）和成员函数（在.cpp文件中）定义分开
  - 类的定义看作是类的外部接口，成员函数的定义看作是类的内部实现
  - C++中，类的定义通常在头文件中，因此通常将函数声明放在头文件中，而将函数的定义放在其他文件中。

成员函数示例

```
//File: Clock.h

class Clock{
public:
  void setTime(int newH, int newM, int newS);
  void showTime();
private:
  int hour, minute, second;
};
```

注：函数showTime() 全名是 Clock::showTime()

- :: 叫作用域操作符
- 类名的用处是指出showTime()是类Clock的一个成员函数
- 没有类名的函数称为非成员函数

```
//File: Clock.cpp

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

void Clock::setTime(int newH, int newM, int newS) {
   hour = newH;
   minute = newM;
   second = newS;
}

void Clock::showTime() {
   cout << hour << ":" << minute << ":" << second;
}

int main() {
    Clock myClock;
    myClock.setTime(8, 30, 30);
    myClock.showTime();
    return 0;
}
```

内联成员函数

为了提高运行时的效率，对于较简单的函数可以把函数定义在类的声明内部，即为内联成员函数（隐式定义）。

```
class Clock{
public:
   void setTime(int newH = 0, int newM = 0, int newS = 0);
   void showTime() { cout << hour << ":" << minute << ":" << second; }
private:
   int hour, minute, second;
};
```

内联成员函数

- 为了提高运行时的效率，对于较复杂的函数可以使用inline关键字声明其为内联形式（显式定义；编译优化会把函数的代码替换到程序中，提高代码效率）。

```
class Clock{
public:
   void setTime(int newH = 0, int newM = 0, int newS = 0);
   inline void showTime();
private:
   int hour, minute, second;
};

inline void Clock::showTime() { cout << hour << ":" << minute << ":" << second; }
```

```cpp
#include <iostream>
#include <chrono>
using namespace std;
```

内联成员函数的更高效率

```cpp
class A
{
public:
    void method();
};
void A::method()
{
    int i = 0;
    int j = i + 1;
}
```

```cpp
class B
{
public:
    inline void method();
};
void B::method()
{
    int i = 0;
    int j = i + 1;
}
```

```cpp
void try1(int times)
{
    A a;
    auto begin = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < times; i++)
    {
        a.method();
    }
    auto end = std::chrono::high_resolution_clock::now();
    double runningtime = std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin).count() / 1e6;
    cout << "runningtime: " << runningtime << "ms" << endl;
}

void try2(int times)
{
    B b;
    auto begin = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < times; i++)
    {
        b.method();
    }
    auto end = std::chrono::high_resolution_clock::now();
    double runningtime = std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin).count() / 1e6;
    cout << "runningtime: " << runningtime << "ms" << endl;
}
```

内联成员函数的更高效率

```
53  int main()
54  {
55      int times = 1e7;
56      try1(times);
57      try2(times);
58  }
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
runningtime: 42.6865ms
runningtime: 43.9386ms
yahui@Yahui:/media/sf_VM$ g++ -O3 test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
runningtime: 7.4e-05ms
runningtime: 3.6e-05ms
```

-O3是指优化编译（执行代码的效率更高）

调用成员函数

- 必须指明对象和成员名

```
//File: Clock.h

class Clock{
public:
  void setTime(int newH, int newM, int newS);
  void showTime();
private:
  int hour, minute, second;
};
```

```
#include “Clock.h”

void func(){
  showTime(); //错误！showTime() 是哪个类的哪个对象的成员？
  Clock::showTime();//错误！应调用具体某个对象的成员，而不是类
}

void func1(){
  Clock myClock; //声明一个对象，属于Clock类
  myClock.showTime(); //正确，调用myClock对象的showTime
}
```

调用成员函数

- 对象可以用指针来引导

```
#include "Clock.h"

void func(Clock * ps){
  ps ->showTime();
}

void main(){
  Clock myClock; //声明一个对象，属于Clock类
  func(&myClock);//将对象的地址传递给函数的参数
}
```

隐含的形参：this指针

- 假设一个Clock类的对象c，在执行c.showTime()时，成员函数实际也接收到了一个对象c的地址，并将这个地址赋值给隐含的形参this。所有对成员的访问都隐含地加上了前缀this->

```
class Clock{
public:
   void showTime() { cout << hour << ":" << minute << ":" << second; }
//等价于：cout << this->hour << ":" << this->minute << ":" << this->second;
private:
   ......
};
```

隐含的形参：this指针

- 一个对象的this指针并不是对象本身的一部分，不会影响sizeof(对象)的结果。
- this作用域是在类内部，当在类的非静态成员函数中访问类的非静态成员的时候，编译器会自动将对象本身的地址作为一个隐含参数传递给函数。也就是说，即使没有写上this指针，编译器在编译的时候也是加上this的，它作为非静态成员函数的隐含形参，对各成员的访问均通过this进行。

类的大小

- 与类大小有关的因素：普通成员变量，虚函数，继承（单一继承，多重继承，重复继承，虚继承）
- 与类大小无关的因素：普通成员函数、静态成员变量及静态成员函数
- 空类的大小为1字节，因为空类可以实例化，实例化必然在内存中占有一个位置，因此，编译器为其优化为一个字节大小。
- 类的大小（注意成员变量的内存对齐）

类的大小实例

```
class EmptyBox{
};

class Box1{
public:
  char a;
};
```

```
class Box2{
public:
  int b;
};
class Box3{
public:
    double c;
};
```

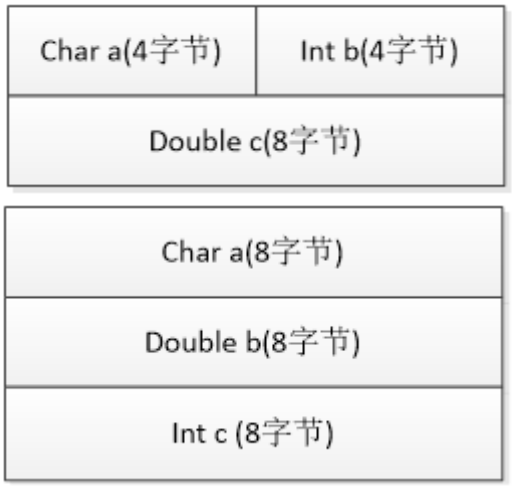
```
class Box4{
public:
  char a;
  int b;
  double c;
};
```

```
class Box5
{
public:
    char a;
    double b;
    int c;
};
```

```
int main() {
  cout << sizeof(EmptyBox) << " ";
  cout << sizeof(Box1) << " ";
  cout << sizeof(Box2) << " ";
  cout << sizeof(Box3) << " ";
  cout << sizeof(Box4) << " ";
  cout << sizeof(Box5) << " ";
}
```

运行结果：

1 1 4 8 16 24



本章内容

1. C++中的类
2. 成员函数
3. 类的静态成员
4. 构造函数
5. 析构函数
6. 类的组合
7. 结构体、联合体、位域

3. 类的静态成员

- 类的成员可以声明为static
- 对于同一个类的所有对象实例来说，静态成员是共享的
- 静态成员独立于单个类对象，无需实例化对象即可进行访问（与普通成员不同），当然也可以通过对象实例进行访问

类的静态成员属性

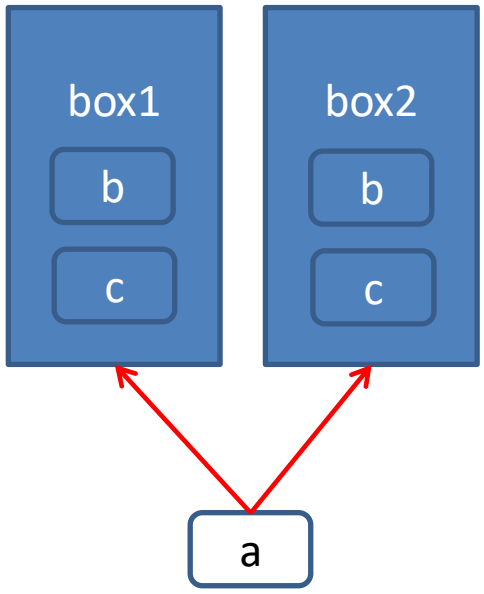
- 将一个成员属性声明为静态，则其存储在独立于类和类的对象实例的区域。
- 即使没有创建对象实例，静态成员属性也依然存在。
- 对象包含普通成员变量的独立副本，而静态成员变量则在所有对象之间共享。

```
class Box{
private:
  static int a;
  int b;
  int c;
public:
  Box(int bb, int cc) :
    b(bb), c(cc) {}
};
```

```
int main() {
  Box box1(1,1), box2(1,1);
  cout << sizeof(Box) << " ";
  cout << sizeof(box1) << " ";
  cout << sizeof(box2) << " ";
}
```

运行结果如下：

8 8 8



类的静态成员的初始化

- 类的普通静态成员数据不能在类体内进行初始化，需要在类外面进行初始化；但类的**常（const）**静态成员数据可以在类体内进行初始化

```cpp
#include <iostream>
#include <cstring>
#include <vector>
using namespace std;

class Exception
{
public:
  static int x = 1;
};

int main()
{
  Exception a;
  a.x;
  return 0;
}
```

```cpp
#include <iostream>
using namespace std;

class Exception
{
public:
  static int x;
};
int Exception::x=1; // 类外初始化

int main()
{
  Exception a;
  a.x = 1;
}
```

```cpp
#include <iostream>
#include <cstring>
#include <vector>
using namespace std;

class Exception
{
public:
  const static int x = 1;
};

int main()
{
  Exception a;
  a.x;
  return 0;
}
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp:9:14: error: ISO C++ forbids in-class initialization of non-cons
t static member ‘Exception::x’
    9 |   static int x = 1;
      |              ^
```

类的静态成员函数

- 通过“类名::函数名()”的方式调用

```
class Clock{
public:
   static void dump() { cout << "my clock"; }
};

int main() {
  Clock::dump();
}
```

静态成员独立于单个类对象，无需实例化对象即可进行访问（与普通成员不同）

类的静态成员函数不能访问普通成员

- 原因：静态成员函数没有隐含的this形参
- 静态成员函数只能够访问静态成员属性、调用静态成员方法，不能访问普通成员属性、调用普通成员方法

```cpp
#include <iostream>
using namespace std;

class Exception
{
public:
    static void func()
    {
        x;
    }
    int x, y;
};
int main()
{
    Exception a;
    a.x = 1;
    return 0;
}
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp: In static member function ‘static void Exception::func()’:
test.cpp:9:5: error: invalid use of member ‘Exception::x’ in static member funct
ion
    9 |     x;
      |     ^
test.cpp:11:7: note: declared here
   11 |   int x, y;
      |       ^
```

类的静态成员函数

- 静态成员函数与普通成员函数的根本区别在于：普通成员函数有 this 指针，可以访问类中的任意成员；而静态成员函数没有 this 指针，只能访问静态成员（包括静态成员变量和静态成员函数）。
- 非static成员函数可以直接访问static成员。

类的常（const）成员函数

- **const成员函数**是指const限定符出现在函数声明中的形参列表之后的**非静态**成员函数
  - 只能读取数据成员，不能改变数据成员
  - 设计类时，一个原则是：不改变成员数据的成员函数都在后面加const，而改变数据成员的成员函数不能加const
  - 类外定义const成员函数时也需要加上const
  - 与static函数不同，const成员函数可以读非const成员变量

```
1.  class Account {
2.    long getAccountNo() const; // const成员函数
3.  };
4.  long Account::getAccountN() { … } // 错误：需要声明const
5.  long Account::getAccountN() const { // 正确
6.      account = account + 3; // 错误：在只读对象内向成员赋值
7.      return account; // 正确：可以读取非const变量
8.  }
```

- 一个类中可以有声明完全一致的const成员函数与非const成员函数（唯一区别是是否在参数列表之后带const）

```
1.  class Account {
2.    long getAccountNo() const; // const成员函数
3.    long getAccountNo(); //非const成员函数
4.  };
```

- 当两个函数都存在时，非const类对象默认调用非const成员函数
  - 仅存在const成员函数时，非const对象可以调用const成员函数

const类对象

- 在定义类对象时，可以定义为const对象
  - 与普通const变量一样，不能修改const对象中的数据，但可以读取对象中的非const数据成员
  - const对象不能调用非const成员函数，只能调用const成员函数

```
1.  class Account {
2.    long getAccountNo(); // 非const成员函数
3.  };
4.  int main() {
5.    const Account account(1234567890L);
6.    long acc = account.getAccountNo(); // 错误：const对象不能调用非const成员函数
7.    … …
8.  }
```

```cpp
#include <iostream>
using namespace std;

class Exception
{
public:
  void func() const {}
  void func2() {}
};
int main()
{
  const Exception a;
  a.func();
  a.func2();
  return 0;
}
```

const对象不能调用非const成员函数的原因：

- const对象的指针为const class A* this，因此传入非const成员函数时编译器报错（类型不匹配，无法从const 指针转换为非const指针）。
- 非const对象的指针为class A* this，可调用const成员函数，因为可从非const指针转换为const指针。

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp: In function ‘int main()’:
test.cpp:14:11: error: passing ‘const Exception’ as ‘this’ argument discards qua
lifiers [-fpermissive]
   14 |   a.func2();
      |           ^
test.cpp:8:8: note:   in call to ‘void Exception::func2()’
    8 |   void func2() {}
      |        ^~~~~
```

本章内容

1. C++中的类
2. 成员函数
3. 类的静态成员
4. 构造函数
5. 析构函数
6. 类的组合
7. 结构体、联合体、位域

4. 构造函数

- 形式：构造函数名与类名相同，没有返回值
- 作用：在对象被创建时使用特定的值构造对象，将对象初始化为一个特定的初始状态
- 时机：在对象创建时被自动调用
- 默认：如果程序中未声明，则系统自动产生出一个默认构造函数，其参数列表为空

构造函数的必要性

```
struct Clock{
   int hour;
   int minute;
   int second;
};

void func(){
   Clock c = {10, 10, 10}; //correct
}
```

```
Class Clock{
public:
   //公有成员......
private:
   int hour;
   int minute;
   int second;
};

void func(){
   Clock c = {10, 10, 10};//error
}
```

- 左侧代码：struct成员的初始化
- 右侧代码：class中私有成员的初始化
- 如果右侧代码可行，那么就实际绕过成员函数访问私有的数据成员了，需构造函数作为代理来完成私有成员的初始化

不同类型的构造函数

```
class Clock {
public:
  //无参构造函数
  Clock();
  //含参构造函数
  Clock(int newH,int newM,int newS);
  //带初始化列表的含参构造函数
  Clock(int newH): hour(newH), minute(0), second(0);
  void setTime(int newH, int newM, int newS);
  void showTime();
private:
   int hour, minute, second;
};
```

无参构造函数

- C++规定，每个类必须有默认的构造函数，没有构造函数就不能创建对象
- 若没有提供任何构造函数，那么C++会自动提供一个默认的无参构造函数
- 只要类中提供了任意一个构造函数，那么C++就不再自动提供默认构造函数

```
class Clock {
private:
  int hour, minute, second;
};

int main() {
  //正确，调用默认的无参构造函数
  Clock myClock;
}
```

```
class Clock {
public:
  Clock(int newH, int newM, int newS);
private:
  int hour, minute, second;
};

int main() {
  Clock myClock; //错误，默认构造函数不存在
  Clock myClock(10, 0, 0); //正确，调用含参的构造函数
}
```

含参构造函数

- 含参构造函数的参数通常用于给对象的成员数据赋初始值

```
Clock::Clock(int newH, int newM, int newS) {
  this->hour = newH;
  this->minute = newM;
  this->second = newS;
}

int main() {
  Clock c(0,0,0);
  c.showTime();
  return 0;
}
```

带初始化列表的含参构造函数

- 含参构造函数的初始化列表以一个冒号开始，接着是以逗号分隔的数据成员列表，每个数据成员后面跟一个放在括号中的初始化表达式
- 用初始化列表来进行初始化，写法方便、简练，尤其当需要初始化的数据成员较多时更具优越性

```
Clock::Clock(int newH):
  hour(newH), minute(0), second(0) {}

int main() {
  Clock c(10);
  c.showTime();
  return 0;
}
```

委托构造函数

- 一个构造函数的代码可以在初始化列表中调用同一个类中的另一个构造函数，这样的构造函数被称为委托构造函数
- 委托构造函数有助于缩短、简化构造函数代码，使类定义更容易理解

```
class Clock {
public:
  Clock(int newH, int newM, int newS) {  cout << "Constructor 1" << endl; }
  Clock(int newH) : Clock(newH, 0, 0) { cout << "Constructor 2" << endl; }
private:
  int hour, minute, second;
};

int main() {
  Clock myClock1(11, 30, 0);
  Clock myClock2(10);
}
```

运行结果如下：

Constructor1
Constructor1
Constructor2

拷贝构造函数

- 作用：用一个已存在的对象去初始化同类型的新对象
- 时机：使用对象作为函数初始参数时触发
- 默认：若程序中未定义拷贝构造函数，编译器会生成一个默认构造函数，用于对原对象中的各个数据成员直接进行数据复制

```
class Clock {
public:
  Clock(int newH, int newM, int newS):
        hour(newH), minute(newM), second(newS) {}
  void showTime() { cout << hour << ":" << minute << ":" << second; }

private:
  int hour, minute, second;
};
```

```
void f(Clock clock) {
  clock.showTime();
}

int main() {
  Clock myClock1(11, 30, 0);
  Clock myClock2(myClock1);
  f(myClock2);
}
```

实现拷贝构造函数

Clock &clock

```
class Clock {
public:
  Clock(int newH, int newM, int newS): hour(newH), minute(newM), second(newS) {}
  Clock(Clock clock) : hour(clock.hour), minute(clock.minute), second(clock.second) {}
private:
  int hour, minute, second;
};

int main() { Clock myClock1(11, 30, 0);  Clock myClock2(myClock1); }
```

- **拷贝构造函数的形参必须为本类的对象引用**
- 引用可以看作是“披着普通变量外衣的指针变量”：具有指针的传址特性，以及普通变量的访问方式。
- 为什么拷贝构造函数不能使用值传递？因为值传递要调用拷贝构造函数。

本章内容

1. C++中的类
2. 成员函数
3. 类的静态成员
4. 构造函数
5. 析构函数
6. 类的组合
7. 结构体、联合体、位域

5. 析构函数

- 形式：~类型名，没有返回值，也不能带有任何参数
- 作用：完成对象被删除前的一些清理工作。
- 时机：在对象的生存期结束的时刻系统自动调用它，然后再释放此对象所属的空间
- 默认：如果程序中未声明析构函数，编译器将自动产生一个隐含的析构函数

析构函数示例

```
class Clock {
public:
  Clock() { cout << "Constructor
    called"<<endl; }
  ~Clock() { cout << "Destructor
    called"<<endl; }
};

int main() {
  Clock clocks[2]; //构造函数调用2次
  Clock *clock = new Clock; //构造函数调用
  delete clock; //析构函数调用

  cout << "-----" << endl;

  clock = new Clock[2]; //构造函数调用2次
  delete[] clock; //析构函数调用2次

  cout << "Main ends." << endl;
  return 0;
}
```

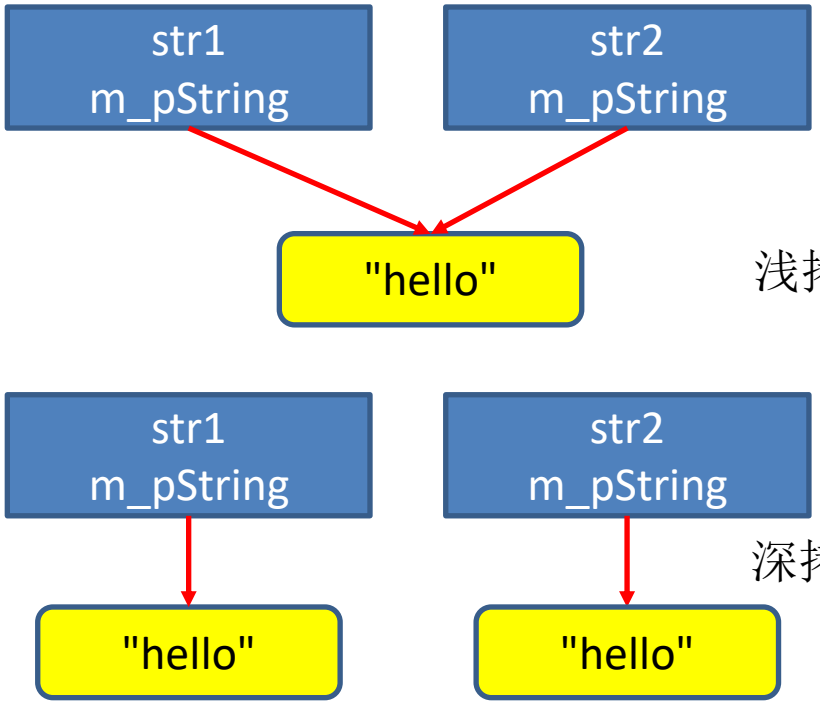
运行结果如下：

```
Constructor called
Constructor called
Constructor called
Destructor called
-----
Constructor called
Constructor called
Destructor called
Destructor called
Main ends.
Destructor called
Destructor called
```

析构函数下的深拷贝与浅拷贝问题

```
class CMyString {
public:
  CMyString(char *str) { //含参构造函数
    m_pString = new char[strlen(str) + 1];
    strcpy(m_pString, str);
  }
  CMyString(CMyString &obj) { //拷贝构造函数
    this->m_pString = obj.m_pString;
  }
  ~CMyString() { //为何有if还是double free?
    if (m_pString != NULL) {
      delete[] m_pString;
      m_pString = NULL;}
  }
private:
  char *m_pString;
};
int main() {
  CMyString str1((char*)"hello");
  CMyString str2(str1);}
```

Double Free!



修改方式1：

```
CMyString(CMyString &obj) {
  m_pString = new char[strlen(obj.m_pString) + 1];
  strcpy(m_pString, obj.m_pString);
}
```

修改方式2：(委托构造函数)

```
CMyString(CMyString &obj): CMyString(obj.m_pString) {}
```

类对象的数组的析构

- delete用于删除new产生的单个类对象，delete[]用于删除new []产生的多个类对象的数组。下例中的delete mys;需要改为delete[] mys;

```
6   class MyClass
7   {
8   public:
9     MyClass(const char *str)
10    {
11      m_str = new char[strlen(str) + 1];
12      strcpy(m_str, str);
13    }
14    ~MyClass() { delete m_str; }
15
16  private:
17    char *m_str;
18  };
19  int main()
20  {
21    int *p = new int;
22    MyClass *mys = new MyClass[3]{MyClass("1"), MyClass("2"), MyClass("3")};
23    delete mys;
24    free(p);
25  }
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
munmap_chunk(): invalid pointer
Aborted (core dumped)
```

C++程序的内存分配

由C/C++编译的程序占用的内存分为以下几部分：

1. **栈区（stack）**：由编译器在需要的时候分配、在不需要的时候释放的存储区，用于存储局部变量、函数参数等。其具有后进先出的栈结构，内存空间连续。
2. **堆区（heap）**：由new或malloc代码分配的内存块，编译器不会去自动释放堆空间，需要编写代码释放堆空间（new对应delete，malloc对应free）。若程序没有释放堆空间，则在程序结束后，操作系统自动回收堆空间。其具有链表结构（不同于数据结构里具有树形结构的堆），内存空间不连续。堆的效率比栈要低得多（尽量不使用：一是效率低，二是容易内存泄漏）。
3. **全局/静态区**：全局变量和静态变量的存储区域。程序结束后由操作系统释放。
4. **常量区**：常量的存储区域。程序结束后由操作系统释放。
5. **代码区**：存放函数体的二进制代码。程序结束后由操作系统释放。

malloc/free与new/delete的区别：

- malloc/free继承自C语言，malloc分配特定大小的堆空间时无法调用类的构造函数。类似地，free释放堆空间时无法调用类的析构函数。
- new/delete分配和释放用于类对象的堆空间时分别调用类的构造与析构函数。

```cpp
#include <iostream>
using namespace std;

int a = 0; // 全局/静态区
char *p1;  // 全局/静态区

class A
{
public:
    A() { cout << "C" << endl; }
    ~A() { cout << "D" << endl; }
};

int main()
{
    int b;                          //栈
    char *p1;                       //栈
    char *p2 = (char *)malloc(10); //堆；10字节的区域（char类型）。

    void *mem = malloc(sizeof(A)); //堆；A大小的区域（无类型）。
    cout << "----------" << endl;
    A *p3 = new (mem) A(); //在上述空间构造A对象

    cout << "----------" << endl;
    A *p4 = new A; //堆；1个A对象的区域。

    static int c = 0; //全局/静态区
    free(p2);
    free(p3);
    cout << "----------" << endl;
    delete (p4);
}
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
----------
C
----------
C
----------
D
```

下列代码有什么问题？（Ubuntu系统下的报错）

```cpp
#include <iostream>
using namespace std;

int a = 0; // 全局/静态区
char *p1;  // 全局/静态区

class A
{
public:
    A() { cout << "C" << endl; }
    ~A() { cout << "D" << endl; }
};

int main()
{
    int b;                          //栈
    char *p1;                       //栈
    char *p2 = (char *)malloc(10); //堆；10字节的区域（char类型）。

    void *mem = malloc(sizeof(A)); //堆；A大小的区域（无类型）。
    cout << "----------" << endl;
    A *p3 = new (mem) A(); //在上述空间构造A对象

    cout << "----------" << endl;
    A *p4 = new A; //堆；1个A对象的区域。

    static int c = 0; //全局/静态区
    free(p2);
    free(p3);
    free(p4);
    cout << "----------" << endl;
    delete (p4);
}
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
----------
C
----------
C
----------
D
free(): double free detected in tcache 2
Aborted (core dumped)
```

本章内容

1. C++中的类
2. 成员函数
3. 类的静态成员
4. 构造函数
5. 析构函数
6. 类的组合
7. 结构体、联合体、位域

6. 类的组合

- 类中的成员数据是另一个类的对象，可以在已有抽象的基础上实现更复杂的抽象

```
class Point {  //Point类定义
public:
   int getX() { return x; }
   int getY() { return y; }
private:
   int x, y;
};

class Line {   //Line类的定义
public://外部接口
   double getLen() { return len; }
private:       //私有数据成员
   Point p1, p2;       //Point类的对象p1,p2
   double len;
};
```

类组合的构造函数设计

- 原则：不仅要负责对本类中的基本类型成员数据赋初值，也要对对象成员初始化
- 初始化次序：
  - 对构造函数初始化列表中列出的成员（包括基本类型成员和对象成员）进行初始化
  - 处理完初始化列表之后，再执行构造函数的函数体

C: > Users > Yahui > Drive > VM > test.cpp > Point

```cpp
#include <iostream>
#include<cmath>
using namespace std;

class Point
{ //Point类定义
public:
    Point(int xx, int yy)
    {
        x = xx;
        y = yy;
    }
    int getX() { return x; }
    int getY() { return y; }

private:
    int x, y;
};
```

```cpp
class Line
{ //Line类的定义
public:
    Line(Point xp1, Point xp2) : p1(xp1), p2(xp2)
    {
        double x = (double)(p1.getX() - p2.getX());
        double y = (double)(p1.getY() - p2.getY());
        len = sqrt(x * x + y * y);
    }
    double getLen() { return len; }

private:
    Point p1, p2;
    double len;
};

int main()
{
    Point myp1(1, 1), myp2(4, 5);
    Line line(myp1, myp2);
    cout << line.getLen() << endl;
    return 0;
}
```

```
yahui@yahui-VirtualBox:/media/sf_VM$ g++ test.cpp
yahui@yahui-VirtualBox:/media/sf_VM$ ./a.out
5
```

本章内容

1. C++中的类
2. 成员函数
3. 类的静态成员
4. 构造函数
5. 析构函数
6. 类的组合
7. 结构体、联合体、位域

7. 结构体、联合体、位域

- 结构体(struct)是一种特殊形态的类
  - **C++中的结构体**与类的区别：类的默认访问权限是private，结构体的默认访问权限是public（注意：C++中的结构体是C中的结构体的扩展）
  - 结构体存在的主要原因：与C语言保持兼容

- 联合体(union)与结构体有一些相似之处，两者的不同在于
  - 结构体中，各成员有各自的内存空间， 一个结构体变量的总长度是各成员长度之和
  - 联合体中，各成员共享一段内存空间，一个联合体变量的长度等于各成员中最长的长度

- 位域(bit-field)：节省存储空间，并使处理简便
  - 有些信息在存储时，并不需要占用一个完整的字节， 只需占几个或一个二进制位
  - 把一个字节中的二进位划分为几个不同的区域，并说明每个区域的位数
  - 每个域有一个域名，允许在程序中按域名进行操作

结构体的定义和初始化

- 结构体定义

```
struct 结构体名称 {
    公有成员
protected:
   保护型成员
private:
    私有成员
};
```

- 一些结构体变量的初始化可以用以下形式

```
类型名 变量名 = { 成员数据1初值, 成员数据2初值, …… };
```

示例：用结构体表示学生的信息

C: > Users > Yahui > Drive > VM > test.cpp > Student

```cpp
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

struct Student
{                   //学生信息结构体
    int num;        //学号
    string name; //姓名，字符串对象
    char sex;       //性别
    int age;        //年龄
};

int main()
{
    Student stu = {97001, "Lin Lin", 'F', 19};
    cout << "Num: " << stu.num << endl;
    cout << "Name: " << stu.name << endl;
    cout << "Sex: " << stu.sex << endl;
    cout << "Age: " << stu.age << endl;
    return 0;
}
```

```
yahui@yahui-VirtualBox:/media/sf_VM$ g++ test.cpp
yahui@yahui-VirtualBox:/media/sf_VM$ ./a.out
Num: 97001
Name: Lin Lin
Sex: F
Age: 19
```

C++中的struct不同于C中的struct

主要区别：

- C中的struct不能有成员函数，只能有成员变量。 C++中的struct两者均可以有。
- C中的struct只有public访问权限。 C++中的struct有三种访问权限。
- C中的struct不能有静态成员。 C++中的struct可以有静态成员。

联合体

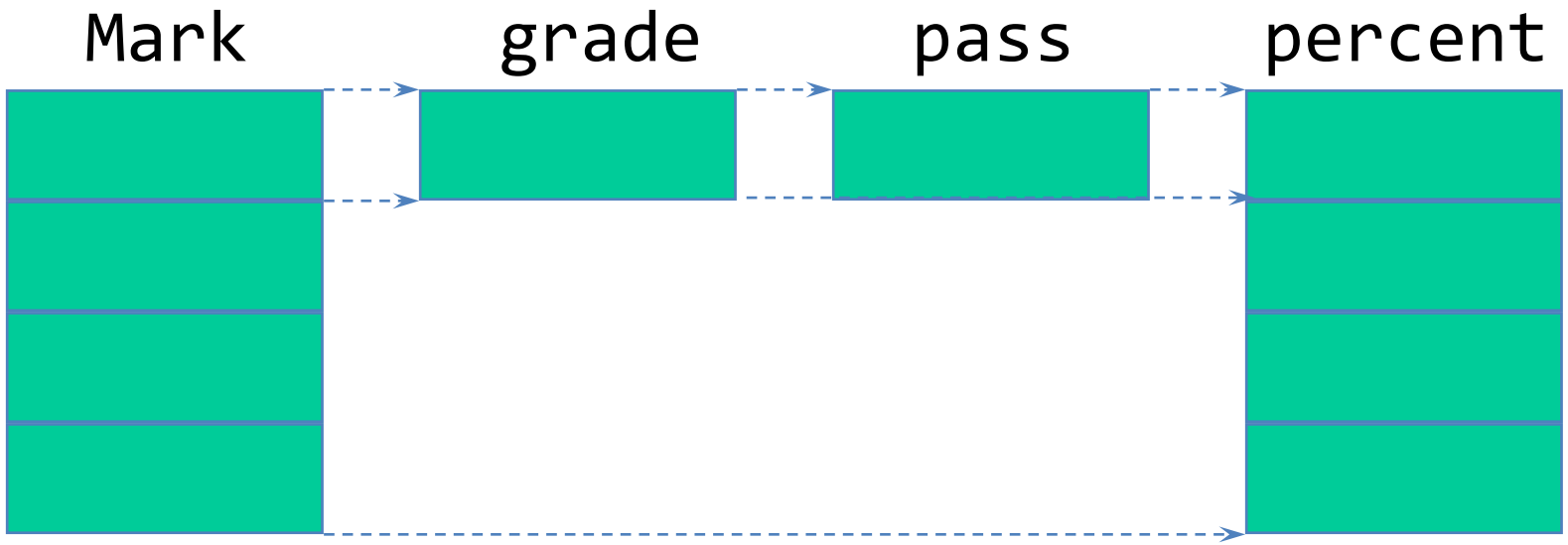
- 声明形式

```
union 联合体名称 {
    公有成员
protected:
    保护型成员
private:
    私有成员
};
```

- 特点：
  - 成员共用相同的内存单元
  - 任何两个成员不会同时有效

联合体的内存分配

```
union Mark {   //表示成绩的联合体
   char grade; //等级制的成绩
   bool pass;  //只记是否通过课程的成绩
   int percent; //百分制的成绩
};
```



无名联合

无名联合没有标记名，只是声明一个成员项的集合，这些成员项具有相同的内存地址，可以由成员项的名字直接访问。

例：

```
union {
   int i;
   float f;
}
```

在程序中可以这样使用：

```
i = 10;
f = 2.2;
```

联合体的构造与析构

- 如果联合体的**非静态成员**包含自定义的构造或析构函数，那么联合体的默认构造与析构函数将被删除。

```cpp
#include <iostream>
#include <cstring>
#include <vector>
using namespace std;

class B
{
public:
  B(int x) { m_x = x; }
  ~B() {}
  int m_x;
};

union A
{
  B b;
};

int main()
{
  A a;
  B b(10);
  a.b = b;
  cout << a.b.m_x << endl;
}
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
test.cpp: In function ‘int main()’:
test.cpp:21:5: error: use of deleted function ‘A::A()’
   21 |   A a;
      |     ^
test.cpp:14:7: note: ‘A::A()’ is implicitly deleted because the default d
efinition would be ill-formed:
   14 | union A
      |       ^
test.cpp:14:7: error: no matching function for call to ‘B::B()’
test.cpp:9:3: note: candidate: ‘B::B(int)’
    9 |   B(int x) { m_x = x; }
      |   ^
test.cpp:9:3: note:   candidate expects 1 argument, 0 provided
test.cpp:6:7: note: candidate: ‘constexpr B::B(const B&)’
    6 | class B
      |       ^
test.cpp:6:7: note:   candidate expects 1 argument, 0 provided
test.cpp:21:5: error: use of deleted function ‘A::~A()’
   21 |   A a;
      |     ^
test.cpp:14:7: note: ‘A::~A()’ is implicitly deleted because the default
definition would be ill-formed:
   14 | union A
      |       ^
test.cpp:16:5: error: union member ‘A::b’ with non-trivial ‘B::~B()’
   16 |   B b;
      |     ^
```

联合体的构造与析构

- 如果联合体的**非静态成员**包含自定义的构造或析构函数，那么联合体的默认构造与析构函数将被删除。两种解决方案：

```
6   class B
7   {
8   public:
9     B(int x) { m_x = x; }
10    ~B() {}
11    int m_x;
12  };
13  union A
14  {
15    A() {}  // Explicit constructor definition
16    ~A(){}; // Explicit destructor definition
17    B b;
18  };
19
20  int main()
21  {
22    A a;
23    B b(10);
24    a.b = b;
25    cout << a.b.m_x << endl;
26  }
```

```
6   class B
7   {
8   public:
9     B(int x) { m_x = x; }
10    ~B() {}
11    int m_x;
12  };
13  union A
14  {
15    // A() {}  // Explicit constructor definition
16    // ~A(){}; // Explicit destructor definition
17    static B b;
18  };
19
20  int main()
21  {
22    A a;
23    B b(10);
24    a.b = b;
25    cout << a.b.m_x << endl;
26  }
```

位域

- 位域的声明形式
  - 数据类型说明符 成员名：位数;
- 位域的作用
  - 通过“打包”，使类的不同成员共享相同的字节，从而节省存储空间。
- 注意事项
  - 具体的打包方式，因编译器而异；
  - 只有bool、char、int和枚举类型的成员，允许定义为位域；
  - 节省空间，但可能增加时间开销。

```cpp
#include <string>
#include <iostream>
using namespace std;

struct S
{
    // three-bit unsigned field,
    // allowed values are 0...7
    unsigned int b : 3; // 位域, just 3 bits
};

int main()
{

    S s = {6};
    ++s.b; // store the value 7 in the bit field
    std::cout << s.b << '\n';
    ++s.b;                  // the value 8 does not fit in this bit field
    std::cout << s.b << '\n'; // formally implementation-defined, typically 0


}
```

```
yahui@Yahui:/media/sf_VM$ g++ test.cpp
yahui@Yahui:/media/sf_VM$ ./a.out
7
0
```

作业

- YOJ-344. 三角形
- YOJ-350. 完善student类的构造函数
- YOJ-605. 约瑟夫问题（面向对象版本）