§9.类和对象基础

9.1.类和对象的基本概念

9.1.1.基本概念（略）

9.1.2.类的声明

在结构体只包含数据成员的基础上，引入成员函数的概念，使结构体同时拥有数据成员和成员函数

```
class student {
    int num;
    char name[20];
    char sex;
    void display()
    {
        cout << "num:"  << num  << endl;
        cout << "name:" << name << endl;
        cout << "sex:"  << sex  << endl;
    }
};
```

```
struct student {
    int num;
    char name[20];
    char sex;
};
```

§9. 类和对象基础

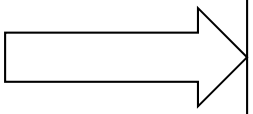
9.1. 类和对象的基本概念

9.1.2. 类的声明

★ 用sizeof(类名)计算类的大小时，成员函数不占用空间

★ 通过类的成员访问限定符(private/public)，可以指定成员的属性是私有(private)或公有(public)，私有不能被外界访问,公有可被外界所访问，由实际应用决定

- 缺省均为私有
- 通常数据成员private,成员函数public
- 成员访问限定符是限"外部"的访问，类的"内部"不受限定符的限制



```
class student {
  private:
    int num;
    char name[20];
    char sex;
  public:
    void display()
    {
        cout << "num:"  << num  << endl;
        cout << "name:" << name << endl;
        cout << "sex:"  << sex  << endl;
    }
};
```

本例：无论三个数据成员的限定符是什么；无论display函数的限定符是什么；都不影响display函数对三个数据成员的访问

★ 在类的定义中，private/public出现的顺序，次数无限制

```
class student {
  public:
    void display()
    {
        cout << "num:"  << num  << endl;
        cout << "name:" << name << endl;
        cout << "sex:"  << sex  << endl;
    }
  private:
    int num;
    char name[20];
    char sex;
};
```

class做为一个整体，不必考虑对数据成员的访问在数据成员的定义之前

```
class student {
  private:
    int num;
  public:
    void display()
    {
        cout << "num:"  << num  << endl;
        cout << "name:" << name << endl;
        cout << "sex:"  << sex  << endl;
    }
  private:
    char name[20];
    char sex;
};
```

§9. 类和对象基础

9.1. 类和对象的基本概念

9.1.3. 对象的定义和访问 ——————————→ 结构体类型 —实例化—> 变量

9.1.3.1. 先定义类，再定义对象

类 —实例化—> 对象

★ 含义相同，称呼不同

```
class student {
    ...
    };
student s1;
student s2[10];
student *s3;
```

```
struct student {
    ...
    };
struct student s1;
struct student s2[10];
struct student *s3;
```

★ 结构体变量/对象占用实际的内存空间，根据不同类型在不同区域进行分配

9.1.3.2. 在定义类的同时定义对象

```
class student {
    ...
    } s1, s2[10], *s3;
student s4;
```

```
struct student {
    ...
    } s1, s2[10], *s3;
struct student s4;
```

★ 可以再次用9.1.3.1的方法定义新的变量/对象

9.1.3.3. 直接定义对象(类无名)

```
class {
    ...
    } s1, s2[10], *s3;
```

```
struct {
    ...
    } s1, s2[10], *s3;
```

★ 因为结构体/类无名，因此无法再用9.1.3.1的方法进行新的变量/对象定义

§9. 类和对象基础

9.1. 类和对象的基本概念

9.1.3. 对象的定义和访问

9.1.3.4. 类与结构体的比较

★ 在C++中，结构体也可以加成员函数，能够实现和类完全一样的功能

```
class student {
  private:
    int num;
    char name[20];
    char sex;
  public:
    void display()
    {
        cout << "num:"  << num  << endl;
        cout << "name:" << name << endl;
        cout << "sex:"  << sex  << endl;
    }
};
```

替换为struct,功能完全相同

§9. 类和对象基础

9.1. 类和对象的基本概念

9.1.3. 对象的定义和访问

9.1.3.4. 类与结构体的比较

★ 在C++中，结构体也可以加成员函数，能够实现和类完全一样的功能

★ 若不指定成员访问限定符，则struct缺省为public，class缺省为private

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

struct student {
    int num;
    char name[20];
    char sex;
    void display() {
        cout << num  << endl;
        cout << name << endl;
        cout << sex  << endl;
        }
};

int main()
{
    student s1;
    s1.num = 1001;
    return 0;
}
```

全部public，
外界(main)可访问，
与C相比，多成员函数

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

class student {
    int num;
    char name[20];
    char sex;
    void display() {
        cout << num  << endl;
        cout << name << endl;
        cout << sex  << endl;
        }
};

int main()
{
    student s1;
    s1.num = 1001;
    return 0;
}
```

全部private，
外界(main)不可访问

```
class student {
    int num;
    char name[20];
    char sex;
    void display()
    {
        cout << num  << endl;
        cout << name << endl;
        cout << sex  << endl;
    }
};
```

全部是private

```
struct student {
    int num;
    char name[20];
    char sex;
    void display()
    {
        cout << num  << endl;
        cout << name << endl;
        cout << sex  << endl;
    }
};
```

全部是public

```
class student {
    int num;
    char name[20];
    char sex;
  public:
    void display()
    {
        ...
    }
};
```

私有

公有

```
struct student {
    int num;
    char name[20];
    char sex;
  public:
    void display()
    {
        ...
    }
};
```

公有

公有

§9. 类和对象基础

9.1. 类和对象的基本概念

9.1.3. 对象的定义和访问

9.1.3.5. 对象成员的访问

```
class student {
  private:
    int name[20];
    char sex;
  public:
    int num;
    void display()
    {
        ...
    }
};
```

类定义

```
int main()
{
    student s1, s2[10];
    s1.sex = 'm';        ×
    s1.num=10001;        ✓
    s1.display();        ✓
    s2[0].sex = 'f';     ×
    s2[0].num=10002;     ✓
    s2[3].display();     ✓
}
```

★ 通过对象名访问对象中的成员

①

```
int main()
{
    student s1, *s3=&s1;
    s1.num    = 10001;   ✓
    (*s3).num = 10001;   ✓
    s3->num   = 10001;   ✓
    s1.display();        ✓
    (*s3).display();     ✓
    s3->display();       ✓
}
```

★ 通过指向对象的指针来访问对象中的成员

②

```
int main()
{
   student s1, &s3=s1;
   s1.num = 10001;  ✓
   s3.num = 10001;  ✓
   s1.display();    ✓
   s3.display();    ✓
}
```

★ 通过对象的引用来访问对象中的成员

③

§9. 类和对象基础

9.1. 类和对象的基本概念

9.1.3. 对象的定义和访问

9.1.3.6. 访问规则

★ 只能访问公有的数据成员和成员函数

★ 数据成员可出现在其基本类型允许出现的任何地方

```
class student {
  private:
    int name[20];
    char sex;
  public:
    int num;
    void display()
    {
        ...
    }
};
```

```
int i;
i=10;
k=i+10;
cout << i;
cin >> i;
```

```
student s1;
s1.num=10;
k=s1.num+10;
cout << s1.num;
cin >> s1.num;
```

★ 成员函数的参数传递规则仍为实参单向传值到形参

§9. 类和对象基础

9.1. 类和对象的基本概念

9.1.4. 成员函数

9.1.4.1. 成员函数的实现

体内实现：class中给出成员函数的定义及实现过程

```
class student {
  ...
  public:
    void display()
    {
        cout<<"num:" <<num <<endl;
        cout<<"name:"<<name<<endl;
        cout<<"sex:" <<sex <<endl;
    }
};
```

体外实现：class中给出成员函数的定义，class外部(class后)给出成员函数的实现

★ 函数实现时需要加类的作用域限定符

★ 体外实现的函数体，仍然算类的"内部"，不受private/public访问限定符限制!!!

```
class student {
  public:
    void display();
};

void student::display()
{
    cout << "num:"  <<num  <<endl;
    cout << "name:" <<name <<endl;
    cout << "sex:"  <<sex  <<endl;
}
```

§9. 类和对象基础

9.1. 类和对象的基本概念

9.1.4. 成员函数

9.1.4.2. 成员函数的性质

★ 对应类的成员函数(类函数)，一般的普通函数称为全局函数

★ 成员函数的定义、实现及调用时参数传递的语法规则与全局函数相同

★ 成员函数也受类的成员访问限定符的约束，只有公有的成员函数可以被外部调用

★ 私有和公有的成员函数均可以访问/调用本类的所有数据成员/成员函数，不受private/public的限制(private/public是用来限制外部对成员的访问)

```
class test {
  private:
    int a;
    int f1();
  public:
    int b;
    int f2();
    int f3();
};
```

```
int test::f1()
{   a=10;  ✓
    b=15;  ✓
    f2();  ✓
}
int test::f2()
{   ...
}
int test::f3()
{   a=20;  ✓
    b=25;  ✓
    f1();  ✓
}
```

```
int main()
{
    test t1;

    t1.a=10;   ×
    t1.f1();   ×
    t1.b=15;   ✓
    t1.f2();   ✓
    t1.f3();   ✓
}
```

§9. 类和对象基础

9.1. 类和对象的基本概念

9.1.4. 成员函数

9.1.4.2. 成员函数的性质

★ 全局函数与成员函数可以同名，按照低层屏蔽高层的原则进行，也可以通过域运算符(::级别最高)强制访问高层（类的数据成员与全局变量也遵循此强制访问规则）

```
class test {
  ...
  public:
    int fun();
    int f1();
};

int fun() 全局函数
{
    ...
}
```

```
int test::fun() 类函数
{
    ...
}

int test::f1()
{
    fun();    类函数
    ::fun();  全局函数
}
```

成员函数内部

```
int main()
{
    test t1;

    t1.fun(); 类函数
    fun();    全局函数
}
```

全局函数中表示不会冲突

§9.类和对象基础

9.1.类和对象的基本概念

9.1.5.访问限制

★ 通过类的成员访问限定符(private/public)，可以指定成员的属性是私有(private)或公有(public)，私有不能被外界访问,公有可被外界所访问，由实际应用决定

- 缺省均为私有
- 通常数据成员private,成员函数public
- 成员访问限定符是限"外部"的访问，类的"内部"不受限定符的限制

§9. 类和对象基础

9.1. 类和对象的基本概念

9.1.6. 类的封装性和信息隐蔽

9.1.6.1. 公有接口和私有实现的分离

★ 公有函数可被外界调用，称为类的公共/对外接口通过对象.公用函数(实参表)的方法进行调用，将函数称为方法，将调用过程称为消息传递

★ 如果允许外界直接改变某个数据成员的值，可直接设置属性为public(不提倡)

★ 其它不愿公开的数据成员和成员函数可设置为私有,对外部隐蔽，但仍可通过公有函数进行访问及修改

```
class student {
  private:
    int num;
  public:
    void set(int n)
    {
        num = n;
    }
    void display()
    {
        cout << num << endl;
    }
};
```

```
int main()
{
  student s1, s2;
   s1.set(10);
   s2.set(15);
   s1.display(); 10
   s2.display(); 15

   return 0;
}
```

set/display函数均间接访问了私有成员num

§9. 类和对象基础

9.1. 类和对象的基本概念

9.1.6. 类的封装性和信息隐蔽

9.1.6.1. 公有接口和私有实现的分离

★ 公有函数的形参称为提供给外部的访问接口，在形参的数量、类型、顺序不变的情况下，私有成员的变化及公有函数实现部分的修改不影响外部的调用

①

```
class student {
  private:
    int num;
  public:
    void set(int n)
    {   num = n;
    }
    void display()
    {   cout << num << endl;
    }
};
```

②

```
class student {
  private:
    int xh;
  public:
    void set(int n)
    {   xh = n;
    }
    void display()
    {   printf("%d\n", xh);
    }
};
```

③

```
class student {
  private:
    int xh;
  public:
    void set(int n)
    {   xh = (n>=0 ? n:0);
    }
    void display()
    {   printf("%d\n", xh);
    }
};
```

```
int main()
{
   student s1, s2;
   s1.set(10);
   s2.set(15);
   s1.display(); 10
   s2.display(); 15
   return 0;
}
```

假设class student由乙编写
main函数由甲编写
则：乙用三种方法
甲的程序均不需要变化

§9. 类和对象基础

9.1. 类和对象的基本概念

9.1.6. 类的封装性和信息隐蔽

9.1.6.1. 公有接口和私有实现的分离

应用实例1：

谷歌公司的Android 4.x内核

```
class picture {
```

类的私有数据成员
及成员函数
外界不可见

```
  void show(char *图片名)
```

函数实现，不可见

```
};
```

***公司的游戏软件

```
int main()
{
    ....
    picture p1;
    p1.show(文件名);
    ....
}
```

谷歌公司的Android 7.x内核

```
class picture {
```

类的私有数据成员
及成员函数
外界不可见
可能已进行过很大调整

```
  void show(char *图片名)
```

函数实现，不可见
实现过程可能与2.3完全不同

```
};
```

谷歌称7.x的显示速度
经优化后比4.x快**%
用户程序不需要变化

§9. 类和对象基础

9.1. 类和对象的基本概念

9.1.6. 类的封装性和信息隐蔽

9.1.6.1. 公有接口和私有实现的分离

应用实例2：

A公司的乙团队：V1.0版本

```
class translation {
    类的私有数据成员
    及成员函数
    外界不可见
  void trans(char *英文)
    函数功能为输出中文
    具体实现过程不可见
};
```

A公司的甲团队

```
int main()
{
    ....
    translation t1;
    t1.trans("****");
    ....
}
```

A公司的乙团队：V1.1版本

```
class translation {
    类的私有数据成员
    及成员函数
    外界不可见
    可能已进行过很大调整
  void trans(char *英文)
    函数实现，不可见
    实现过程可能与1.0完全不同
};
```

V1.1比V1.0的翻译结果
更准确，更贴切
用户程序不需要变化
两个团队能同时工作

思考：

1、甲乙两个团队哪个更不可替代？

2、你的职业期望是哪个团队？

3、进入不同团队对不同知识的要求？

§9. 类和对象基础

9.1. 类和对象的基本概念

9.1.6. 类的封装性和信息隐蔽

9.1.6.1. 公有接口和私有实现的分离

9.1.6.2. 类声明和成员函数定义的分离

★ 将类的声明(*.h)与类成员函数的实现(*.cpp)分开

例：假设程序由ex1.cpp、ex2.cpp和ex.h共同构成

```
/* ex.h */
class student {
  private:
    数据成员1;
    ...;
    数据成员n;
  public:
    成员函数1;
    ...;
    成员函数2;
}
```

```
/* ex2.cpp */
#include <iostream>
#include "ex.h"
using namespace std;
main及其它函数的实现
```

```
/* ex1.cpp */
#include <iostream>
#include "ex.h"
using name space std;

返回值 student::成员函数1()
{
    成员函数1的实现;
}

...

返回值 student::成员函数n()
{
    成员函数n的实现;
}
```

§9. 类和对象基础

9.1. 类和对象的基本概念

9.1.6. 类的封装性和信息隐蔽

9.1.6.1. 公有接口和私有实现的分离

9.1.6.2. 类声明和成员函数定义的分离

★ 将类的声明(*.h)与类成员函数的实现(*.cpp)分开

★ 在需要外部调用的地方，只要提供声明部分即可，类的实现可通过库文件(*.lib)或动态链接库(*.dll)的方式提供，而不必提供实现的源码

★一个程序包含多源程序文件的方法已掌握

★ 建立库文件/动态链接库的方法请自学(荣誉课)

§9. 类和对象基础

9.2. 构造函数

9.2.1. 对象的初始化

对象的初值：与普通变量相同，在静态数据区分配的对象，数据成员初值为0；

在动态数据区分配的对象，数据成员的初值随机

对象的初始化方法：

(1)若全部成员都是公有,可按结构体的方式进行初始化(若有私有成员，不能用此方法)

```
#include <iostream>
using namespace std;
class Time {
   public:
     int f2();  //函数不占用对象空间
     int hour;
     int minute;
     int sec;
  };
int main()
{
   Time t1={14,15,23};
   cout << t1.hour << t1.minute
                   << t1.sec << endl;
}
```

以下两种形式均报错：

```
Time t1(14,15,23);
Time t1=(14,15,23);
```

```
#include <iostream>
using namespace std;
class Time {
   public:
     int hour;
     int minute;
   private:
     int sec;
     int f2();  //函数不占空间
  };
int main()
{
   Time t1={14,15,23};
   cout << t1.hour << t1.minute
            << endl;
} //编译报错
```

```
error C2440: “初始化”: 无法从“initializer list”转换为“Time”
message : 无构造函数可以接受源类型，或构造函数重载决策不明确
```

```
#include <iostream>
using namespace std;
class Time {
     int f1(); //缺省私有
   public:
     int hour;
     int minute;
     int sec;
};
int main()
{
   Time t1={14,15,23};
   cout << t1.hour << t1.minute
              << t1.sec << endl;
}
```

§9. 类和对象基础

9.2. 构造函数

9.2.1. 对象的初始化

对象的初始化方法：

(1)若全部成员都是公有,可按结构体的方式进行初始化(若有私有成员，不能用此方法)

(2)写一个赋初值的公有成员函数，在其它成员被调用之前进行调用

```
class Time {
    private:
        int hour;
        int minute;
        int sec;
    public:
        void set(int h, int m, int s)
        {    hour=h;
             minute=m;
             sec=s;
         }
    };
int main()
{   Time t;

    t.set(14,15,23);
    t.其它
}
```

(3)声明类时对数据成员进行初始化

（C++11标准支持，目前双编译器均可）

```
class Time {
    public:
        int hour=0;
        int minute=0;
        int sec=0;
    };
```

不同对象的值被统一初始化，无法个性化

§9. 类和对象基础

9.2. 构造函数

9.2.2. 构造函数的引入及使用

引入：完成对象的初始化工作,对象建立时被自动调用

形式：与类同名，无返回类型(非void,也不是缺省int)

```
class Time {
   private:
      int hour;
      int minute;
      int sec;
   public:
      Time()
      {  hour=0;
         minute=0;
         sec=0;
       }
   };
int main()
{
   Time t; //t的三个成员都是0
   t.display();
}
```

体内实现

```
class Time {
   private:
      int hour;
      int minute;
      int sec;
   public:
      Time();
   };
Time::Time()
{  hour=0;
   minute=0;
   sec=0;
}
int main()
{
   Time t; //t的三个成员都是0
   Time t.display();
}
```

体外实现

§9. 类和对象基础

9.2. 构造函数

9.2.2. 构造函数的引入及使用

引入：完成对象的初始化工作，对象建立时被自动调用

形式：与类同名，无返回类型(非void，也不是缺省int)

使用：

★ 对象建立时被自动调用

★ 构造函数必须公有

★ 若不指定构造函数，则系统缺省生成一个构造函数，形式为无参空体

★ 若用户定义了构造函数，则缺省构造函数不再存在

★ 构造函数既可以体内实现，也可以体外实现

★ 允许定义带参数的构造函数，以解决无参构造函数初始化各对象的值相同的情况

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

class Time {
   private:
      int hour;
      int minute;
      int sec;
   public:
      Time(int h, int m, int s);
      void display()
      {  cout << hour << minute << sec << endl;
      }
   };
Time::Time(int h, int m, int s)
{
   hour   = h;
   minute = m;
   sec    = s;
}
int main()
{
   Time t1(14,15,23);
   Time t2{14,15,23};
   Time t3={14,15,23};
   t1.display();
   t2.display();
   t3.display();
   Time t4; //错误，没有对应的无参构造函数
}
```

也允许体内实现

三种形式均可

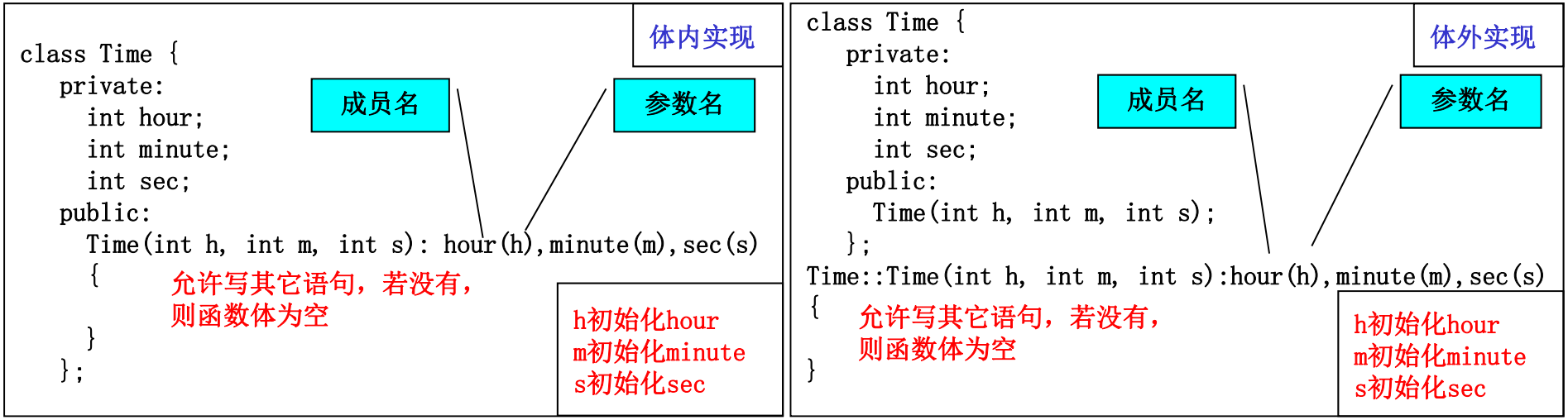
§9. 类和对象基础

9.2. 构造函数

9.2.2. 构造函数的引入及使用

使用：

★ 有参构造函数可以使用参数初始化表来对数据成员进行初始化



★ 有参构造函数可以使用参数初始化表来对数据成员进行初始化(仅适用于简单的赋值)

```
Time::Time(int h, int m, int s)
{   if (h>=0 && h<=23)
        hour = h;
    else
        hour = 0;
    ....
}
```

无法通过参数初始化表实现

§9. 类和对象基础

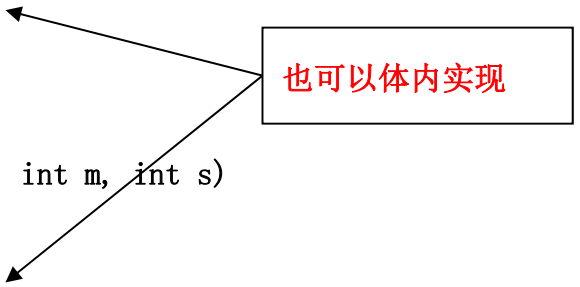
9.2. 构造函数

9.2.2. 构造函数的引入及使用

使用：

★ 构造函数允许重载

```
class Time {
   ...
   public:
      Time();
      Time(int h, int m, int s);
   };
Time::Time()
{  hour   = 0;
   minute = 0;
   sec    = 0;
}
Time::Time(int h, int m, int s)
{  hour   = h;
   minute = m;
   sec    = s;
}
int main()
{  Time t(14,15,23); //正确
   Time t2;          //正确
   ...
}
```



§9. 类和对象基础

9.2. 构造函数

9.2.2. 构造函数的引入及使用

使用：

★ 构造函数允许带默认参数，但要注意可能与重载产生二义性冲突

无参与带缺省参数的重载，不冲突
适应带0/2/3个参数的情况

```
class Time {
   ...
   public:
      Time();
      Time(int h, int m, int s=0);
   };
Time::Time()
{  hour   = 0;
   minute = 0;
   sec    = 0;
}
Time::Time(int h, int m, int s)
{  hour   = h;
   minute = m;
   sec    = s;
}
int main()
{  Time t1(14,15,23); //正确
   Time t2(14,15);    //正确
   Time t3;           //正确
}
```

无参与带缺省参数的重载，冲突!!!

```
class Time {
   ...
   public:
      Time();
      Time(int h=0, int m=0, int s=0)
   };
Time::Time()
{  hour   = 0;
   minute = 0;
   sec    = 0;
}
Time::Time(int h, int m, int s)
{  hour   = h;
   minute = m;
   sec    = s;
}
int main()
{  Time t1(14,15,23); //正确
   Time t2(14,15);    //正确
   Time t3(14);       //正确
   Time t4;           //错误
}
```

§9.类和对象基础

9.2.构造函数

9.2.2.构造函数的引入及使用

使用：

★ 构造函数也可以显式调用，一般用于带参构造函数

```
class Test {
  private:
    int a;
  public:
    Test(int x) {
      a=x;
      }
};
Test fun()
{  ...
   return Test(10);  //显式
}

int main()
{
   Test t1(10);       //隐式
   Test t2=Test(10); //显式
   Test t3=Test{10}; //显式
}
```

§9. 类和对象基础

9.2. 析构函数

引入：在对象被撤销时(生命期结束)时被自动调用，完成一些善后工作(主要是内存清理)，
但不是撤销对象本身

形式：

~类名();

★ 无返回值(非void,也不是int)，无参，不允许重载

使用：

★ 对象撤销时被自动调用，用户不能显式调用

★ 析构函数必须公有

★ 若不指定析构函数，则系统缺省生成一个析构函数,形式为无参空体

★ 若用户定义了析构函数，则缺省析构函数不再存在

★ 析构函数既可以体内实现，也可以体外实现

★ 在数据成员没有动态内存申请需求的情况下，一般不需要定义析构函数
(动态内存申请为荣誉课内容，此处不再展开)

§9. 类和对象基础

9.4. 构造函数与析构函数的调用时机

构造函数：

★ 自动对象(形参)　　：函数中变量定义时
★ 静态局部对象　　：第一次调用时
★ 静态全局/外部全局对象：程序开始时
~~★ 动态申请的对象　　：~~

main开始前

析构函数：

★ 自动对象(形参)　　：函数结束时
★ 静态局部对象　　：程序结束时(在全局之前)
★ 静态全局/外部全局对象：程序结束时
~~★ 动态申请的对象　　：~~

main结束后

§9. 类和对象基础

9.4. 构造函数与析构函数的调用时机

```
#include <iostream>
using namespace std;
class Time {
   private:
      int hour;
      int minute;
      int second;
   public:
      Time(int h=0, int m=0, int s=0);
      ~Time();
};
Time::Time(int h, int m, int s)
{
    hour   = h;
    minute = m;
    second = s;
    cout << "Time Begin" << endl;
}
Time::~Time()
{   cout << "Time End" << endl;
}
```

```
void fun()
{   Time t1;
    cout << "addr:" << &t1 << endl;
    cout << "fun" <<endl;
}
int main()
{   cout << "main begin" <<endl;
    fun();
    cout << "continue" << endl;
    fun();
    cout << "main end" <<endl;
}
```

程序的运行结果：

```
main begin
Time Begin
addr:地址a
fun
Time End
continue
Time Begin
addr:地址a(同上)
fun
Time End
main end
```

1、函数调用时分配空间
   结束时回收空间
2、函数多次调用则多次
   分配/回收空间
（解决函数模块遗留问题）

§9. 类和对象基础

9.4. 构造函数与析构函数的调用时机

```
#include <iostream>
using namespace std;
class Time {
   private:
      int hour;
      int minute;
      int second;
   public:
      Time(int h=0, int m=0, int s=0);
      ~Time();
};
Time::Time(int h, int m, int s)
{
    hour   = h;
    minute = m;
    second = s;
    cout << "Time Begin" << endl;
}
Time::~Time()
{   cout << "Time End" << endl;
}
```

```
void fun()
{   static Time t1;
    cout << "fun" <<endl;
}
int main()
{   cout << "main begin" <<endl;
    fun();
    cout << "continue" << endl;
    fun();
    cout << "main end" <<endl;
}
```

程序的运行结果：

```
main begin
Time Begin
fun
continue
fun
main end
Time End
```

1、函数第1次调用时分配
2、后续函数调用不分配
3、全部程序结束后回收
（解决函数模块遗留问题）

§9. 类和对象基础

9.4. 构造函数与析构函数的调用时机

```
#include <iostream>
using namespace std;
class Time {
   private:
      int hour;
      int minute;
      int second;
   public:
      Time(int h=0, int m=0, int s=0);
      ~Time();
};
Time::Time(int h, int m, int s)
{
    hour   = h;
    minute = m;
    second = s;
    cout << "Time Begin" << endl;
}
Time::~Time()
{   cout << "Time End" << endl;
}
```

```
Time t1;
void fun()
{   cout << "fun begin" <<endl;
    cout << "fun end" <<endl;
}
int main()
{   cout << "main begin" <<endl;
    fun();
    cout << "main end" <<endl;
}
```

程序的运行结果：

```
Time begin
main begin
fun begin
fun end
main end
Time End
```

§9. 类和对象基础

9.5. 对象数组

9.5.1. 形式

类型 对象名[整型常量表达式] ：一维数组

类型 对象名[整常量1][整常量2] ：二维数组

```
Time t[10];
Time s[3][4];
```

9.5.2. 定义对象时进行初始化

★ 若未定义构造函数或构造函数无参，则按简单对象使用无参构造函数的规则进行

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

class Time {
    private:
        int hour, minute, sec;
    public:
        void display()
        {
            cout << hour << minute  << sec << endl;
        }
};

int main()
{
     Time t[10];
     for (int i=0; i<10; i++)
         t[i].display();
}
```

10个元素的三个成员的值都是随机的，因为调用缺省构造，什么也没做

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

class Time {
    private:
        int hour, minute, sec;
    public:
        Time() { hour=0;  minute=0;  sec=0;}
        void display()
        {  cout << hour << minute << sec << endl;
        }
};

int main()
{
     Time t[10];
     for (int i=0; i<10; i++)
         t[i].display();
}
```

10个元素的三个成员的值都是0，因为调用无参构造

§9. 类和对象基础

9.5. 对象数组

9.5.2. 定义对象时进行初始化

★ 若带参构造函数只带一个参数，可用数组定义时初始化的方法进行

10个元素的三个成员中
hour=1-10,其它两个为0
调用一个参数的构造

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

class Time {
    private:
        int hour, minute, sec;
    public:
        Time(int h)
         {
             hour   = h;
             minute = 0;
             sec    = 0;
         }
        void display()
        {
            cout << hour << minute << sec << endl;
        }
};

int main()
{
     Time t[10] ={1,2,3,4,5,  6,7,8,9,10};
     for (int i=0; i<10; i++)
         t[i].display();
}
```

10个元素的三个成员中
hour=1-10,其它两个为0
两个构造用一个参数的

```
#include <iostream>
using namespace std;
class Time {
    private:
        int hour, minute, sec;
    public:
        Time()
        {
             hour   = 0;
             minute = 0;
             sec    = 0;
        }
        Time(int h)
        {    hour   = h;
             minute = 0;
             sec    = 0;
        }
        void display()
        {
            cout << hour << minute << sec << endl;
        }
};
int main()
{    Time t[10] ={1,2,3,4,5,  6,7,8,9,10};
     for (int i=0; i<10; i++)
         t[i].display();
}
```

§9. 类和对象基础

9.5. 对象数组

9.5.2. 定义对象时进行初始化

★ 若带参构造函数有带一个参数和多个参数共存(可以是带默认参数的构造函数)，则可用数组定义时初始化的方法进行，每个数组元素只传一个参数

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

class Time {
    private:
        int hour, minute, sec;
    public:
        Time()
        {   hour=0;  minute=0;  sec=0;
        }
        Time(int h)
        {   hour=h;  minute=0;  sec=0;
        }
        Time(int h,int m)
        {   hour=h; minute=m; sec=0;
         }
        void display()
        {  cout << hour << minute << sec << endl;
        }
};
int main()
{
     Time t[10] ={1,2,3,4,5,  6,7,8,9,10};
     for (int i=0; i<10; i++)
         t[i].display();
}
```

10个元素的三个成员中
hour=1-10,其它两个为0
多个构造用一个参数的

无参
1、2
重载

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

class Time {
    private:
        int hour, minute, sec;
    public:
        Time(int h=0,int m=0,  int s=0)
        {
             hour   = h;
             minute = m;
             sec    = s;
        }
        void display()
        {  cout << hour << minute << sec << endl;
        }
};
int main()
{
     Time t[10] ={1,2,3,4,5,  6,7,8,9,10};
     for (int i=0; i<10; i++)
         t[i].display();
}
```

10个元素的三个成员中
hour=1-10,其它两个为0
调用带一个参数的构造

带默认参数的构造函数
可带0/1/2/3个参数

§9. 类和对象基础

9.5. 对象数组

9.5.2. 定义对象时进行初始化

★ 如果希望初始化时多于一个参数，则初始化时显式给出构造函数及实参表

★ 初始化的数量不能超过数组大小

★ 定义数组时可不定义大小，有初始化表决定

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


class Time {
    private:
        int hour;
        int minute;
        int sec;
    public:
        Time(int h=0,int m=0,  int s=0)
        {  hour=h; minute=m; sec=s;
        }
        void display()
        {  cout << hour << minute << sec << endl;
        }
};

int main()
{    Time t[10] ={Time(1,2,3),Time(4,5),6,7,8,9,10};
     for (int i=0; i<10; i++)
         t[i].display();
}
```

```
123     ... t[0] 三参构造
450     ... t[1] 两参构造
600     ... t[2] 一参构造
700
800
900
1000
000     ... t[7] 零参构造
000
000
```

```
Time t[10]={Time(1,2,3),Time(4,5),6,7,8,9,10};
        不能比7小
```

```
Time t[]={Time(1,2,3),Time(4,5),6,7,8,9,10};
        自动为7
```

问：以下两种的差别在哪里？

```
Time t[10] ={ {1,2,3},  {4,5},6,7,8,9,10 };
Time t[10] ={ (1,2,3),  (4,5),6,7,8,9,10 };
```

§9.类和对象基础

9.6.对象指针

9.6.1.指向对象的指针

形式：

类名 *指针变量名

```
Time *t;
```

指针的赋值：

```
Time t1, *t;          Time t1, *t=&t1;
t = &t1;
```

定义后赋值语句赋值　　定义时赋初值

```
Time t2[10], *t;      Time t2[10], *t=t2;
t = t2;
```

t指向t2[0]，t++则指向t2[1]

t++ ⇔ t+sizeof(time)

Time换成 int
Time换成结构体的student
方法相同

§9. 类和对象基础

9.6. 对象指针

9.6.1. 指向对象的指针

使用：

Time换成结构体的student
方法相同，仅限public

Time类定义

```
class Time {
   private:
      int minute;
      int sec;
   public:
      int hour; //公有
      Time(int h=0, int m=0, int s=0);
      ~Time();
      void display();
   };
```

指向简单变量的指针

```
Time t1, *t=&t1;

 t  : t1对象的地址
*t : t1对象
(*t).hour       ⇔ t->hour       ⇔ t1.hour;
(*t).display() ⇔ t->display() ⇔ t1.display()
```

指向数组变量的指针

```
Time t2[10], *t=t2;

 t  : t2数组的第[0]个对象的地址
*t : t2数组的第[0]个对象

 (*t).hour       ⇔ t->hour       ⇔ t2[0].hour
 (*t).display() ⇔ t->display() ⇔ t2[0].display()

 t+3    : t2数组的第[3]个对象的地址
*(t+3) : t2数组的第[3]个对象

 (*(t+3)).hour       ⇔ t[3].hour       ⇔ (t+3)->hour ⇔       t2[3].hour
 (*(t+3)).display() ⇔ t[3].display() ⇔ (t+3)->display() ⇔ t2[3].display()
```

§9.类和对象基础

9.6.对象指针

9.6.1.指向对象的指针

9.6.2.指向对象成员的指针

Time换成结构体的student
方法相同，仅限public

9.6.2.1.指向对象的数据成员的指针

定义：数据成员的基类型 *指针变量名

赋值：指针变量名 = 数据成员的地址

```
Time t1;
int *p;
p=&t1.hour;
```

使用：

```
*p ⇔ t1.hour;
```

★ 对象的数据成员必须是public

9.6.2.2.指向对象的成员函数的指针(荣誉课内容，略)

§9. 类和对象基础

9.6. 对象指针

9.6.1. 指向对象的指针

9.6.2. 指向对象成员的指针(其中：指向成员函数的指针 略)

9.6.3. this指针

含义：指向当前被访问的成员函数所对应的对象的指针，名称固定为this，基类型为类名

```
void Time::display()
{
    cout << hour   << endl;
    cout << minute << endl;
    cout << sec    << endl;
}
```

相当于

```
void Time::display(Time *this)
{
    cout << this->hour   << endl;
    cout << this->minute << endl;
    cout << this->sec    << endl;
} //编译会错，只是含义上相当于!!!
```

```
Time t1, t2;
t1.display() 时，this指向t1
        ⇔ t1.display(&t1);
t2.display() 时，this指向t2
        ⇔ t2.display(&t2);
```

```
void Time::set(int h, int m, int s)
{
    hour   = h;
    minute = m;
    sec    = s;
}
```

相当于

```
void Time::set(Time *this,int h,int m,int s)
{
    this->hour   = h;
    this->minute = m;
    this->sec    = s;
} //编译会错，只是含义上相当于!!!
```

```
Time t1, t2;
  t1.set(14,15,23) 时，this指向t1
        ⇔ t1.set(&t1,14,15,23);
  t2.set(16,30,0) 时，this指向t2
        ⇔ t2.set(&t2,16,30,0);
```

§9. 类和对象基础

9.6. 对象指针

9.6.3. this指针

含义：指向当前被访问的成员函数所对应的对象的指针，名称固定为this，基类型为类名

使用：

★ 隐式使用，相当于通过对象调用成员函数时传入该对象的自身的地址

★ 也可以显式使用(但不能显式定义)

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

class Time {
    private:
        int hour, minute, sec;
    public:
        Time(int h, int m, int s)
        {    hour = h;
             minute = m;
             sec = s;
        }
        void display()
        {
             cout << this->hour << this->minute
                             << this->sec << endl;
        }
};

int main()
{
    Time t1(12, 13, 24);
    t1.display();
}
```

可以显式使用

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

class Time {
    private:
        int hour, minute, sec;
    public:
        Time(int h, int m, int s)
        {    hour = h;
             minute = m;
             sec = s;
        }
        void display(Time *this)
        {   cout << this->hour << this->minute
                                << this->sec << endl;
        }
};

int main()
{
    Time t1(12, 13, 24);
    t1.display(&t1);
}
```

不能显式定义

```
//特殊约定，this不能显式，其它名字可以
void display(Time *that)
{   cout << this->hour << that->minute
                       << this->sec << endl;
}
```

§9.类和对象基础

9.7.对象的赋值与复制

9.7.1.对象的赋值

含义：将一个对象的所有数据成员的值对应赋值给另一个已有对象的数据成员

形式：类名 对象名1，对象名2;

...

对象名1=对象名2; //执行语句的方式

★ 两个对象属于同一个类，且不能在定义时赋值

★ 系统默认的赋值操作是将右对象的全部数据成员的值对应赋给左对象的全部数据成员

(理解为整体内存拷贝，但不包括成员函数)，在对象的数据成员无动态内存申请时可直接使用

★ 若对象数据成员是指针并涉及动态内存申请，则需要自行实现

(通过=运算符的重载实现，荣誉课内容)

§9.类和对象基础

9.7.对象的赋值与复制

9.7.2.对象的复制

含义：建立一个新对象,其值与某个已有对象完全相同

使用：

类 对象名(已有对象名)

类 对象名=已有对象名

两种形式
本质一样

```
Time t1(14,15,23), t2(t1), t3=t1;
```

★ 与对象赋值的区别：定义语句/执行语句中

```
Time t1(14,15,23), t2, t3=t1; //复制
t2 = t1; //赋值
```

★ 系统默认的复制操作是将已有对象的全部数据成员的值对应赋给新对象的全部数据成员
(理解为整体内存拷贝，但不包括成员函数)，在对象的数据成员无动态内存申请时可直接使用

★ 若对象数据成员是指针并涉及动态内存申请，则需要自行实现
(通过重定义复制/拷贝构造函数来实现，荣誉课内容)

§9. 类和对象基础

9.8. 友元

9.8.1. 引入

当在外部访问对象时，private全部禁止，public全部允许，为使应用更灵活，引入友元(friend)的概念，允许友元访问private部分

★ 友元不是面向对象的概念，它破坏了数据的封装性，但方便使用，提高了运行效率

问题：在全局函数display(外部)中如何访问私有成员？

```
class Time {
   private:
      int hour;
      int minute;
      int sec;
   public:
      ...
   };


void display(Time t)
{
    想访问 t.hour;
}
```

方法1：通过公有函数间接访问

```
class Time {
   private:
      int hour;
      int minute;
      int sec;
   public:
      int get_hour() { return hour; }
      void set_hour(int h) { hour = h; }
      ...
   };

void display(Time t)
{
    通过 t.get_hour() 读
    通过 t.set_hour(12) 赋值
}
```

当频繁调用时，效率较低

方法2：成员直接公有

```
class Time {
   public:
      int hour;
      int minute;
      int sec;
   public:
      ...
   };

void display(Time t)
{
    直接 t.hour;
}
```

缺点：所有外部函数都能访问
不仅局限于一个display()
失去了类的封装和隐蔽性

§9.类和对象基础

9.8.友元

9.8.1.引入

可以成为类的友元的成分：

★ 全局函数

★ 其它类的成员函数

★ 其它类

友元的声明方式：

在类的声明中，相应要成为友元的函数/类前加friend关键字即可

§9. 类和对象基础

9.8. 友元

9.8.2. 声明全局函数为友元函数

```
class Time {
   private:
      int hour;
      int minute;
      int sec;
      friend void display(Time &t);
   public:
      ...
   };
```

全局函数

```
void display(Time &t)
{
  cout << t.hour ; ✓
}
```

```
void fun(Time &t)
{
  cout << t.hour ; ×
}
```

```
void main()
{
    Time t1;
    display(t1);
}
```

★ 不能直接写成员名，要通过对象来调用
因为不是成员函数，没有this指针

★ 声明友元的位置不限private/public

§9. 类和对象基础

9.8. 友元

9.8.3. 声明其它类的成员函数为友元函数

```
class Time;
```
← 在test中引用Time时，Time尚未定义，因此要提前声明

```
class test {
   public:
      void display(Time &t);
   };
```

```
class Time {
   private:
      int hour;
      ...
      friend void test::display(Time &t);
   };
```

```
void test::display(Time &t)
{
   cout << t.hour << ... << endl;
}
```

~~class Time;~~
← 如无提前声明，test中每个Time前加class也可以（数量多时不方便）

```
class test {
   public:
      void display( class Time &t);
   };
```

```
class Time {
   private:
      int hour;
      ...
      friend void test::display(Time &t);
   };
```

```
void test::display(Time &t)
{
   cout << t.hour << ... << endl;
}
```

§9. 类和对象基础

9.8. 友元

9.8.3. 声明其它类的成员函数为友元函数

```
class Time;                          在test中引用Time时，Time尚未定义因此要提前声明

class test {
   public:
      void display(Time &t);
   };

class Time {
   private:                          声明友元不限定private/public
      int hour;                      但友元函数所在类要符合限定规则
      ...
      friend void test::display(Time &t);
   };

void test::display(Time &t)
{
   cout << t.hour << ... << endl;
}
                  //成员.对象方式访问
```

§9. 类和对象基础

9.8. 友元

9.8.4. 友元类

★ 提前声明遵循刚才的原则

```
class test;//有提前声明
class Time {
   private:
      ...
      friend test;
   };
class test {
   ...
   };
```

test的所有成员函数都可以访问Time的私有成员

```
class test;  //无提前声明
class Time {
   private:
      ...
      friend class test;
   };
class test {
   ...
   };
```

★ 友元是单向而不是双向的

本例中：Time中不能访问test的私有

★ 友元不可传递

```
class A {
   friend class B;
};
class B {
   friend class C;
};
class C {
   C不能访问A的私有成员
};
```

★ C++规定同类的不同对象互为友元

```
class Student {
   private:
      int num;
   public:
      void display();
};
void Student::display()
{  Student s;
   ...
   if (this->num > s.num) {...}
}
```

互为友元