

§ 9. 怎样使用类和对象

9.1. 构造函数

9.1.1. 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
```

```
#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;
}
```

老版C++标准: 错误

要求所有成员, 包括成员函数都必须公有 (VC++6.0)

新版C++标准: 正确

只要所有数据成员均公有即可 (VS2015、CodeBlocks)

§ 9. 怎样使用类和对象

9.1. 构造函数

9.1.1. 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) 新版C++允许在声明类时进行初始化(老版不允许)

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

VC++6.0: 错误

VS2015、CodeBlocks: 正确

§ 9. 怎样使用类和对象

9.1. 构造函数

9.1.2. 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)

问题：

Time t1(); 编译报warning,
sizeof(t1) 编译报error;
为什么? t1是什么?

```
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
    Time t1(); //见问题
}
```

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 t1(); //见问题
}
```

体外实现

§ 9. 怎样使用类和对象

9.1. 构造函数

9.1.2. 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. 怎样使用类和对象

9.1. 构造函数

9.1.2. 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

<pre>class Time { private: int hour; int minute; int sec; public: Time(int h, int m, int s): hour(h), minute(m), sec(s) { 允许写其它语句, 若没有, 则函数体为空 } };</pre>	<div>体内实现</div> <div>成员名</div> <div>参数名</div> <div>h初始化hour m初始化minute s初始化sec</div>
<pre>class Time { private: int hour; int minute; int sec; public: Time(int h, int m, int s); }; Time::Time(int h, int m, int s):hour(h), minute(m), sec(s) { 允许写其它语句, 若没有, 则函数体为空 }</pre>	<div>体外实现</div> <div>成员名</div> <div>参数名</div> <div>h初始化hour m初始化minute s初始化sec</div>

★ 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. 怎样使用类和对象

9.1. 构造函数

9.1.2. 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  
    ...  
}
```

也可以体内实现



§ 9. 怎样使用类和对象

9.1. 构造函数

9.1.2. 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

<pre>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 }</pre>	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
--	---------------------------------------

<pre>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 }</pre>	无参与带缺省参数的重载, 冲突!!!
---	-----------------------

§ 9. 怎样使用类和对象

9.1. 构造函数

9.1.2. 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. 怎样使用类和对象

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

★ 在没有数据成员需要动态内存申请的情况下，一般不需要定义析构函数

★ 在有数据成员需要动态内存申请的情况下，也可以不定义析构函数而通过其他方法释放 (不提倡!!!)

```
class Time {
private:
    int hour;
    int minute;
    int sec;
    char *s;
public:
    Time();
    ~Time();
}; //定义析构

Time::Time()
{
    hour = 0;
    minute = 0;
    sec = 0;
    s = new char[80]; //申请
}

Time::~~Time()
{
    delete s; //释放
}

int main()
{
    Time t1;
    ...
} // 不需要显式调用析构，自动
```

```
class Time {
private:
    int hour;
    int minute;
    int sec;
    char *s;
public:
    Time();
    Release();
}; //未定义析构

Time::Time()
{
    hour = 0;
    minute = 0;
    sec = 0;
    s = new char[80];
}

Time::Release()
{
    delete s; //释放
}

int main()
{
    Time t1;
    ...
    t1.Release();
} // 必须显式调用Release()
```

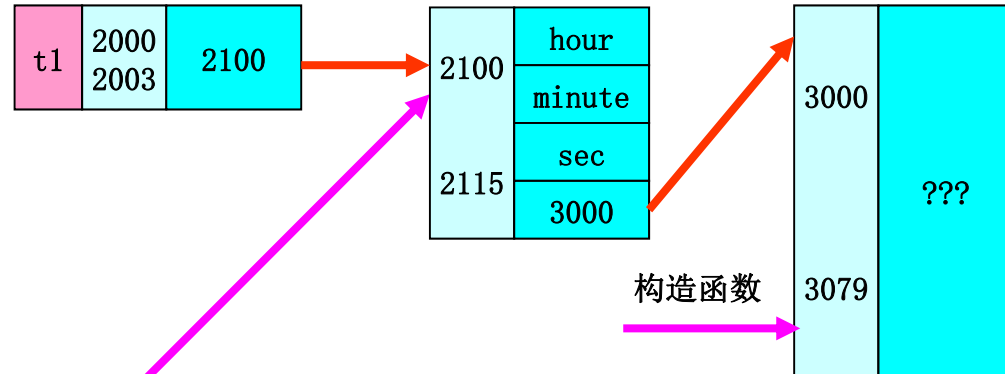
§ 9. 怎样使用类和对象

9.2. 析构造函数

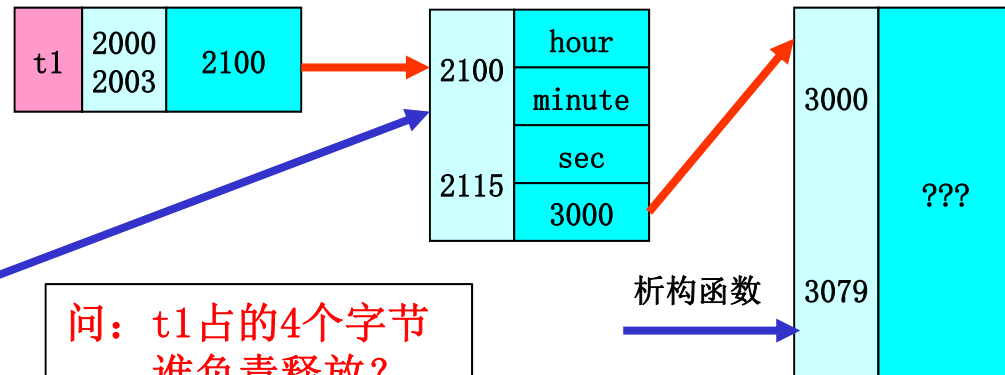
使用:

★ 不要与对象的动态申请混淆

```
#include <iostream>
using namespace std;
class Time {
private:
    int hour;
    int minute;
    int sec;
    char *s;
public:
    Time();
    ~Time();
};
Time::Time()
{
    hour = 0;
    minute = 0;
    sec = 0;
    s = new char[80]; //申请
}
Time::~Time()
{
    delete s; //释放
}
int main()
{
    Time *t1 = new Time; //申请16字节
    cout << "main begin" << endl;
    delete t1;
    cout << "main end" << endl;
}
```



构造函数



析构造函数

问: `t1` 占的4个字节
谁负责释放?
答:

§ 9. 怎样使用类和对象

9.3. 调用构造函数和析构函数的顺序

9.3.1. 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
- ★ new/malloc申请的对象 : new/malloc时

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
- ★ new/malloc申请的对象 : delete/free时

main结束后

本页的某个说法有错，是哪个？参考后面的例子给出答案!!!

§ 9. 怎样使用类和对象

9.3. 调用构造函数和析构函数的顺序

9.3.1. 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 << "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
Time End
continue
Time Begin
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

§ 9. 怎样使用类和对象

9.3. 调用构造函数和析构函数的顺序

9.3.1. 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

§ 9. 怎样使用类和对象

9.3. 调用构造函数和析构函数的顺序

9.3.1. 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. 怎样使用类和对象

9.3. 调用构造函数和析构函数的顺序

9.3.1. 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;
}
```

```
int main()
{
    cout << "main begin" << endl;
    Time *t1 = new Time;
    cout << "new end" << endl;
    delete t1;
    cout << "main end" << endl;
}
```

程序的运行结果：



§ 9. 怎样使用类和对象

9.3. 调用构造函数和析构函数的顺序

9.3.1. 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;
}
```

```
int main()
{
    cout << "main begin" << endl;
    Time *t1 = (Time *)malloc(sizeof(Time));
    cout << "new end" << endl;
    free(t1);
    cout << "main end" << endl;
}
```

程序的运行结果：

?

将C++方式的new/delete更换为
C方式的malloc/free，会怎样???

§ 9. 怎样使用类和对象

9.3. 调用构造函数和析构函数的顺序

9.3.1. 构造函数与析构函数的调用时机

9.3.2. 相同性质的多个对象的调用顺序

★ 构造函数按定义的顺序依次执行，析构函数按构造函数进栈的概念反向依次执行

```
#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" << hour <<endl;
}
Time::~~Time()
{
    cout << "Time End" << hour << endl;
}
```

```
void fun()
{
    Time t1(15), t2(16);
    cout << "fun" <<endl;
}
int main()
{
    cout << "main begin" <<endl;
    fun();
    cout << "main end" <<endl;
}
```

程序的运行结果：

```
main begin
Time Begin15
Time Begin16
fun
Time End16
Time End15
main end
```

t1, t2都是自动变量
构造: t1, t2
析构: t2, t1

§ 9. 怎样使用类和对象

9.3. 调用构造函数和析构函数的顺序

9.3.1. 构造函数与析构函数的调用时机

9.3.2. 相同性质的多个对象的调用顺序

★ 构造函数按定义的顺序依次执行，析构函数按构造函数进栈的概念反向依次执行

```
#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" << hour <<endl;
}
Time::~~Time()
{
    cout << "Time End" << hour << endl;
}
```

```
void fun()
{
    static Time t1(15), t2(16);
    cout << "fun" <<endl;
}
int main()
{
    cout << "main begin" <<endl;
    fun();
    cout << "main end" <<endl;
}
```

程序的运行结果：

?

t1, t2都是静态局部变量
构造：t1, t2
析构：t2, t1

§ 9. 怎样使用类和对象

9.3. 调用构造函数和析构函数的顺序

9.3.1. 构造函数与析构函数的调用时机

9.3.2. 相同性质的多个对象的调用顺序

★ 构造函数按定义的顺序依次执行，析构函数按构造函数进栈的概念反向依次执行

```
#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" << hour <<endl;
}
Time::~~Time()
{
    cout << "Time End" << hour << endl;
}
```

```
Time t1(15), t2(16);
```

```
int main()
{
    cout << "main" <<endl;
}
```

?

t1, t2都是全局变量
构造: t1, t2
析构: t2, t1

§ 9. 怎样使用类和对象

9.3. 调用构造函数和析构函数的顺序

9.3.1. 构造函数与析构函数的调用时机

9.3.2. 相同性质的多个对象的调用顺序

★ 构造函数按定义的顺序依次执行，析构函数按构造函数进栈的概念反向依次执行

```
#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" << hour <<endl;
}
Time::~~Time()
{
    cout << "Time End" << hour << endl;
}
```

```
void fun()
{
    Time t1(15);
    static Time t2(16);
    cout << "fun" <<endl;
}
int main()
{
    cout << "main begin" <<endl;
    fun();
    cout << "main end" <<endl;
}
```

程序的运行结果:

?

t1, t2是不同性质的变量
不遵循栈规则

§ 9. 怎样使用类和对象

9.3. 调用构造函数和析构函数的顺序

9.3.1. 构造函数与析构函数的调用时机

9.3.2. 相同性质的多个对象的调用顺序

★ 构造函数按定义的顺序依次执行，析构函数按构造函数进栈的概念反向依次执行

```
#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" << hour <<endl;
}
Time::~~Time()
{
    cout << "Time End" << hour << endl;
}
```

```
int main()
{
    Time *t1, *t2;
    t2=new Time(16);
    t1=new Time(15);
    cout << "main begin" <<endl;
    delete t2;
    cout << "main end" <<endl;
    delete t1;
}
```

程序的运行结果：

?

动态申请的变量，按new的顺序调构造
按delete的顺序调析构，不遵循规则

§ 9. 怎样使用类和对象

9. 4. 对象数组

9. 4. 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. 4. 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. 怎样使用类和对象

9. 4. 对象数组

9. 4. 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

```
#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
调用一个参数的构造

```
#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();
}
```

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

§ 9. 怎样使用类和对象

9. 4. 对象数组

9. 4. 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. 怎样使用类和对象

9. 4. 对象数组

9. 4. 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();
}
```

10个元素:

t[0]的hour=1, minute=2, sec=3

t[1]的hour=4, minute=5, sec=0

t[2]-t[6]的hour为6-10, m/s为0

t[7]-t[9]的三个值全部为0

t[0] : 调用带三个参数的构造

t[1] : 两个

t[2]-t[6]: 一个

t[7]-t[9]: 零个

```
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. 怎样使用类和对象

9. 4. 对象数组

9. 4. 3. 对象数组的构造函数与析构函数的调用顺序

★ 构造函数从0到n-1，析构函数从n-1到0

```
#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" << hour <<endl;
}
Time::~Time()
{
    cout << "Time End" << hour << endl;
}
```

```
void fun()
{
    Time t[10]={1, 2, 3, 4, 5,
                6, 7, 8, 9, 10};
    cout << "fun" << endl;
}
int main()
{
    cout << "main begin" <<endl;
    fun();
    cout << "main end" <<endl;
}
```

程序的运行结果:

```
main begin
Time Begin1
Time Begin2
...
Time Begin10
fun
Time End10
Time End9
...
Time End1
main end
```

§ 9. 怎样使用类和对象

9.5. 对象指针

9.5.1. 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换成第7章的student
方法相同

§ 9. 怎样使用类和对象

9.5. 对象指针

9.5.1. 指向对象的指针

使用:

Time换成第7章的student
方法相同, 仅限public

```
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();  
};
```

Time类定义

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

指向简单变量的指针

t : t1对象的地址

*t : t1对象

(*) . hour ⇔ t->hour ⇔ t1.hour;

(*) . display() ⇔ t->display() ⇔ t1.display()

```
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

(*) . hour ⇔ t->hour ⇔ t2[0].hour

(*) . 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()

指向数组变量的指针

§ 9. 怎样使用类和对象

9.5. 对象指针

9.5.1. 指向对象的指针

9.5.2. 指向对象成员的指针

Time换成第7章的student
方法相同，仅限public

9.5.2.1. 指向对象的数据成员的指针

定义：数据成员的基本类型 *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. 怎样使用类和对象

9.5. 对象指针

9.5.2. 指向对象成员的指针

9.5.2.1. 指向对象的数据成员的指针

9.5.2.2. 指向对象的成员函数的指针

```
/* 第6章:指向函数的指针 */
#include <iostream>
using namespace std;
void fun()
{ cout << "fun()" << endl;
}
int main()
{ void (*p)();
  p=fun; //赋值, 正确
  p();  //调用, 正确
}
```

```
#include <iostream>
using namespace std;
class Time {
private:
  int hour;
public:
  Time() { //构造
    hour=0;
  }
  void display() { //打印
    cout << hour << endl;
  }
};
int main()
{ Time t1;
  void (*p)();
  p=t1.display; //赋值, 错误
  p();          //调用, 错误
}
```

```
#include <iostream>
using namespace std;
class Time {
private:
  int hour;
public:
  Time() { //构造
    hour=0;
  }
  void display() { //打印
    cout << hour << endl;
  }
};
int main()
{ Time t1;
  void (Time::*p)();
  p=&Time::display; //赋值, 正确
  (t1.*p)();        //调用, 正确
}
```

全局函数的指针:

- (1) 返回类型匹配
- (2) 形参表匹配

成员函数的指针:

- (1) 返回类型匹配
- (2) 形参表匹配
- (3) 类匹配

§ 9. 怎样使用类和对象

9.5. 对象指针

9.5.2. 指向对象成员的指针

9.5.2.1. 指向对象的数据成员的指针

9.5.2.2. 指向对象的成员函数的指针

定义：成员函数返回类型（类::*指针变量名）（形参表）

赋值：指针变量名 = &类::成员函数名

★ 对象的成员函数必须是public

使用：

（对象名.*指针变量名）（实参表）

```
Time t1, t2;  
void (Time::*p)();  
p=&Time::display;  
(t1.*p)() ⇔ t1.display()  
(t2.*p)() ⇔ t2.display()  
(t1.p)(); //错误, t1无p成员
```

§ 9. 怎样使用类和对象

9.5. 对象指针

9.5.1. 指向对象的指针

9.5.2. 指向对象成员的指针

9.5.3. 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. 怎样使用类和对象

9.5. 对象指针

9.5.3. 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 *abc)
{ cout << this->hour << abc->minute
 << this->sec << endl;
}

§ 9. 怎样使用类和对象

9.6. 共用数据的保护

9.6.1. 引入及含义

一个数据可以通过不同的方式进行共享访问，因此可能导致数据因为误操作而改变，为了达到既能共享，又不会因误操作而改变，引入共用数据保护的概念

<pre>void fun(int *p) { *p=10; } int main() { int k=15; fun(&k); }</pre>	通过 指针 ，fun中改变了main的局部变量k的值	<pre>void fun(int &p) { p=10; } int main() { int k=15; fun(k); }</pre>	通过 引用 ，fun中改变了main的局部变量k的值
--	-----------------------------------	--	-----------------------------------

§ 9. 怎样使用类和对象

9.6. 共用数据的保护

9.6.1. 引入及含义

9.6.2. 常对象与常对象成员

常对象：

const 类名 对象名(初始化实参表)

或 类名 const 对象名(初始化实参表)

const Time t1(15);

Time const t2(16, 30, 0);

T1始终是15:00:00

T2始终是16:30:00

- ★ 与第2章中的常变量含义相同，在整个程序的执行过程中值不可再变化
- ★ 必须在定义时进行初始化
- ★ 不能调用普通成员函数 (即使不改变数据成员的值)

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

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

int main()
{
    const Time t1(15);
    Time const t2(16, 30, 0);
    t1.minute = 12; //编译报错
    t2.sec = 27;    //编译报错
    t1.display();  //编译报错
}
```

§ 9. 怎样使用类和对象

9.6. 共用数据的保护

9.6.1. 引入及含义

9.6.2. 常对象与常对象成员

常对象成员：

常对象中的所有数据成员在程序执行过程中值均不可变，如果只需要限制部分成员的值在执行过程中不可变，则需要引入常对象成员的概念

- 常数据成员：该数据成员的值在执行中不可变
- 常成员函数：该函数只能引用成员的值，不能修改

常数据成员：

```
class 类名 {  
    const 数据类型 数据成员名  
    或 数据类型 const 数据成员名  
};  
  
class Time {  
    private:  
        const int hour;  
        int const minute;  
        int sec;  
};
```

常成员函数：

```
class 类名 {  
    返回类型 成员函数名(形参表) const;  
};  
  
class Time {  
    public:  
        void display() const;  
};
```

§ 9. 怎样使用类和对象

9.6. 共用数据的保护

9.6.2. 常对象与常对象成员

使用:

★ 常数据成员要在构造函数中初始化, 使用中值不可变, 在构造函数中初始化时, 必须用参数初始化表形式, 而不能用赋值形式

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

class Time {
public:
    const int hour;
    int minute, sec;
    Time(int h=0, int m=0, int s=0)
    {
        hour    = h;
        minute = m;
        sec     = s;
    }
};

int main()
{
    Time t1;
    t1.hour = 10; //错误
}
```

错误

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

class Time {
public:
    const int hour;
    int minute, sec;
    Time(int h=0, int m=0, int s=0) : hour(h)
    {
        minute = m;
        sec     = s;
    }
};

int main()
{
    Time t1;
    t1.hour = 10; //错误
}
```

正确

§ 9. 怎样使用类和对象

9.6. 共用数据的保护

9.6.2. 常对象与常对象成员

使用:

★ 常数据成员要在构造函数中初始化, 使用中值不可变, 在构造函数中初始化时, 必须用参数初始化表形式, 而不能用赋值形式

★ 常成员函数只能引用类的数据成员 (无论是常数据成员) 的值, 而不能修改数据成员的值

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

class Time {
public:
    int hour, minute, sec;
    void set(int h=0,int m=0,int s=0)
    {   hour    = h;
        minute = m;
        sec     = s;
    }
};

int main()
{   Time t1;
    t1.set(14, 15, 23);
}
```

正确

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

class Time {
public:
    int hour, minute, sec;
    void set(int h=0,int m=0,int s=0) const
    {   hour    = h;
        minute = m;
        sec     = s;
    }
};

int main()
{   Time t1;
    t1.set(14, 15, 23);
}
```

错误

§ 9. 怎样使用类和对象

9.6. 共用数据的保护

9.6.2. 常对象与常对象成员

使用:

★ 常成员函数写成下面形式, 编译不报错但不起作用

const 返回类型 成员函数名(形参表)

或 返回类型 const 成员函数名(形参表)

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

赋值正确, 说明
set不是常成员函数

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

int main()
{
    Time t1;
    t1.set(14, 15, 23);
}
```

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

赋值正确, 说明
set不是常成员函数

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

int main()
{
    Time t1;
    t1.set(14, 15, 23);
}
```

§ 9. 怎样使用类和对象

9.6. 共用数据的保护

9.6.2. 常对象与常对象成员

使用:

★ 常成员函数可以调用本类的另一个常成员函数，但不能调用本类的非常成员函数
(即使该非常成员函数不修改数据成员的值)

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

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

int main()
{
    Time t1;
    t1.fun();
}
```

错误

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

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

int main()
{
    Time t1;
    t1.fun();
}
```

正确

§ 9. 怎样使用类和对象

9.6. 共用数据的保护

9.6.2. 常对象与常对象成员

使用:

★ 若希望常成员函数能强制修改数据成员, 则要将数据成员定义为mutable

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

class Time {
public:
    int hour, minute, sec;
    void set(int h=0, int m=0, int s=0) const
    {
        hour   = h;
        minute = m;
        sec    = s;
    }
};

int main()
{
    Time t1;
    t1.set(14, 15, 23);
}
```

错误

```
#include <iostream>
using namespace std;
class Time {
public:
    mutable int hour, minute, sec;
    void set(int h=0, int m=0, int s=0) const
    {
        hour   = h;
        minute = m;
        sec    = s;
    }
};

int main()
{
    Time t1;
    t1.set(14, 15, 23);
}
```

正确

```
#include <iostream>
using namespace std;
class Time {
public:
    int mutable hour, minute, sec;
    void set(int h=0, int m=0, int s=0) const
    {
        hour   = h;
        minute = m;
        sec    = s;
    }
};

int main()
{
    Time t1;
    t1.set(14, 15, 23);
}
```

正确

§ 9. 怎样使用类和对象

9.6. 共用数据的保护

9.6.2. 常对象与常对象成员

使用:

★ 若定义对象为常对象, 则只能调用其中的常成员函数(不能修改数据成员的值), 而不能调用其中的普通成员函数(即使该成员不修改数据成员的值)

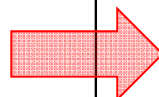
```
class Time {  
    public:  
        int hour, minute, sec;  
        Time(int h=0, int m=0, int s=0)  
        {   hour = h;   minute = m;   sec = s;  
        }  
        void display()  
        {   cout << hour << minute << sec << endl;  
        }  
};  
int main()  
{   const Time t1(13, 14, 23);  
    t1.minute = 12; //编译报错  
    t1.display();   //编译报错  
}
```

本节开始的例子

```
class Time {  
    public:  
        int hour, minute, sec;  
        Time(int h=0, int m=0, int s=0)  
        {   hour = h;   minute = m;   sec = s;  
        }  
        void display() const  
        {   cout << hour << minute << sec << endl;  
        }  
};  
int main()  
{   const Time t1(13, 14, 23);  
    t1.display();   //正确  
}
```

```
class Time {  
    public:  
        int hour, minute, sec;  
        Time(int h=0, int m=0, int s=0)  
        {   hour = h;   minute = m;   sec = s;  
        }  
        void display() const  
        {   cout << hour << minute << sec << endl;  
            sec++; //错误  
        }  
};  
int main()  
{   const Time t1(13, 14, 23);  
    t1.display();  
}
```

问: 如果想在display中
修改sec的值?



```
class Time {  
    public:  
        int hour, minute;  
        mutable int sec;  
        Time(int h=0, int m=0, int s=0)  
        {   hour = h;   minute = m;   sec = s;  
        }  
        void display() const  
        {   cout << hour << minute << sec << endl;  
            sec++; //正确  
        }  
};  
int main()  
{   const Time t1(13, 14, 23);  
    t1.display();  
}
```

§ 9. 怎样使用类和对象

9.6. 共用数据的保护

9.6.2. 常对象与常对象成员

使用:

★ 不能定义构造/析构函数为常成员函数

★ 全局函数不能定义const

```
void fun() const    //编译报错
{
    return;
}
int main()
{
    fun();
}
```

§ 9. 怎样使用类和对象

9.6. 共用数据的保护

9.6.2. 常对象与常对象成员 (P. 279 表9.1 引伸)

	普通数据成员	const 数据成员	mutable 数据成员	普通成员函数	const 成员函数
普通对象	读写	读	读写	可调用	可调用
const对象	读	读	读写	不可调用	可调用
普通成员函数	读写	读	读写	可调用	可调用
const成员函数	读	读	读写	不能调用	可调用

§ 9. 怎样使用类和对象

9.6. 共用数据的保护

9.6.3. 指向对象的常指针

⇔ 6.8.2. 常指针

9.6.4. 指向常对象的指针变量

⇔ 6.8.1. 指向常量的指针变量

9.6.5. 对象的常引用

常指针与常引用基本类似

另:6.8.3. 指向常量的常指针(6.8.1+6.8.2)也适用对象

§ 9. 怎样使用类和对象

9.7. 对象的动态建立和释放

C语言方法: `Time *p1, *p2;`

```
    申请: p1 = (Time *)malloc(sizeof(Time));  
          p2 = (Time *)malloc(10*sizeof(Time));  
          if (p1==NULL) { ... }  
          if (p2==NULL) { ... }  
    释放: free(p1);  
          free(p2);
```

★ C++中一般不建议使用C方法动态申请

- C方式动态内存申请和释放时不会调用构造和析构函数
- 第7章例题中, struct中有string类, 则malloc/free会出错

C++方法: `Time *p1, *p2;`

```
    申请: p1 = new(nothrow) Time;  
          p2 = new(nothrow) Time[10];  
          if (p1==NULL) { ... }  
          if (p2==NULL) { ... }  
    释放: delete p1;  
          delete []p2;
```

★ C++中delete时, 只要是数组, 必须加[]

§ 9. 怎样使用类和对象

9.7. 对象的动态建立和释放

C++方法: `Time *p1, *p2;`

申请: `p1 = new(nothrow) Time;`
`p2 = new(nothrow) Time[10];`
`if (p1==NULL) { ... }`
`if (p2==NULL) { ... }`

释放: `delete p1;`

`delete []p2;`

★ C++中delete时, 只要是数组, 必须加[]

§ 7. 用户自定义数据类型

7.2.10.2.2. 用delete释放空间

★ 普通变量: `int *p=new int;`
`delete p;`

★ 一维数组: `char *name=new char[10];`
`delete []name;`

● 某些资料上说可以 `delete name`, 因为一维数组可理解为数组首元素地址, 不加[]

● 对int/char等基本类型的数组, 加不加均正确, 错误例子见第9章

★ 二维数组: `float (*f)[4]=new float[3][4];`
`delete []f;`

● 二维以上必须加一个[], 否则编译警告

=> 四个例子, 哪个有错???

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

int main()
{
    int *p = new int[10];
    delete p;
    return 0;
}
```

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

int main()
{
    int *p = new int[10];
    delete []p;
    return 0;
}
```

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

class Time {
private:
    int hour, minute, second;
public:
    Time(int h=0, int m=0, ints=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;
}

int main()
{
    Time *t1 = new Time[10];
    delete t1;
    return 0;
}
```

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

class Time {
private:
    int hour, minute, second;
public:
    Time(int h=0, int m=0, ints=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;
}

int main()
{
    Time *t1 = new Time[10];
    delete []t1;
    return 0;
}
```

§ 9. 怎样使用类和对象

9.8. 对象的赋值与复制

9.8.1. 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
- ★ 将对象2的全部数据成员的值对应赋给对象1的全部数据成员，不包括成员函数
(理解为整体内存拷贝，可参考memcpy函数)
- ★ 若对象数据成员是指针及动态分配的数据，则可能导致不可预料的后果

§ 9. 怎样使用类和对象

9.8. 对象的赋值与复制

9.8.1. 对象的赋值

★ 若对象数据成员是指针及动态分配的数据，则可能导致不可预料的后果

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
class test {
private:
    int a;
    int b;
    char c[20];
public:
    test(const char *s="A") {
        a=0; b=0;
        strcpy(c, s);
    }
    void set(const char *s) {
        strcpy(c, s);
    }
    void display() {
        cout << c << endl;
    }
};
int main()
{
    test t1("hello"), t2;
    t1.display();
    t2.display();
    t2=t1;
    t2.display();
    t1.set("china");
    t1.display();
    t2.display();
}
```

无动态内存申请
执行正确

带缺省参数的构造函数

重新为c赋值

打印c的内容

2000	0
	0
t1	hello \0 14个 后续
2027	

2100	0
	0
t2	A\0 18个 后续
2127	

2000	0
	0
t1	hello \0 14个 后续
2027	

28字节
整体拷贝

2100	0
	0
t2	A\0 18个 后续
2127	

2000	0
	0
t1	china \0 14个 后续
2027	

2100	0
	0
t2	hello \0 14个 后续
2127	

§ 9. 怎样使用类和对象

9.8. 对象的赋值与复制

9.8.1. 对象的赋值

★ 若对象数据成员是指针及动态分配的数据，则可能导致不可预料的后果

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
class test {
private:
    int a;
    int b;
    char *c;
public:
    test(const char *s="A") {
        a=0; b=0;
        c = new char[20];
        strcpy(c, s);
    }
    void set(const char *s) {
        strcpy(c, s);
    }
    void display() {
        cout << c << endl;
    }
};

int main()
{
    test t1("hello"), t2;
    t1.display();
    t2.display();
    t2=t1;
    t2.display();
    t1.set("china");
    t1.display();
    t2.display();
}
```

有动态内存申请
执行结果错误

注意：目前程序不完整
仅在构造函数中动态
申请，未定义析构函
数进行释放

hello
A
hello
china
china

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
class test {
private:
    int a;
    int b;
    char *c;
public:
    test(const char *s="A") {
        a=0; b=0;
        c = new char[20];
        strcpy(c, s);
    }
    ~test() {
        delete c;
    }
    void set(const char *s) {
        strcpy(c, s);
    }
    void display() {
        cout << c << endl;
    }
};

int main()
{
    test t1("hello"), t2;
    t1.display();
    t2.display();
    t2=t1;
    t2.display();
    t1.set("china");
    t1.display();
    t2.display();
}
```

有动态内存申请
执行结果错误

同左例，加入析构函数后，
不但执行结果错误，而且
死机 (VS2015/VC++6.0)
CodeBlocks不死机但不代
表没错

hello
A
hello
china
china

错误原因的图解及具体解释:

1、造成4000-4019这20个字节的内存丢失

2、t1/t2的c成员同时指向一块内存, 通过t1的c修改改内存块, 会导致t2的c值同时改变

3、若定义了析构函数, 则main函数执行完成后系统会调用析构函数(按t2, t1的顺序), t2调用析构函数释放3000-3019后, 再调用t1的析构函数会导致重复释放3000-3019, 错!!!

如何保证有动态内存时的赋值正确性?

第10章 重载=运算符

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
class test {
private:
    int a;
    int b;
    char *c;
public:
    test(const char *s="A") {
        a=0; b=0;
        c = new char[20];
        strcpy(c, s);
    }
    ~test() {
        delete c;
    };
    void set(const char *s) {
        strcpy(c, s);
    }
    void display() {
        cout << c << endl;
    }
};
int main()
{
    test t1("hello"), t2;
    t1.display();
    t2.display();
    t2=t1;
    t2.display();
    t1.set("china");
    t1.display();
    t2.display();
}
```

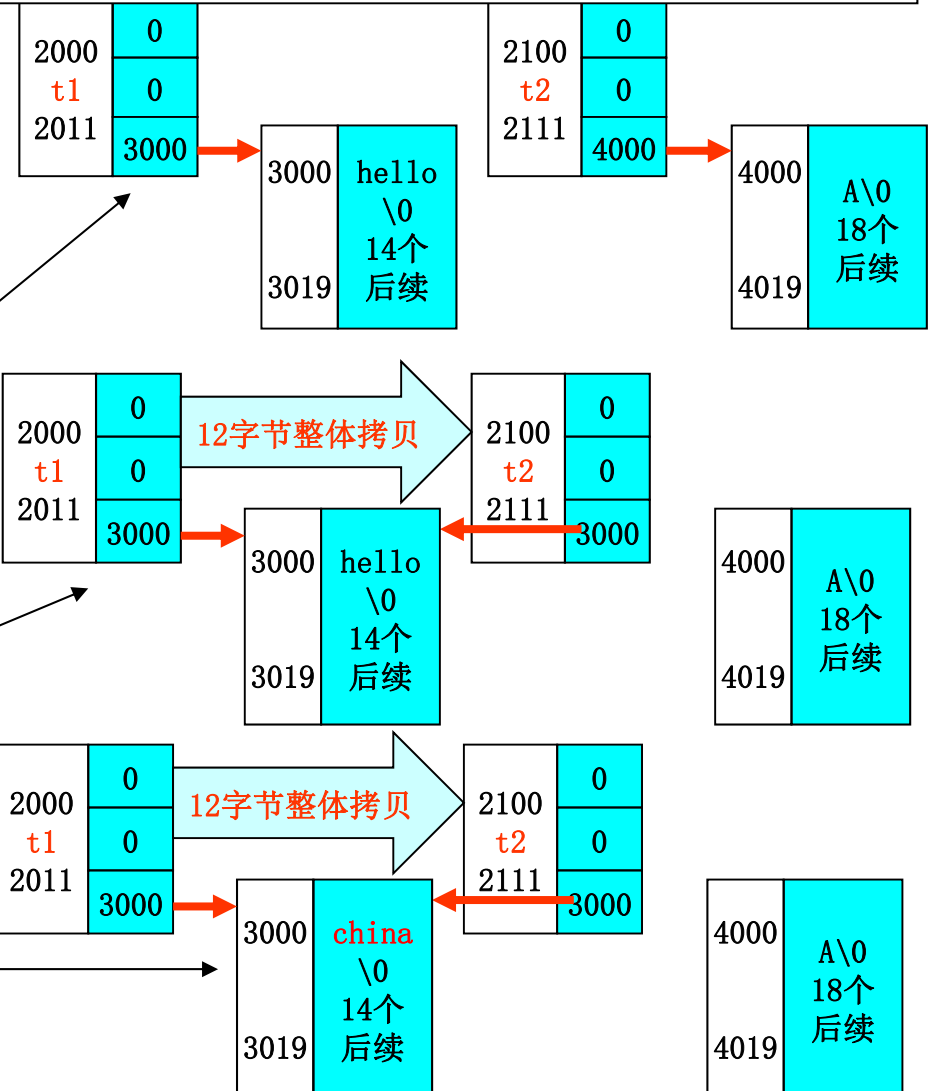
有动态内存申请
执行结果错误

同左例, 加入析构函数后,
不但执行结果错误, 而且
死机 (VS2015/VC++6.0)
CodeBlocks不死机但不代表
没错

hello
A

hello

china
china



§ 9. 怎样使用类和对象

9.8. 对象的赋值与复制

9.8.2. 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

对象复制的实现：建立新对象时自动调用复制构造函数（也称为拷贝构造函数）

复制构造函数：

类名(const 类名 &引用名)

★ 用一个对象的值去初始化另一个对象

★ 若不定义复制构造函数，则系统自动定义一个，参数为const型引用，函数体为对应成员内存拷贝

★ 若定义了复制构造函数，则系统缺省定义消失

★ 允许体内实现或体外实现

★ 复制构造函数和普通构造函数（可能多个）的地位平等，调用其中一个后就不再调用其它构造函数

§ 9. 怎样使用类和对象

9.8. 对象的赋值与复制

9.8.2. 对象的复制

```
#include <iostream>
using namespace std;
class Time {
private:
    int hour, minute, second; //三个私有成员
public:
    Time(int h=0, int m=0, int s=0) { //构造, 适合0-3参
        hour = h; minute = m; second = s;
    }
    void display() { //打印
        cout<<hour<<":"<<minute<<":"<<second<<endl;
    }
};

int main()
{
    Time t1(14, 15, 23), t2(t1), t3=t1;
    t1.display();
    t2.display();
    t3.display();
}
```

```
14:15:23
14:15:23
14:15:23
```

问题: $t2(t1)$ 、 $t3=t1$ 匹配哪个构造函数?

现有的 $\text{Time}(\text{int } h=0, \text{int } m=0, \text{int } s=0)$
可用于一个整数参数, 例如: $\text{Time } t1(15)$,
难道.....?

匹配了缺省的复制构造函数, 相当于:
 $\text{Time}(\text{int } h=0, \text{int } m=0, \text{int } s=0);$
 $\text{Time}(\text{const Time } \&);$ //缺省为拷贝
12 字节

这两个构造函数重载即使都是一个参数,
也可以区分: $t2(t1)$
 $t2(14)$

★ 复制构造函数和普通构造函数(可能多个)的地位平等, 调用其中一个后就不再调用其它构造函数

```
//本例中复制构造函数的显式定义
Time(const Time &t);
//本例中复制构造函数的体外实现
Time::Time(const Time &t)
{
    hour   = t.hour;
    minute = t.minute;
    sec    = t.sec;
}
```

```
class Time {
    ...
public:
    Time(int h=0);
    Time(int h, int m, int s=0);
    Time(const Time &t);
};

int main()
{
    Time t1;
    Time t2(10);
    Time t3(1, 2, 3);
    Time t4(4, 5);
    Time t5(t2);
    Time t6 = t4;
}
```

§ 9. 怎样使用类和对象

9.8. 对象的赋值与复制

9.8.2. 对象的复制

复制构造函数的调用时机：

- ★ 用已有对象初始化一个新建立的对象时
- ★ 函数形参为对象，实参向形参进行单向传值时
- ★ 函数的返回类型是对象时
- ★ 不包括执行语句中的赋值(=)操作，执行赋值(=)操作通过赋值运算符(=)的重载来实现(第10章)
- ★ 除非有动态内存申请或其它特殊功能，否则不需要定义复制构造函数

§ 9. 怎样使用类和对象

9.8. 对象的赋值与复制

9.8.2. 对象的复制

本例仅用于证明复制构造函数的调用时机，无实际及具体含义

★ 用VS2015编译运行

★ 用VC++6.0编译运行

★ 用CodeBlocks编译运行

```
#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);
    Time(const Time &t); //复制构造函数
    void display();
};

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

Time::Time(const Time &t)
{
    hour   = t.hour   - 1;
    minute = t.minute - 1;
    sec    = t.sec    - 1;
    cout << "复制构造" << endl;
}

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

```
void main()
{
    Time t1(14, 15, 23), t2(t1);
    t2.display();
}
```

用对象初始化新对象

复制构造
13:14:22

```
void fun(Time t)
{
    t.display();
}
```

函数形参为对象

```
void main()
{
    Time t1(14, 15, 23);
    fun(t1);
}
```

复制构造
13:14:22

```
Time fun()
{
    Time t1(14, 15, 23);
    return t1;
}

void main()
{
    Time t2 = fun();
    t2.display();
}
```

函数返回值为对象

为什么是两次？

VS2015

复制构造
13:14:22

CodeBlocks
14:15:23

VC++6.0:

复制构造
复制构造
12:13:21

§ 9. 怎样使用类和对象

9.8. 对象的赋值与复制

9.8.2. 对象的复制

本例仅用于证明复制构造函数的调用时机，无实际及具体含义

★ 用VS2015编译运行

★ 用VC++6.0编译运行

★ 用CodeBlocks编译运行

```
#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);
    Time(const Time &t); //复制构造函数
    void display();
};

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

Time::Time(const Time &t)
{
    hour   = t.hour   - 1;
    minute = t.minute - 1;
    sec    = t.sec    - 1;
    cout << "复制构造" << endl;
}

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

```
Time fun()
{
    Time t1(14, 15, 23);
    return t1;
}
```

VC++6.0 为什么是一次？

```
void main()
{
    Time t2;
    t2 = fun();
    t2.display();
}
```

VS2015

复制构造

13:14:22

CodeBlocks

14:15:23

VC++6.0:

复制构造

13:14:22

MinGW为什么一次也没有：

编译时采用了NRV (Named Return value) 技术，编译时去掉复制构造函数，提高运行速度，目前不是标准，且有可能有隐患

VC++6.0为什么是两次：

return时一次，函数返回值定义时赋初值给t2一次

VS2015为什么是一次：

自行研究

```
Time fun()
{
    Time t1(14, 15, 23);
    return t1;
}
```

函数返回值为对象

VC++6.0为什么是两次？

```
void main()
{
    Time t2 = fun();
    t2.display();
}
```

VS2015

复制构造

13:14:22

CodeBlocks

14:15:23

VC++6.0:

复制构造

复制构造

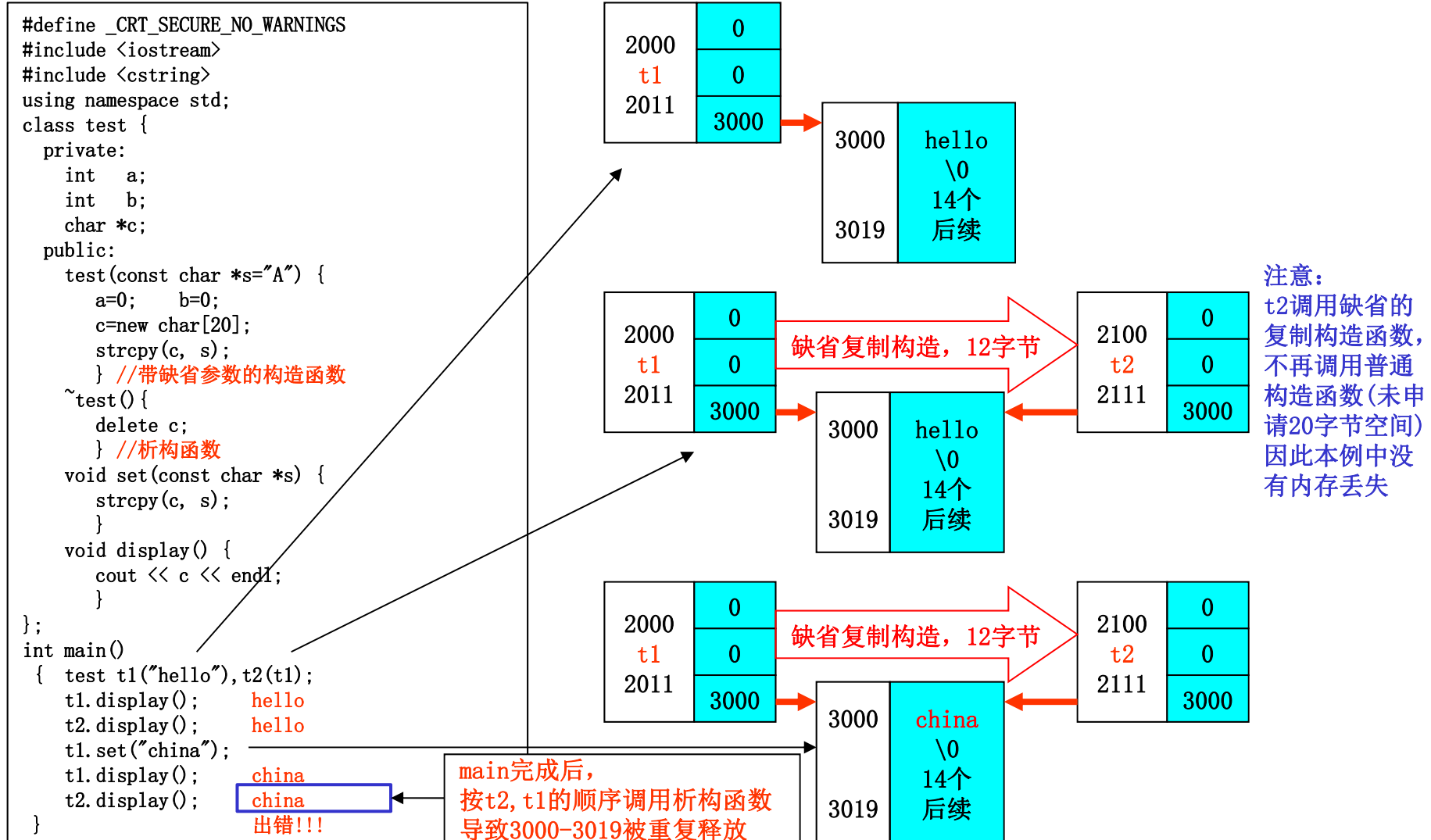
12:13:21

§ 9. 怎样使用类和对象

9.8. 对象的赋值与复制

9.8.2. 对象的复制

★ 除非有动态内存申请或其它特殊功能，否则不需要定义复制构造函数



§ 9. 怎样使用类和对象

注意：无法解决赋值问题
`t2 = t1;`

赋值仍然会错，具体要用
第10章的运算符重载来解决

9.8. 对象的赋值与复制

9.8.2. 对象的复制

★ 除非有动态内存申请或其它特殊功能，否则不需要定义复制构造函数

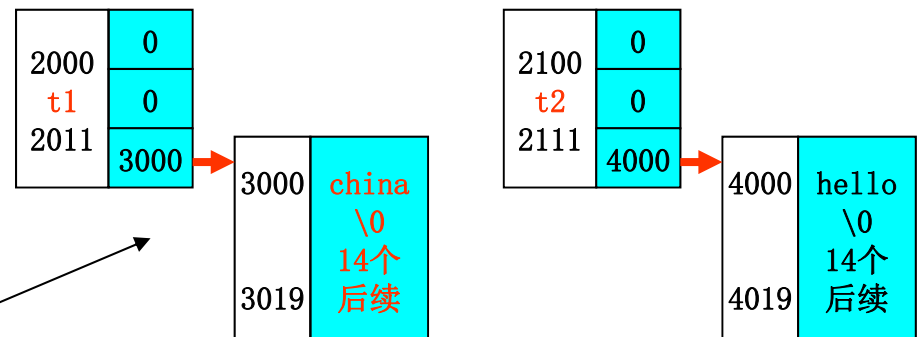
有动态内存申请的情况下，不能使用系统缺省的复制构造函数，必须自己定义

```
class test {
private:
    int  a;
    int  b;
    char *c;
public:
    test(char *s="A") {
        a=0;  b=0;
        c=new char[20];
        strcpy(c, s);
    } //带缺省参数的构造函数
    test(const test &t); //复制构造函数的声明
    ~test() {
        delete c;
    } //析构函数
    void set(char *s) {
        strcpy(c, s);
    }
    void display() {
        cout << c << endl;
    }
};

test::test(const test &s) //复制构造的体外实现
{
    a=s.a;
    b=s.b;
    c=new char[20];
    strcpy(c, s.c);
}
```

仔细体会两者实现的区别

```
int main()
{
    test t1("hello"), t2(t1);
    t1.display();           hello
    t2.display();           hello
    t1.set("china");
    t1.display();           china
    t2.display();           hello
}
```



```
//系统缺省的构造函数实现方法
test(const test &s)
{
    a=s.a;
    b=s.b;
    c=s.c;
}
```

实质上是执行系统函数 `memcpy`:
`memcpy(this, &s, sizeof(test));`
相当于一次直接复制了12个字节

§ 9. 怎样使用类和对象

9.9. 静态成员

9.9.1. 引入

希望在同一个类的多个对象间实现数据共享

- ★ 一个对象修改，另一个对象访问得到修改后的值
- ★ 类似与全局变量的概念，但属于类，仅供该类的不同对象间共享数据

9.9.2. 静态数据成员

```
定义: class 类名 {  
    private/public:  
        static 数据类型 成员名;  
        ...  
};
```

§ 9. 怎样使用类和对象

9.9. 静态成员

9.9.2. 静态数据成员

使用:

★ 静态数据成员不属于任何一个对象，不在对象中占用空间，单独在静态数据区分配空间(初值为0，不随对象的释放而释放)，一个静态数据成员只占有一个空间，所有对象均可共享访问

★ 静态数据成员同样受类的作用域限制

★ 静态数据成员必须进行初始化，初始化位置在类定义体后，函数体外进行(此时不受类的作用域限制)

数据类型 类名::静态数据成员名=初值;

★ 虽然可以通过this指针访问，但是数据不在一起

```
#include <iostream>
using namespace std;
class Time {
private:
    static int hour;
    int minute, sec;
public:
    Time(int h=0, int m=0, int s=0)
    { minute = m; sec = s; }
    void display()
    { cout << hour << minute << sec << endl; }
};
int Time::hour = 5; //虽然是private, 可以
int main()
{ Time t1;
  cout << sizeof(Time) << endl;      8
  t1.display();                      500
}
```

```
#include <iostream>
using namespace std;
class Time {
private:
    static int hour;
    int minute, sec;
public:
    Time(int h=0, int m=0, int s=0)
    { minute = m; sec = s; }
    void display()
    { cout << this->hour << minute << sec << endl; }
};
int Time::hour = 5; //虽然是private, 可以
int main()
{ Time t1;
  cout << sizeof(Time) << endl;      8
  t1.display();                      500
}
```

§ 9. 怎样使用类和对象

9.9. 静态成员

9.9.2. 静态数据成员

使用：

★ 不能通过参数初始化表进行初始化，但可以通过赋值方式初始化

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

class Time {
    private:
        static int hour;
        int 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 Time::hour = 5; //必须要有，否则编译错

int main()
{
    Time t1;
    cout << sizeof(Time) << endl;
    t1.display();
}
```

错误

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

class Time {
    private:
        static int hour;
        int 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 Time::hour = 5; //必须要有，否则编译错

int main()
{
    Time t1;
    cout << sizeof(Time) << endl;
    t1.display();
}
```

正确

§ 9. 怎样使用类和对象

9.9. 静态成员

9.9.2. 静态数据成员

使用:

- ★ 不能通过参数初始化表进行初始化
- ★ 既可以通过类型引用, 也可以通过对象名引用

```
int main()
{
    Time t1(14, 15, 23), t2;
    t1.display();    0:15:23
    t2.display();    0:0:0
    t1.hour = 8;     //对象名引用
    t1.display();    8:15:23
    t2.display();    8:0:0
    Time::hour = 19; //类型引用
    t1.display();    19:15:23
    t2.display();    19:0:0
}
```

为什么不是14:15:23 ?

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

class Time {
public:
    static int hour;
    int minute;
    int sec;
public:
    Time();
    Time(int h, int m, int s);
    void display();
};

int Time::hour = 10;

Time::Time()
{
    hour=0; minute=0; sec=0;
}

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

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

- ★ 静态数据成员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

§ 9. 怎样使用类和对象

9.9. 静态成员

9.9.3. 静态成员函数

定义: `class` 类名 {
 private/public:
 static 返回类型 函数名(形参表);
 ...
};

调用:

类名::成员函数名(实参表);

任意对象名.成员函数名(实参表);

使用:

★ 没有this指针, 不属于某个对象

普通成员函数:

Time t1;

t1.display() ⇔ t1.display(&t1);

静态成员函数:

无

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

class test {
    public:
        static void fun(int x)
        {
            cout << x << endl;
        }
};

int main()
{
    test t1;
    t1.fun(10);    //10
    test::fun(15); //15
}
```

§ 9. 怎样使用类和对象

9.9. 静态成员

9.9.3. 静态成员函数

使用:

- ★ 没有this指针, 不属于某个对象
- ★ 允许体内实现或体外实现
- ★ 静态成员函数中可以直接访问静态数据成员

```
#include <iostream>
using namespace std;
class test {
    private:
        static int a;
    public:
        static void fun()
        {   cout << this->a << endl;
        }
};
int test::a = 10;
int main()
{   test t1;
    t1.fun();    //10
    test::fun();//10
}
```

无this指针
报错!!!

体内实现

```
#include <iostream>
using namespace std;
class test {
    private:
        static int a;
    public:
        static void fun()
        {   cout << a << endl;
        }
};
int test::a = 10;
int main()
{   test t1;
    t1.fun();    //10
    test::fun();//10
}
```

体外实现

```
#include <iostream>
using namespace std;
class test {
    private:
        static int a;
    public:
        static void fun();
};
int test::a = 10;
void test::fun() //此处不能static
{   cout << a << endl;
}
int main()
{   test t1;
    t1.fun();    //10
    test::fun();//10
}
```


§ 9. 怎样使用类和对象

9.9. 静态成员

P. 293-296 例9.11及说明

9.9.3. 静态成员函数

使用:

★ 在静态成员函数中不能对非静态数据成员进行直接访问，而要通过对象参数的方式
(不提倡，建议静态成员函数只访问静态数据成员)

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

class test {
private:
    static int a;
    int b;
public:
    static void fun()
    {
        a=10; //正确
        b=11; //错误，因为没有this指针，不知道
              //应该访问那个对象的b成员
    }
};

int test::a = 5;

int main()
{
    test t1;
    t1.fun();
    test::fun();
}
```

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

class test {
private:
    static int a;
    int b;
public:
    static void fun(test &t)
    {
        a=10;    //正确
        t.b=11;  //正确
    }
};

int test::a = 5;

int main()
{
    test t1, t2;
    t1.fun(t1);
    test::fun(t2);
}
```

§ 9. 怎样使用类和对象

9. 10. 友元

9. 10. 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

<p>问题：在全局函数display(外部)中如何访问私有成员？</p> <pre>class Time { private: int hour; int minute; int sec; public: ... }; void display(Time t) { 想访问 t.hour; }</pre>	<p>方法1：通过公有函数间接访问</p> <pre>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 }</pre> <p>当频繁调用时，效率较低</p>	<p>方法2：成员直接公有</p> <pre>class Time { public: int hour; int minute; int sec; public: ... }; void display(Time t) { 直接 t.hour; }</pre> <p>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</p>
---	---	--

§ 9. 怎样使用类和对象

9. 10. 友元

9. 10. 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. 怎样使用类和对象

9.10. 友元

9.10.2. 声明全局函数为友元函数

<pre>class Time { private: int hour; int minute; int sec; friend void display(Time &t); public: ... };</pre>	
<pre>void display(Time &t) { cout << t.hour ; ✓ }</pre>	<pre>void fun(Time &t) { cout << t.hour ; ✗ }</pre>
<pre>void main() { Time t1; display(t1); }</pre>	

全局函数

★ 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. 怎样使用类和对象

9. 10. 友元

9. 10. 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);  
};
```

声明友元不限定private/public
但友元函数所在类要符合限定规则

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

//成员. 对象方式访问

§ 9. 怎样使用类和对象

9. 10. 友元

9. 10. 3. 声明其它类的成员函数为友元函数

类的提前声明：(P. 299 倒数5-3行，不够完善)

- ★ 在提前声明到实际的类的声明出现之间，仅允许以对象、对象的引用、对象的指针等对象的**整体形式**声明友元函数的形参及返回类型，但不能出现任何类的成员形式(包括数据成员和成员函数)(因为此时类有多少个成员，成员的类型等均未知)
- ★ 若友元函数的函数体实现中仅包含对象的引用或指针的**整体形式**，则允许出现在提前声明到实际的类的声明出现之间，否则必须放在实际的类的声明出现之后

请参考第7章 7. 2. 10 结构体在不同位置定义时的使用的相关内容并进行对比

§ 9. 怎样使用类和对象

9. 10. 友元

9. 10. 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);  
};
```

```
void display(Time *t);  
void display(Time t);  
Time display(Time &t);  
Time& display(Time t);  
Time* display(Time *t);
```

都正确，对应修改

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

声明友元不限定private/public
但友元函数所在类要符合限定规则

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

//成员. 对象方式访问

§ 9. 怎样使用类和对象

9.10. 友元

9.10.3. 声明其它类的成员函数为友元函数

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

错误, 因为现在还不知道
test类有display成员函数
即使成员函数不占空间

§ 9. 怎样使用类和对象

9.10. 友元

9.10.3. 声明其它类的成员函数为友元函数

```
class Time;
class test {
public:
    Time* fun(void) {
        Time *t1;
        return t1;
    }
};

class Time {
private:
    int hour;
    ...
    friend Time* test::fun(void);
};
```

正确，因为函数体实现中仅出现指针，
未访问任何成员
(整个函数的实现无意义，不讨论)

§ 9. 怎样使用类和对象

9.10. 友元

9.10.3. 声明其它类的成员函数为友元函数

```
class Time;  
class test {  
    public:  
        Time fun(void) {  
            Time *t1;  
            return *t1;  
        }  
};
```

错误，因为 return *t1 时不知道
基类型的大小
(整个函数的实现无意义，不讨论)

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

§ 9. 怎样使用类和对象

9.10. 友元

9.10.3. 声明其它类的成员函数为友元函数

```
class Time;  
class test {  
    public:  
        Time& fun(Time t) {  
            int a=11+13;  
            return t;  
        }  
};
```

错误，因为 return t 时t不知道大小

(整个函数的实现无意义，不讨论)

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

§ 9. 怎样使用类和对象

9.10. 友元

9.10.3. 声明其它类的成员函数为友元函数

```
class Time;  
class test {  
    public:  
        Time fun(Time &t) {  
            int a = 10+2;  
            return t;  
        }  
};
```

错误，因为 return t 时要调用缺省复制构造函数(内存拷贝)，但不知道大小
(整个函数的实现无意义，不讨论)

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

§ 9. 怎样使用类和对象

9.10. 友元

9.10.3. 声明其它类的成员函数为友元函数

```
class Time;  
class test {  
    public:  
        Time& fun(Time &t) {  
            int a = 10+2;  
            return t;  
        }  
};
```

正确，因为 t 是引用，不占空间
return t 时也不调用缺省构造函数
(整个函数的实现无意义，不讨论)

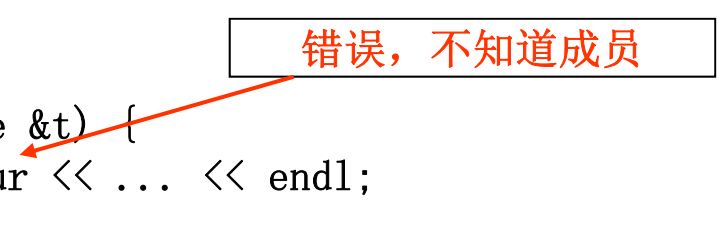
```
class Time {  
    private:  
        int hour;  
        ...  
        friend Time& test::fun(Time &t);  
};
```

§ 9. 怎样使用类和对象

9.10. 友元

9.10.3. 声明其它类的成员函数为友元函数

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



§ 9. 怎样使用类和对象

9.10. 友元

9.10.3. 声明其它类的成员函数为友元函数

```
class Time;  
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;  
}
```

正确，在Time后体外实现

§ 9. 怎样使用类和对象

9.10. 友元

9.10.4. 友元类

★ 提前声明遵循刚才的原则

```
class test:
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

★ 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 (num > s.num) {...}
}
```


§ 9. 怎样使用类和对象

9. 11. 类模板

9. 11. 1. 函数模板 (4. 7的内容)

§ 4. 利用函数实现指定的功能

4.9. 函数模板 (C++特有)

函数重载的不足：对于参数个数相同，类型不同，而实现过程完全相同的函数，仍要分别给出各个函数的实现

```
int max(int x, int y)
{
    return x>y?x:y;
}

double max(double x, double y)
{
    return x>y?x:y;
}
```

问题：两段一样的代码
能否合并为一段？

函数模板：建立一个通用函数，其返回类型及参数类型不具体指定，用一个虚拟类型来代替，该通用函数称为函数模板，调用时再根据不同的实参类型来取代模板中的虚拟类型，从而实现不同的功能

```
#include <iostream>
using namespace std;
template <typename T>
T max(T x, T y)
{
    cout << sizeof(x) << ' ' ;
    return x>y?x:y;
}

int main()
{
    int a=10, b=15;
    double f1=12.34, f2=23.45;
    cout << max(a, b) << endl;
    cout << max(f1, f2) << endl;
    return 0;
}
```

一段代码, 两个功能
1、两个int型求max
2、两个double型求max

4 15
8 23.45

§ 4. 利用函数实现指定的功能

4.9. 函数模板 (C++特有)

使用:

★ 仅适用于参数个数相同、类型不同，实现过程完全相同的情况

★ typename可用class替代

★ 类型定义允许多个

```
template <typename T1, typename t2>
template <class T1, class t2>
```

```
#include <iostream>
using namespace std;
template <typename T1, typename T2>
char max(T1 x, T2 y)
{   cout << sizeof(x) << ' ';
    cout << sizeof(y) << ' ';
    return x>y ? 'A' : 'a';
}
int main()
{   int    a = 10, b = 15;
    double f1 = 12.34, f2 = 23.45;
    cout << max(a, f1) << endl;
    cout << max(f2, b) << endl;
    return 0;
}
```

```
4 8 a
8 4 A
```

```
#include <iostream>
using namespace std;
template <typename T> ⇔ template <class T>
T max(T x, T y)
{   cout << sizeof(x) << ' ';
    return x>y?x:y;
}
int main()
{   int    a=10, b=15;
    double f1=12.34, f2=23.45;
    cout << max(a, b) << endl;
    cout << max(f1, f2) << endl;
    return 0;
}
```

§ 9. 怎样使用类和对象

9.11. 类模板

9.11.2. 类模板

引入：多个类，功能及实现完全相同，仅数据类型不同

```
class compare_int {
private:
    int x, y;
public:
    compare_int(int a, int b)
    { x=a; y=b; }
    int max()
    { return x>y?x:y; }
    int min()
    { return x<y?x:y; }
};
```

```
class compare_float {
private:
    float x, y;
public:
    compare_float(float a, float b)
    { x=a; y=b; }
    float max()
    { return x>y?x:y; }
    float min()
    { return x<y?x:y; }
};
```

合并为一个类型
为虚类型T的类

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

template <class T>
class compare {
private:
    T x, y;
public:
    compare(T a, T b)
    { x=a; y=b; }
    T max()
    { return x > y ? x : y; }
    T min()
    { return x < y ? x : y; }
};

int main()
{
    compare <int> c1(10,15);
    compare <float> c2(10.1f, 15.2f);
    cout << c1.max() << endl; 15
    cout << c2.min() << endl; 10.1
}
```

§ 9. 怎样使用类和对象

9.11. 类模板

9.11.2. 类模板

使用:

- ★ 仅适用于参数个数**相同**、类型**不同**，实现过程**完全相同**的情况
- ★ 类模板可以看作是类的抽象，称为参数化的类
- ★ 类模板成员函数体外实现时形式有所不同

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

template <class T>
class compare {
private:
    T x, y;
public:
    compare(T a, T b)
        { x=a; y=b; }
    T max()
        { return x>y?x:y; }
    T min()
        { return x<y?x:y; }
};

int main()
{
    compare <int> c1(10,15);
    compare <float> c2(10.1f, 15.2f);
    cout << c1.max() << endl;
    cout << c2.min() << endl;
}
```

体内实现

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

template <class T>
class compare {
private:
    T x, y;
public:
    compare(T a, T b);
    T max();
    T min();
};

template <class T> //每个体外实现的函数前都要
compare<T>::compare(T a, T b)
{
    x=a;
    y=b;
}

template <class T> //每个体外实现的函数前都要
T compare<T>::max()
{
    return x>y?x:y;
}

template <class T> //每个体外实现的函数前都要
T compare<T>::min()
{
    return x<y?x:y;
}

int main()
{
    compare <int> c1(10,15);
    compare <float> c2(10.1f, 15.2f);
    cout << c1.max() << endl;
    cout << c2.min() << endl;
}
```

体外实现

普通类构造函数的体外实现:
test::test(int x, int y)
{ ... }

普通类成员函数的体外实现:
int test::fun(int x, int y)
{ ... }

§ 9. 怎样使用类和对象

9. 11. 类模板

9. 11. 2. 类模板

使用:

- ★ 仅适用于参数个数**相同**、类型**不同**，实现过程**完全相同**的情况
- ★ 类模板可以看作是类的抽象，称为参数化的类
- ★ 类模板成员函数体外实现时形式有所不同
- ★ 类型定义允许多个

```
template <class T1, class t2>
```

```
template <class t1, class t2>
class test {
    private:
        t1 x;
        t2 y;
    public:
        test(t1 a, t2 b) {
            x=a;
            y=b;
        }
        ...
};

int main()
{ test <int,float> c1(10, 15.1);
  test <int,int>    c2(10, 15);
}
```