第四章 包含动态内存申请的类

主讲教师：同济大学电子与信息工程学院 陈宇飞
同济大学电子与信息工程学院 龚晓亮

目录

- 含动态内存申请的构造与析构函数
- 构造函数与析构函数的调用时机
- 对象的动态建立和释放
- 对象的赋值与复制

4.1 含动态内存申请的构造与析构函数

- 使用：
  - 有数据成员需要动态内存申请的情况下，可在构造函数中申请空间，在析构函数中释放空间
  - 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（但不提倡）

- 常规做法：系统自动调用析构函数（非显式）

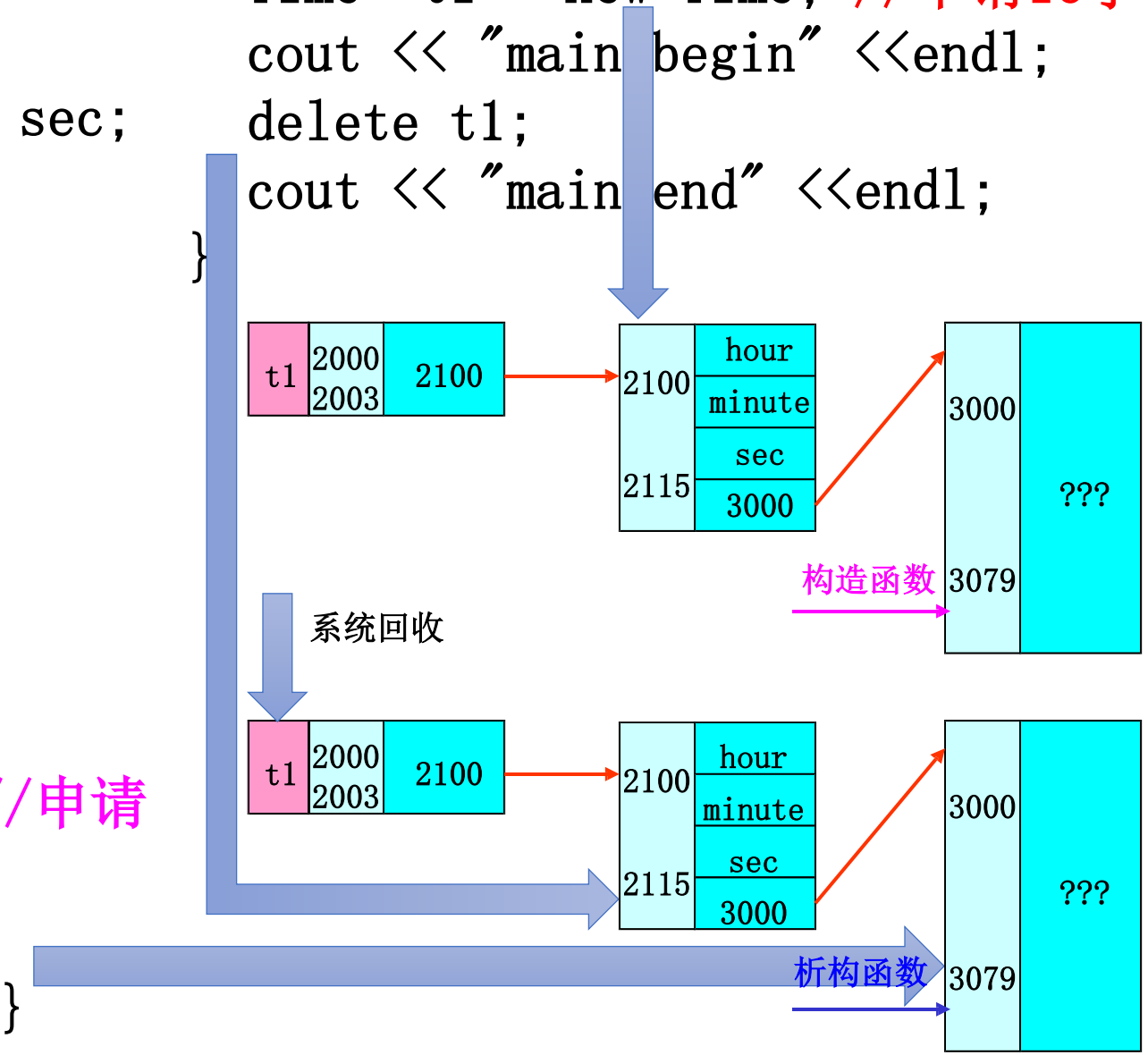
```
class Time {
  private:
    int hour;
    int minute;
    int sec;
    char *s;
  public:
    Time(); //构造函数
    ~Time();//析构函数
};
Time::Time()
{
    hour = 0;
    minute = 0;
    sec = 0;
    //接左侧
    s = new char[80];//动态申请
}
Time::~Time()
{
    delete s; //释放
}
int main()
{
    Time t1;
    ...
}
```

- 不提倡：调用Release函数（显式）

```
class Time {
  private:
    int hour;
    int minute;
    int sec;
    char *s;
  public:
    Time(); //构造函数
    Release(); //定义成员函数
};
Time::Time()
{
    hour = 0;
    minute = 0;
    sec = 0;
    //接左侧
    s = new char[80];//动态申请
}
Time::Release()
{
    delete s; //释放
}
int main()
{
    Time t1;
    ...
    t1.Release();
}
```

```
#include <iostream>
using namespace std;
class Time {
  private:
      int hour, minute, 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;
}
```




```
main begin
main end
```

目录

- 含动态内存申请的构造与析构函数
- 构造函数与析构函数的调用时机
- 对象的动态建立和释放
- 对象的赋值与复制

4.2 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　　：new时

- 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

  静态全局/外部全局对象　：程序结束时

  动态申请的对象　　：delete时

```
#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;
}
```

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

```
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;
}
```

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

```
#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;
}
```

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

```
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;
}
```

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

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

```
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;
}
```

1、main开始前分配
2、main结束后回收

```
#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;
}
```

```
main begin
Time Begin
new end
Time End
main end
```

```
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;
}
```

1、new时分配
2、delete时回收

```
#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;
}
```

```
main begin
new end
main end
```

```
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;
}
```

malloc仅仅是分配内存，不会调用构造函数

free不会调用析构函数

```
#include <iostream>
using namespace std;
class Time {
  private:
    int hour,minute,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;
}
```

```
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

```
#include <iostream>
using namespace std;
class Time {
  private:
    int hour,minute,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;
}
```

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

t1,t2都是静态局部变量

构造：t1,t2

析构：t2,t1

```
#include <iostream>
using namespace std;
class Time {
  private:
    int hour,minute,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;
}
```

```
Time Begin15
Time Begin16
main
Time End16
Time End15
```

t1,t2都是全局变量
构造：t1,t2
析构：t2,t1

```
#include <iostream>
using namespace std;
class Time {
  private:
    int hour,minute,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;
}
```

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

t1,t2是不同性质的变量

不遵循栈规则

```
#include <iostream>
using namespace std;
class Time {
  private:
    int hour,minute,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;
}
```

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

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

目录

- 含动态内存申请的构造与析构函数
- 构造函数与析构函数的调用时机
- 对象的动态建立和释放
- 对象的赋值与复制

4.3 对象的动态建立和释放

- C++方法：

```
Time *p1, *p2;
申请：p1 = new(nothrow) Time; if (p1==NULL) { ... }
      p2 = new(nothrow) Time(); if (p2==NULL) { ... }
      //new Time()创建对象时候，系统除了执行默认构造函数会
      执行的那些操作外，还会为基本数据类型和指针类型的成员
      用0赋初值
      p3 = new(nothrow) Time[2]; if (p3==NULL) { ... }
释放：delete p1;
      delete p2;
      delete []p3;
```

```
#include<iostream>
using namespace std;
class Time
{
private:
  int hour;
  int minute;
  int second;
public:
  void display();
};
void Time::display()
{
  cout << hour << endl;
  cout << minute << endl;
  cout << second << endl;
}
```

```
int main()
{  Time* p1 = new(nothrow) Time;
   if (p1 == NULL) { return -1; }
   p1->display();  //随机值
   delete p1;
   Time* p2 = new(nothrow) Time();
   if (p2 == NULL) { return -1; }
   p2->display();  //0, 0, 0
   delete p2;
   Time* p3 = new(nothrow) Time[2];
   if (p3 == NULL) { return -1; }
   p3[0].display();  //随机值
   p3[1].display();  //随机值
   delete []p3;
   return 0;
}
```

```
-842150451
-842150451
-842150451
0
0
0
-842150451
-842150451
-842150451
-842150451
-842150451
-842150451
-842150451
```

4.3 对象的动态建立和释放

- 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
  - 模块02例中，struct中有string类，则malloc/free会出错

```
#include<iostream>
using namespace std;
class Time
{
private:
  int hour;
  int minute;
  int second;
public:
  Time() { cout << "called!\n"; }
  void display();
};
void Time::display()
{
  cout << hour << endl;
  cout << minute << endl;
  cout << second << endl;
}
```

```
int main()
{  Time* p1 = new(nothrow) Time;
                    //调用构造函数
   if (p1 == NULL) { return -1; }
   p1->display();
   delete p1;

   Time* p2 = (Time*)malloc(sizeof(Time));
                    //未调用构造函数
   if (p2 == NULL) { return -1; }
   p2->display();
   free(p2);

   retuen 0;
}
```

```
called!
-842150451
-842150451
-842150451
-842150451
-842150451
-842150451
-842150451
```

目录

- 含动态内存申请的构造与析构函数
- 构造函数与析构函数的调用时机
- 对象的动态建立和释放
- 对象的赋值与复制

4.4 对象的赋值与复制

- 基本概念

	赋值	复制
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	建立一个新对象,其值与某个已有对象完全相同
形式	//执行语句 类名 对象名1，对象名2; 对象名1=对象名2;	//定义语句 类名 对象名(已有对象名) 类名 对象名=已有对象名
实现	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 (整体内存拷贝)	建立新对象时自动调用拷贝构造函数 浅拷贝：缺省拷贝构造函数（内存拷贝） 深拷贝：拷贝构造函数重载（动态分配）

思考：1）若对象数据成员无动态分配的数据，结果是否与预期一致？若不一致如何解决？

2）若对象数据成员是指针及动态分配的数据呢？

4.4 对象的赋值与复制

- 对象的赋值
  - 有动态内存申请

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

```
#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;
 }
};
```

4.4 对象的赋值与复制

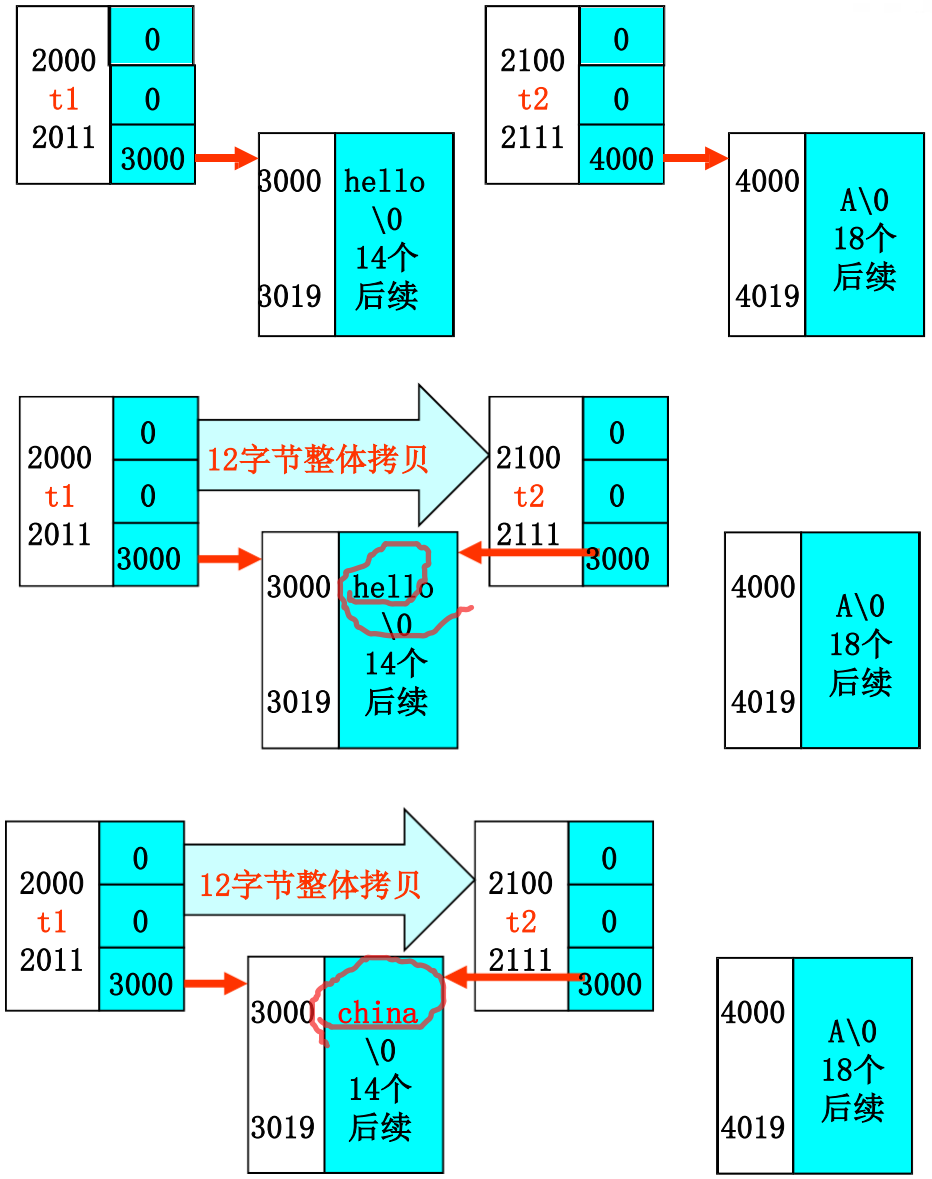
- 对象的赋值
  - 有动态内存申请　　//上例运行结果：

//有动态内存申请时，执行结果错且有错误弹窗

4.4 对象的赋值与复制

- 对象的赋值
  - 有动态内存申请

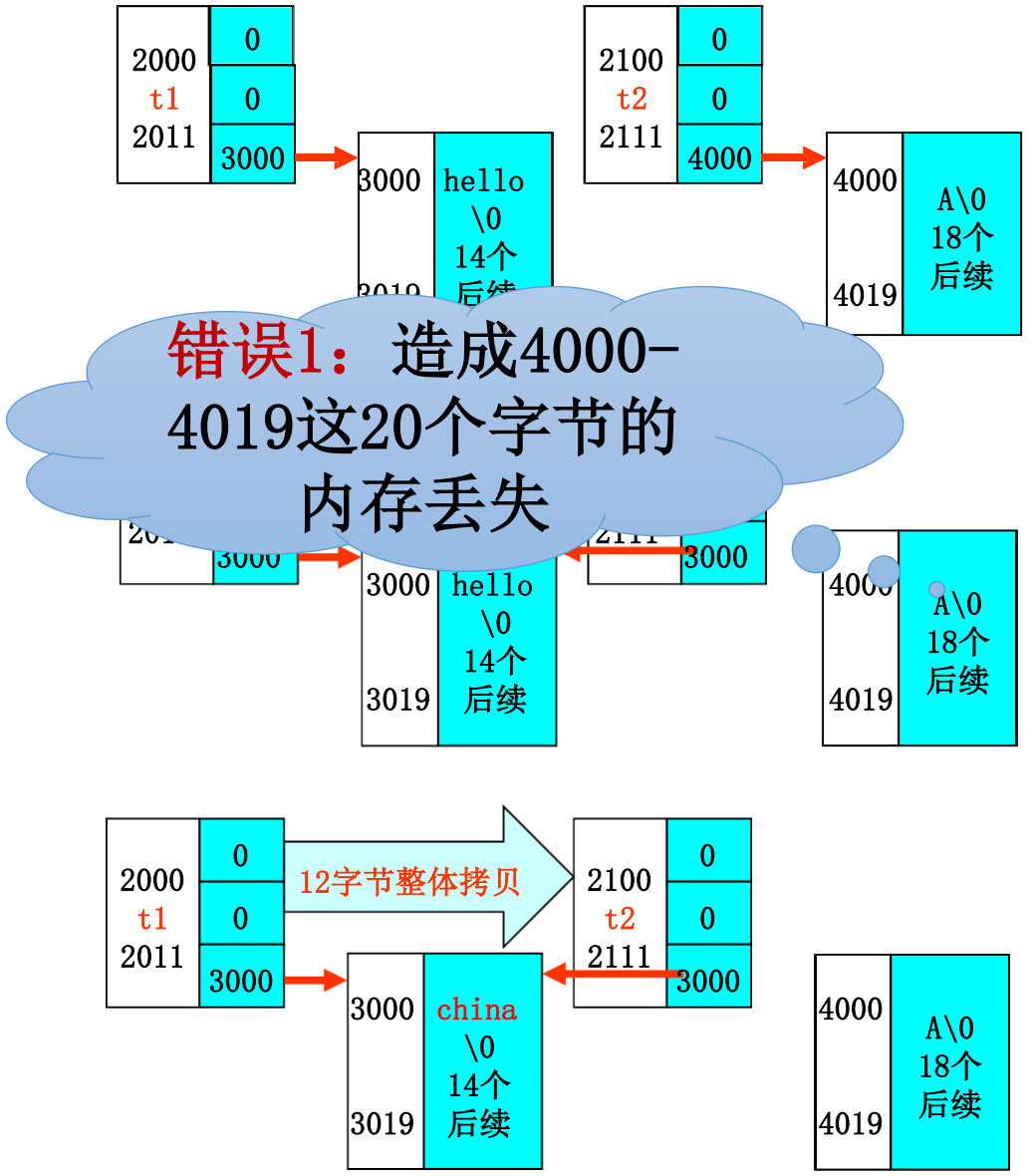
```
int main()
{   test t1("hello"), t2;
    t1.display();       hello
    t2.display();       A
    t2=t1;
    t2.display();       hello
    t1.set("china");
    t1.display();       china
    t2.display();       china
}
```



4.4 对象的赋值与复制

- 对象的赋值
  - 有动态内存申请

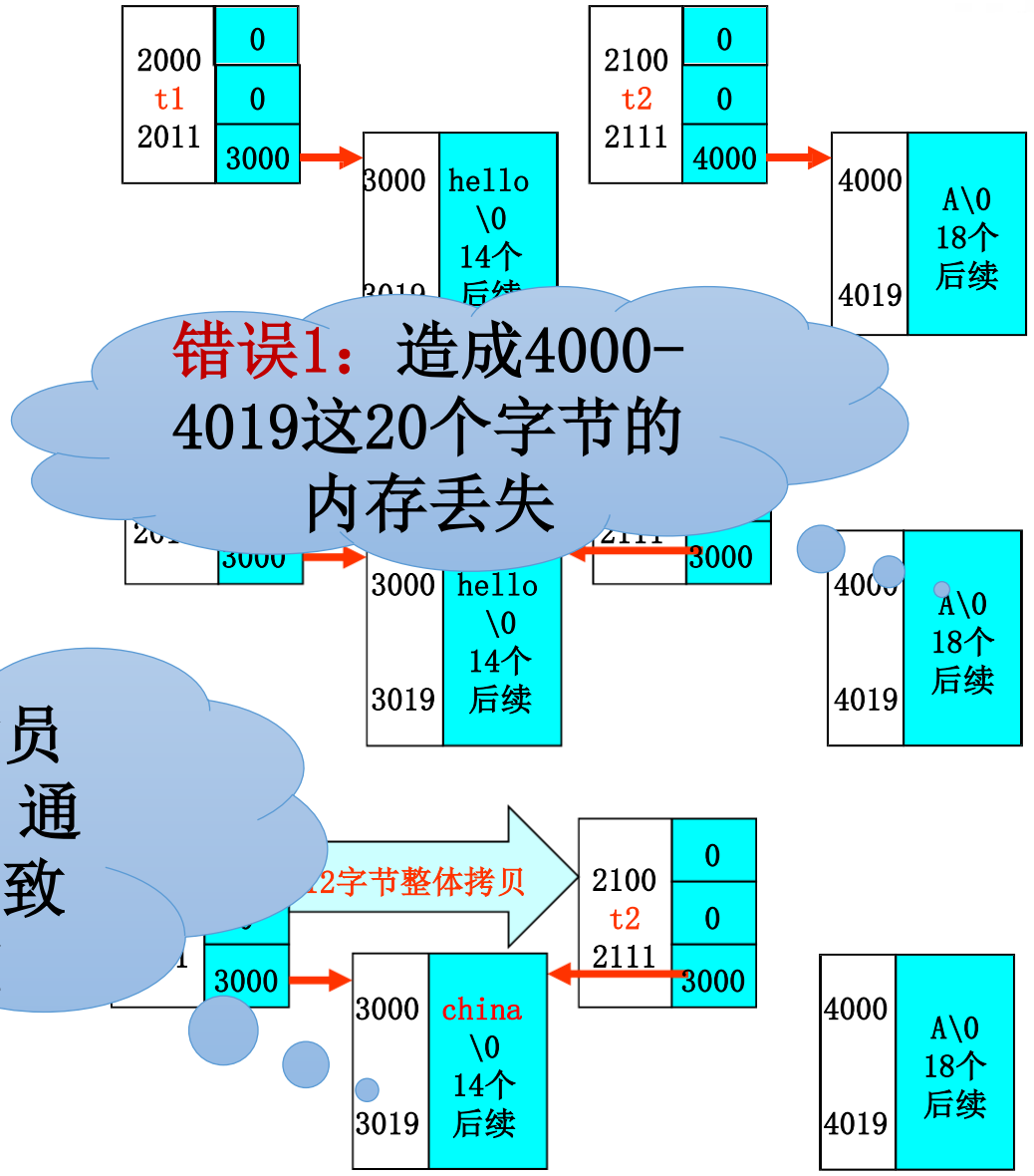
```
int main()
{   test t1("hello"), t2;
    t1.display();       hello
    t2.display();       A
    t2=t1;
    t2.display();       hello
    t1.set("china");
    t1.display();       china
    t2.display();       china
}
```



4.4 对象的赋值与复制

- 对象的赋值
  - 有动态内存申请

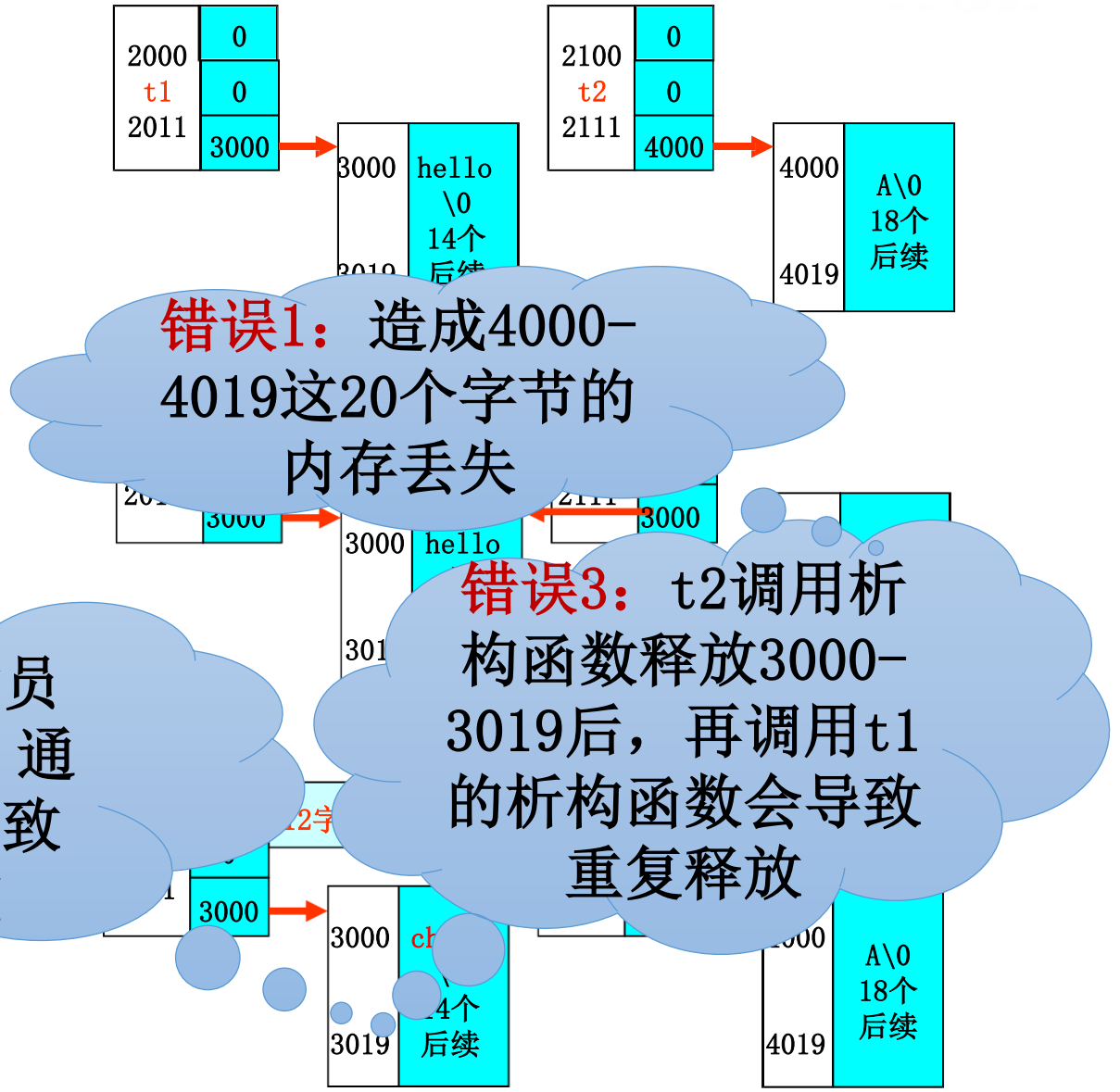
```
int main()
{   test t1("hello"), t2;
    t1.display();       hello
    t2.display();       A
    t2=t1;
    t2.display()
    t1.set("
    t1.di
    t2.dis
}
```



4.4 对象的赋值与复制

- 对象的赋值
  - 有动态内存申请

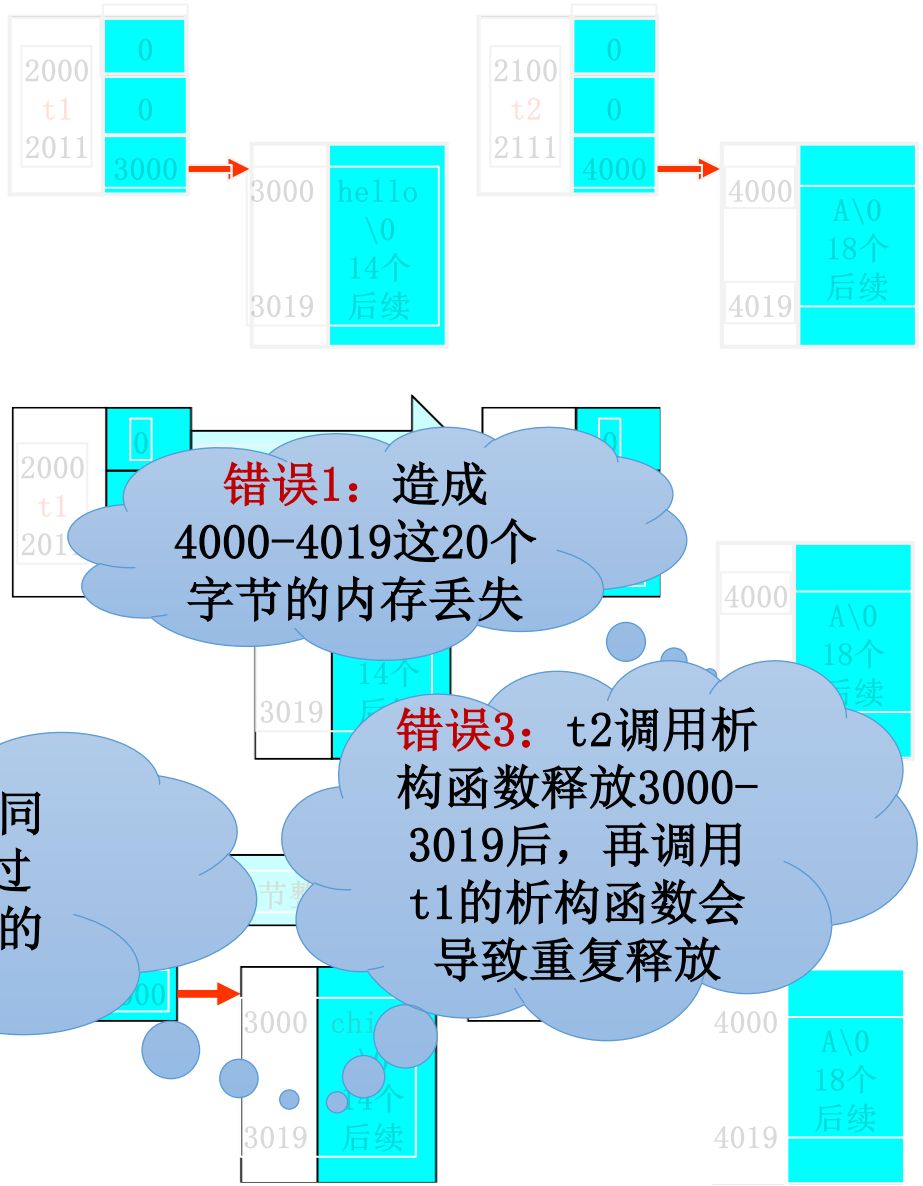
```
int main()
{   test t1("hello"), t2;
    t1.display();       hello
    t2.display();       A
    t2=t1;
    t2.display()
    t1.set("
    t1.di
    t2.dis
}
```



4.4 对象的赋值与复制

- 对象的赋值
  - 有动态内存申请

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



• 解决方法：运算符重载！！（后续内容，此处了解即可）

```
#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; }
    test &operator=(const test &t);
         //重载=的声明
};
test &test::operator=(const test &t)
         //重载=体外实现
{  a = t.a;   b = t.b;
   delete c;          //释放原空间
   c=new char[20];    //申请新空间
   strcpy(c, t.c);
   return *this;      //返回对象自身
}
int main()
{  …
}
```

4.4 对象的赋值与复制

- 对象的赋值
  - 有动态内存申请 //上例运行结果(运算符重载为后续内容，此处仅了解)

test1.cpp

Project1 | test | operator=(const test & t)

```
1  #define _CRT_SECURE_NO_WARNINGS
2  #include <iostream>
3  #include <cstring>
4  using namespace std;
5  class test {
6  private:
7      int a; int b; char* c;
8  public:
9      test(const char* s = "A")
10     {
11         a = 0; b = 0;
12         c = new char[20];
13         strcpy(c, s);
14     }
15     ~test() { delete c; };
16     void set(const char* s) { strcpy(c, s); }
17     void display() { cout << c << endl; }
18     test& operator=(const test& t); //重载=的声明
19 };
20 test& test::operator=(const test& t)//重载=体外实现
```

Microsoft Visual Studio 调试控制台

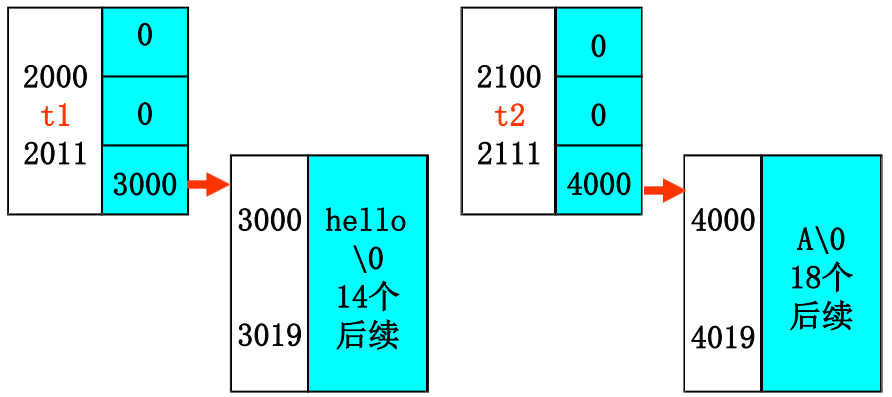
```
hello
A
hello
china
hello

C:\Users\april\source\repos\Project1\Debug\Project1.exe (进程 14356)已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .
```

4.4 对象的赋值与复制

- 对象的赋值
  - 有动态内存申请

```
int main()
{   test t1("hello"), t2;
    t1.display();    hello
    t2.display();    A
    t2=t1;
    t2.display();
    t1.set("china");
    t1.display();
    st2.display();
}
```

```
//解决方法：运算符重载！！
test &test::operator=(const test &t)
{  a = t.a;    b = t.b;
   delete c;                  //释放原空间
   c=new char[20];            //申请新空间
   strcpy(c, t.c);
   return *this;              //返回对象自身
}
```

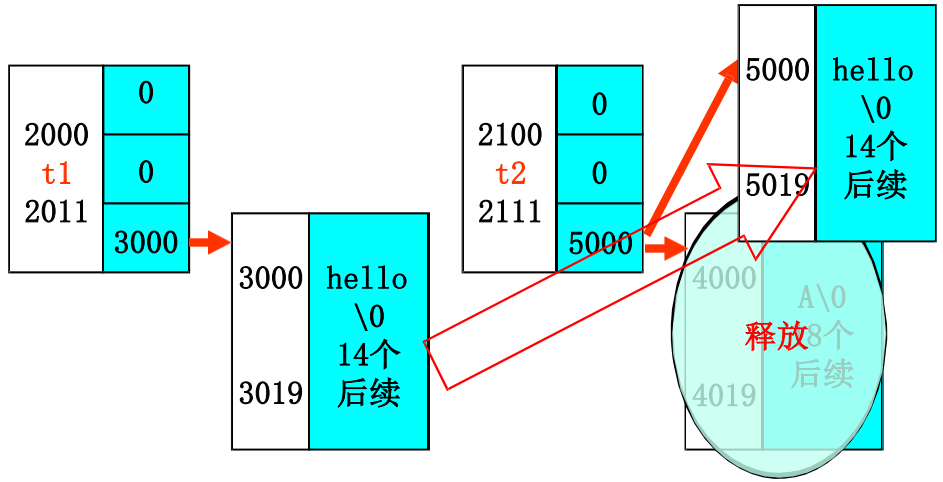


4.4 对象的赋值与复制

- 对象的赋值
  - 有动态内存申请

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

```
//解决方法：运算符重载！！
test &test::operator=(const test &t)
{  a = t.a;    b = t.b;
   delete c;                  //释放原空间
   c=new char[20];            //申请新空间
   strcpy(c, t.c);
   return *this;          //返回对象自身
}
```

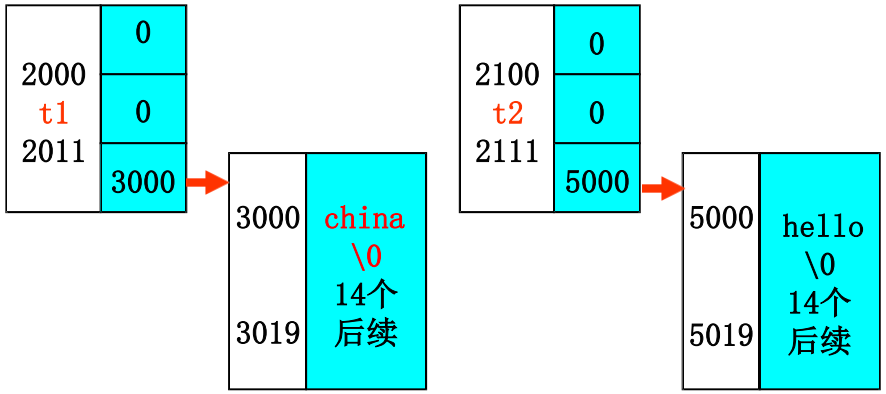


4.4 对象的赋值与复制

- 对象的赋值
  - 有动态内存申请

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

```
//解决方法：运算符重载！！
test &test::operator=(const test &t)
{  a = t.a;    b = t.b;
   delete c;                  //释放原空间
   c=new char[20];            //申请新空间
   strcpy(c, t.c);
   return *this;              //返回对象自身
}
```



4.4 对象的赋值与复制

- 对象的复制
  - 拷贝构造函数/复制构造函数
  - 形式：类名(const 类名 &引用名)
    - 用一个对象的值去初始化另一个对象
    - 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
    - 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（浅拷贝）
    - 若定义了复制构造函数，则系统缺省定义的消失（可做深拷贝）

4.4 对象的赋值与复制

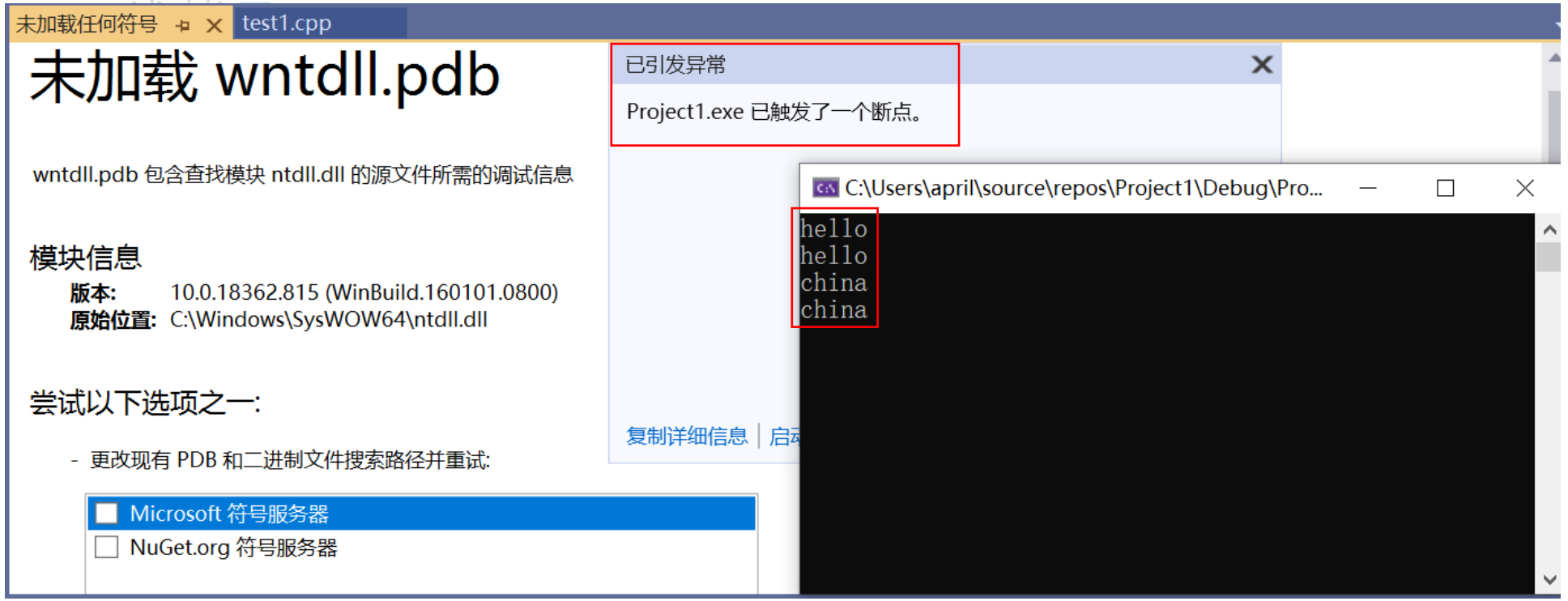
- 对象的复制
  - 浅拷贝

```
#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);
   t1.display();
   t2.display();
   t1.set("china");
   t1.display();
   t2.display();
}
```

4.4 对象的赋值与复制

- 对象的复制
  - 浅拷贝　　//上例运行结果：

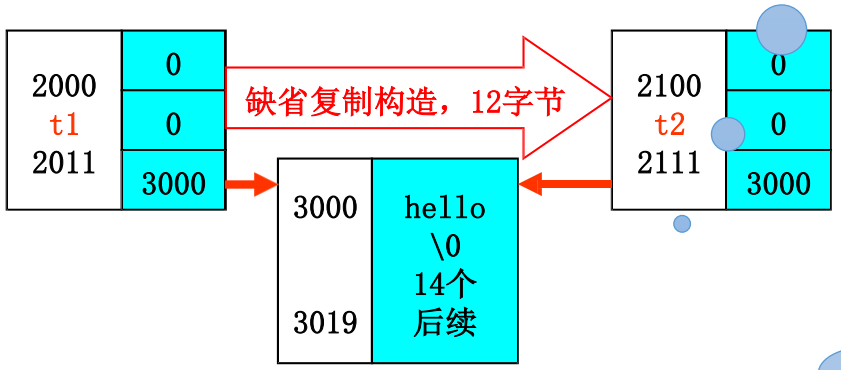


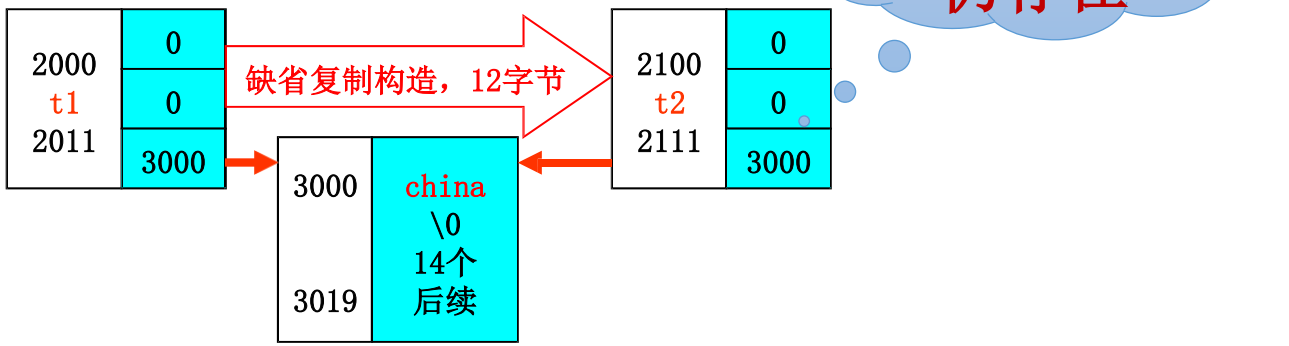
//有动态内存申请时，执行结果错且有错误弹窗

4.4 对象的赋值与复制

- 对象的复制
  - 浅拷贝

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





4.4 对象的赋值与复制

- 对象的复制
  - 浅拷贝

如何解决？

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

• 深拷贝 --动态成员不是简单赋值，而是重新动态分配空间

```
#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(const test &t);
       //复制构造函数的声明
    ~test(){  delete c; }
    void set(const 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);
}
int main()
{…}
```

4.4 对象的赋值与复制

- 对象的复制
  - 深拷贝 //上例运行结果：

test1.cpp

Project1 (全局范围)

```
1  #define _CRT_SECURE_NO_WARNINGS
2  #include <iostream>
3  #include <cstring>
4  using namespace std;
5  class test {
6  private:
7      int a;
8      int b;
9      char* c;
10 public:
11     test(const char* s = "A")
12     {
13         a = 0; b = 0;
14         c = new char[20];
15         strcpy(c, s);
16     }
17     test(const test& t); //复制构造函数的声明
18     ~test() { delete c; }
19     void set(const char* s) { strcpy(c, s); }
20     void display() { cout << c << endl; }
```

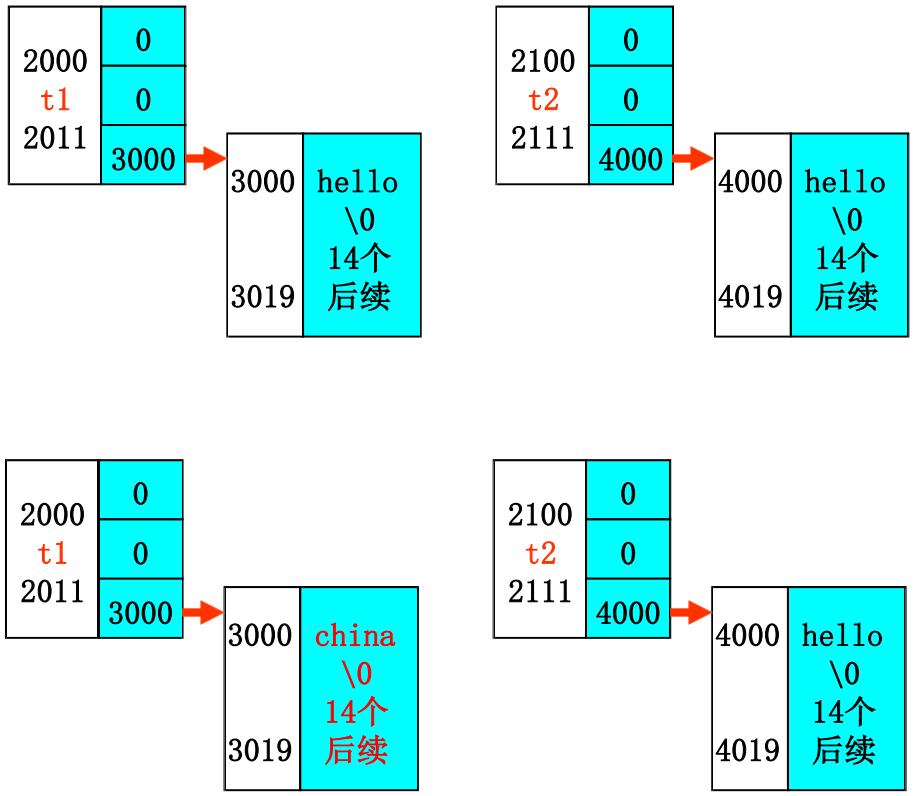
Microsoft Visual Studio 调试控制台

```
hello
hello
china
hello

C:\Users\april\source\repos\Project1\Debug\Project1.exe (进程 20248)已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .
```

4.4 对象的赋值与复制

- 对象的复制
  - 深拷贝

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



4.4 对象的赋值与复制

- 对象的赋值与复制（小结）
  - 对象的赋值发生在执行语句时，对象的复制发生在定义语句时
  - 赋值的操作是整体内存拷贝，复制的操作是自动调用拷贝构造函数
  - 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甚至报错
    - 解决方法：

      赋值：运算符重载（后续章节）

      复制：拷贝构造函数重载（深拷贝）

总结

- 含动态内存申请的构造与析构函数
- 构造函数与析构函数的调用时机
- 对象的动态建立和释放
- 对象的赋值与复制