



§ 13. 动态内存申请

1. 结构体类型的定义及使用 (第07模块 复习)

- ★ 结构体类型的声明
- ★ 字节对齐
- ★ 结构体变量的定义和初始化（普通变量、数组、指针、引用）
- ★ 指向结构体变量的指针和指向结构体变量中某个成员的指针
- ★ 结构体（struct）和类（class）的区别



§ 13. 动态内存申请

2. 指向结构体变量的指针与链表

2.1. 链式结构的基本概念

★ 数组的不足

- 1、大小必须在定义时确定，导致空间浪费
是否可以按需分配空间
- 2、占用连续空间，导致小空间无法充分利用
是否可以充分利用不连续的空间
- 3、在插入/删除元素时必须前后移动元素
插入/删除时能否不移动元素

★ 链表

不连续存放数据, 用指针指向下一数据的存放地址

例：数据1，2，3，4，5，分别存放在数组和链表中

存放5个元素：

数组：连续的20字节

链表：非连续的40字节

(每个结点的8字节连续)

问：本例中，存储相同数量数据，链表所占空间是数组的两倍，为什么不把这个问题当做是链表的缺点？

在数组/链表含有大量数据时：
1、频繁在任意位置插入/删除，哪种方式好？
2、频繁存取第i个元素的值，哪种方式好？(i随机)

数组

2000	1
2003	
2004	2
2007	
2008	3
2011	
2012	4
2015	
2016	5
2019	

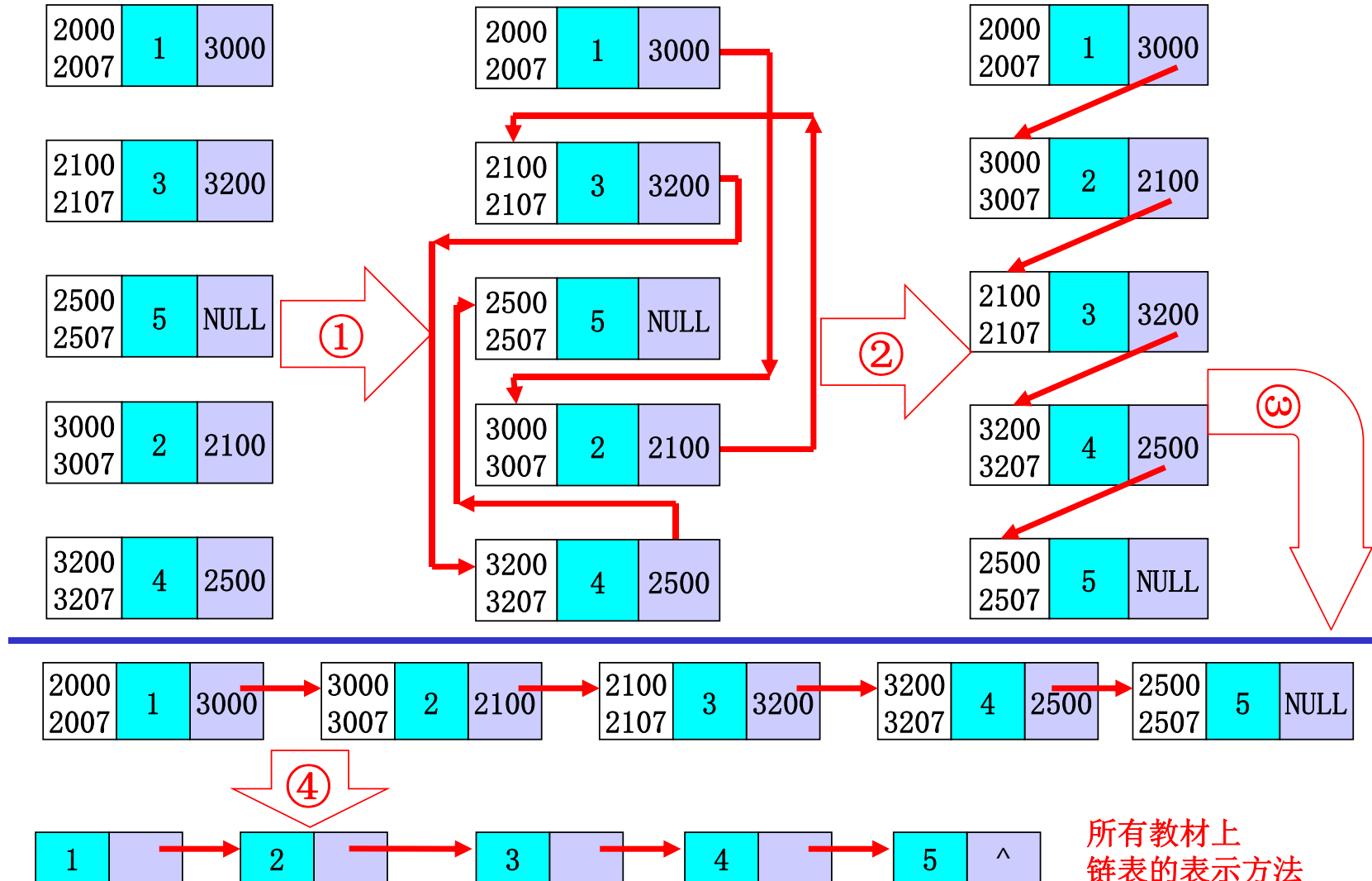
链表

2000	1	3000
2007		
2100	3	3200
2107		
2500	5	NULL
2507		
3000	2	2100
3007		
3200	4	2500
3207		



§ 13. 动态内存申请

例：数据1，2，3，4，5，分别存放在数组和链表中





§ 13. 动态内存申请

2. 指向结构体变量的指针与链表

2.1. 链式结构的基本概念

结点：存放数据的基本单位

{ 数据域：存放数据的值
指针域：存放下一个同类型节点的地址

链表：由若干结点构成的链式结构

表头结点：第一个结点

表尾结点：链表的最后一个结点，指针域为NULL(空)

头指针：指向链表的表头节点的指针



例：一个简单的静态方式链表(非链表的常规用法)

```
struct student {  
    long num;  
    float score;  
    struct student *next;  
};
```

指向结构体自身的指针
成员类型不允许是自身的结构体类型，
但可以是指针(因为指针占用空间已知)

```
int main()  
{  
    student a,b,c, *head, *p;  
    a.num = 31001; a.score=89.5;  
    b.num = 31003; b.score=90;  
    c.num = 31007; c.score=85;  
    head = &a;    a.next = &b;    b.next = &c;    c.next = NULL;  
    p=head;  
    do {  
        cout << p->num << " " << p->score << endl;  
        p=p->next;  
    } while(p!=NULL);  
}
```



例：一个简单的静态方式链表(非链表的常规用法)

```
struct student {
    long num;
    float score;
    struct student *next;
};

int main()
{
    student a,b,c, *head, *p;
    a.num = 31001; a.score=89.5;
    b.num = 31003; b.score=90;
    c.num = 31007; c.score=85;
    head = &a;    a.next = &b;    b.next = &c;    c.next = NULL;
    p=head;
    do {
        cout << p->num << " " << p->score << endl;
        p=p->next;
    } while(p!=NULL);
}
```

a	2000 2011	?	?
---	--------------	---	---

(结点)

b	3000 3011	?	?
---	--------------	---	---

(结点)

c	2500 2511	?	?
---	--------------	---	---

(结点)

head	2100 2103	?
------	--------------	---

p	2200 2203	?
---	--------------	---



例：一个简单的静态方式链表(非链表的常规用法)

```
struct student {
    long num;
    float score;
    struct student *next;
};

int main()
{
    student a, b, c, *head, *p;
    a.num = 31001; a.score=89.5;
    b.num = 31003; b.score=90;
    c.num = 31007; c.score=85;
    head = &a;   a.next = &b;   b.next = &c;   c.next = NULL;
    p=head;
    do {
        cout << p->num << " " << p->score << endl;
        p=p->next;
    } while(p!=NULL);
}
```

a	2000 2011	31001 89.5	?
---	--------------	---------------	---

(结点)

b	3000 3011	31003 90	?
---	--------------	-------------	---

(结点)

c	2500 2511	31007 85	?
---	--------------	-------------	---

(结点)

head	2100 2103	?
------	--------------	---

p	2200 2203	?
---	--------------	---



例：一个简单的静态方式链表(非链表的常规用法)

```
struct student {
    long num;
    float score;
    struct student *next;
};

int main()
{
    student a,b,c, *head, *p;
    a.num = 31001; a.score=89.5;
    b.num = 31003; b.score=90;
    c.num = 31007; c.score=85;
    head = &a;  a.next = &b;  b.next = &c;  c.next = NULL;
    p=head;
    do {
        cout << p->num << " " << p->score << endl;
        p=p->next;
    } while(p!=NULL);
}
```

a	2000	31001	3000
	2011	89.5	

(结点)

b	3000	31003	2500
	3011	90	

(结点)

c	2500	31007	NULL
	2511	85	

(结点)

head	2100	2000
	2103	

p	2200	?
	2203	



例：一个简单的静态方式链表(非链表的常规用法)

```
struct student {  
    long num;  
    float score;  
    struct student *next;  
};  
  
int main()  
{  
    student a,b,c, *head, *p;  
    a.num = 31001; a.score=89.5;  
    b.num = 31003; b.score=90;  
    c.num = 31007; c.score=85;  
    head = &a;    a.next = &b;    b.next = &c;    c.next = NULL;  
    p=head;  
    do {  
        cout << p->num << " " << p->score << endl;  
        p=p->next;  
    } while(p!=NULL);  
}
```

a	2000	31001	3000
	2011	89.5	

(结点)

b	3000	31003	2500
	3011	90	

(结点)

c	2500	31007	NULL
	2511	85	

(结点)

head	2100	2000
	2103	

p	2200	?
	2203	



例：一个简单的静态方式链表(非链表的常规用法)

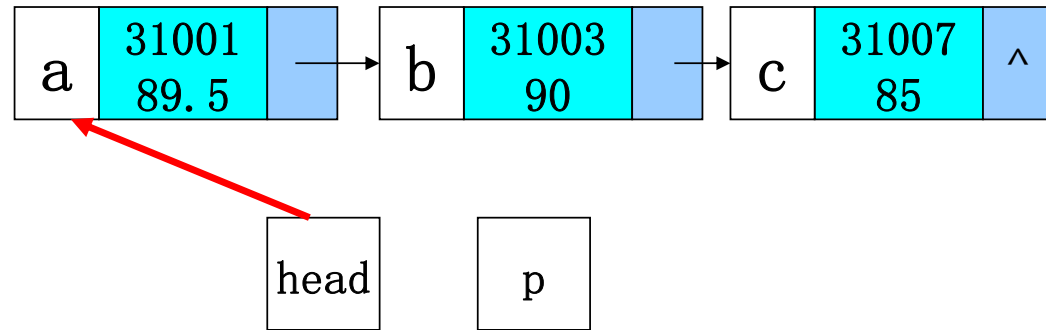
```
struct student {  
    long num;  
    float score;  
    struct student *next;  
};
```

```
int main()  
{  
    student a,b,c, *head, *p;  
    a.num = 31001; a.score=89.5;  
    b.num = 31003; b.score=90;  
    c.num = 31007; c.score=85;
```

```
    head = &a;    a.next = &b;    b.next = &c;    c.next = NULL;
```

```
    p=head;
```

```
    do {  
        cout << p->num << " " << p->score << endl;  
        p=p->next;  
    } while(p!=NULL);  
}
```

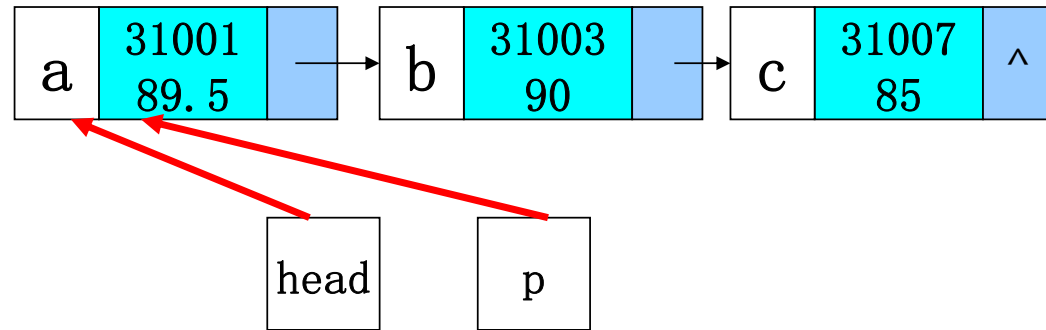




例：一个简单的静态方式链表(非链表的常规用法)

```
struct student {  
    long num;  
    float score;  
    struct student *next;  
};
```

```
int main()  
{  
    student a,b,c, *head, *p;  
    a.num = 31001; a.score=89.5;  
    b.num = 31003; b.score=90;  
    c.num = 31007; c.score=85;  
    head = &a;    a.next = &b;    b.next = &c;    c.next = NULL;  
    p=head;  
    do {  
        cout << p->num << " " << p->score << endl;  
        p=p->next;  
    } while(p!=NULL);  
}
```

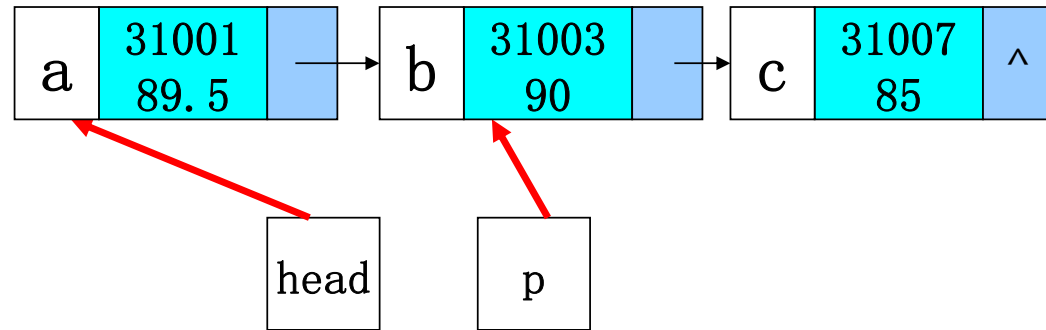




例：一个简单的静态方式链表(非链表的常规用法)

```
struct student {  
    long num;  
    float score;  
    struct student *next;  
};
```

```
int main()  
{  
    student a,b,c, *head, *p;  
    a.num = 31001; a.score=89.5;  
    b.num = 31003; b.score=90;  
    c.num = 31007; c.score=85;  
    head = &a;    a.next = &b;    b.next = &c;    c.next = NULL;  
    p=head;  
    do {  
        cout << p->num << " " << p->score << endl;  
        p=p->next;  
    } while(p!=NULL);  
}
```



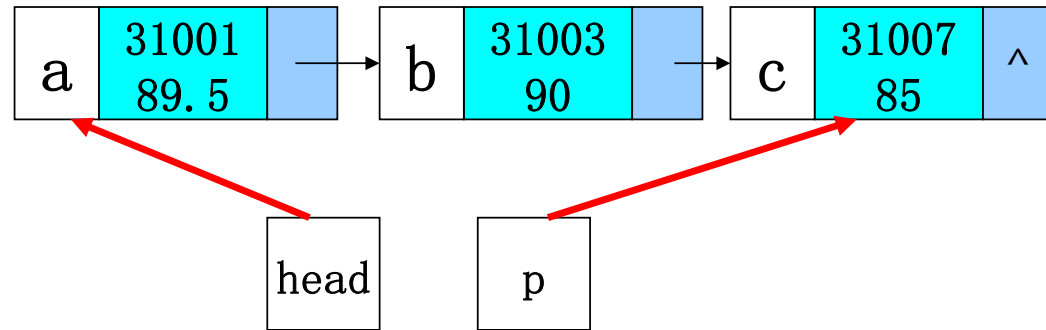
31001 89.5



例：一个简单的静态方式链表(非链表的常规用法)

```
struct student {  
    long num;  
    float score;  
    struct student *next;  
};
```

```
int main()  
{  
    student a,b,c, *head, *p;  
    a.num = 31001; a.score=89.5;  
    b.num = 31003; b.score=90;  
    c.num = 31007; c.score=85;  
    head = &a;    a.next = &b;    b.next = &c;    c.next = NULL;  
    p=head;  
    do {  
        cout << p->num << " " << p->score << endl;  
        p=p->next;  
    } while(p!=NULL);  
}
```



```
31001 89.5  
31003 90
```



例：一个简单的静态方式链表(非链表的常规用法)

```
struct student {  
    long num;  
    float score;  
    struct student *next;  
};
```

```
int main()  
{
```

```
    student a,b,c, *head, *p;  
    a.num = 31001; a.score=89.5;  
    b.num = 31003; b.score=90;  
    c.num = 31007; c.score=85;
```

```
    head = &a;    a.next = &b;    b.next = &c;    c.next = NULL;
```

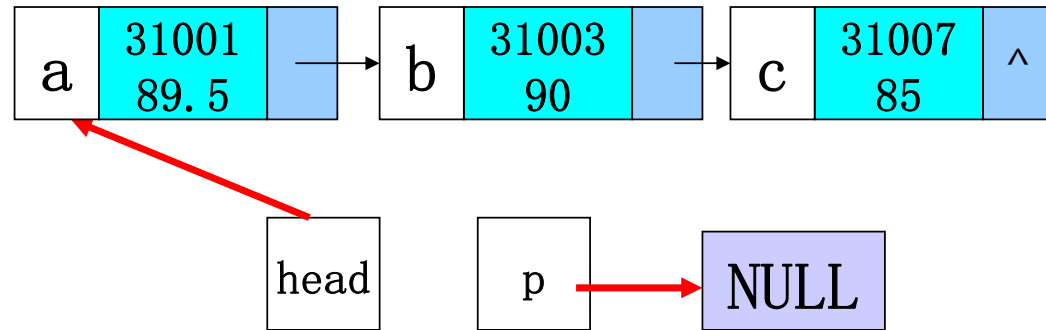
```
    p=head;
```

```
    do {
```

```
        cout << p->num << " " << p->score << endl;  
        p=p->next;
```

```
    } while(p!=NULL);
```

```
}
```



```
31001 89.5  
31003 90  
31007 85
```



§ 13. 动态内存申请

2. 指向结构体变量的指针与链表

2. 1. 链式结构的基本概念

2. 2. 链表与数组的比较

数组	链表
大小在声明时固定	大小不固定
处理的数据个数有差异时，须按最大值声明	根据需要随时增加/减少结点
内存地址连续，可直接计算得到某个元素的地址	内存地址不连续，必须依次查找
逻辑上连续，物理上连续	逻辑上连续，物理上不连续



§ 13. 动态内存申请

3. 内存的动态申请与释放

3.1. C中的相关函数

- ★ `void *malloc(unsigned size);`
 - 申请size字节的连续内存空间, 返回该空间首地址, 对申请到的空间不做初始化操作
 - 如果申请不到空间, 返回NULL
- ★ `void *calloc(unsigned n, unsigned size);`
 - 申请n*size字节的连续内存空间, 返回该空间首地址, 对申请到的空间做初始化为0 (\0)
 - 如果申请不到空间, 返回NULL
- ★ `void *realloc(void *ptr, unsigned newsize);`
 - 稍后见专题讨论
- ★ `void free(void *p);`

释放p所指的内存空间 (p必须是malloc/calloc/realloc返回的首地址)

- 因为是系统库函数, 需要包含头文件 (VS系列可不要)

`#include <stdlib.h> //C方式`

`#include <cstdlib> //C++方式`

3.2. C++中的相关运算符

- ★ 用 `new` 运算符申请空间 (如果申请不到空间, `new`缺省会抛出`bad_alloc`异常, 需要使用try-catch方式处理异常; 也可以在new时加`nothrow`来强制禁用抛出异常并返回NULL)
 - try-throw-catch称为C++的异常处理机制, 后面再专题介绍
- ★ 用 `delete` 运算符释放空间
- 因为是运算符, 不需要包含头文件

★ 用malloc/calloc等申请的空间, 用free释放, 用new申请的空间, 用delete释放



§ 13. 动态内存申请

3. 内存的动态申请与释放

申请对象	C的函数方式	C++的运算符
普通变量	形式1: 先定义指针变量, 再申请 <pre>int *p; p = (int *)malloc(sizeof(int)); p = (int *)calloc(1, sizeof(int));</pre>	形式1: 先定义指针变量, 再申请 <pre>int *p; p = new int;</pre>
	形式2: 定义指针变量的同时申请 <pre>int *p = (int *)malloc(sizeof(int)); int *p = (int *)calloc(1, sizeof(int));</pre>	形式2: 定义指针变量的同时申请 <pre>int *p = new int;</pre>
	说明: 虽然初次申请时也可以用 <pre>p = (int *)realloc(NULL, sizeof(int));</pre> 但一般不用	形式3: 申请空间时赋初值 <pre>int *p; 或 int *p=new int(10); p=new int(10);</pre>



§ 13. 动态内存申请

3. 内存的动态申请与释放

申请对象	C的函数方式	C++的运算符
一维数组	形式1: 先定义指针变量, 再申请 <pre>int *p; p = (int *)malloc(10*sizeof(int)); p = (int *)calloc(10, sizeof(int));</pre>	形式1: 先定义指针变量, 再申请 <pre>int *p; p = new int[10]; //申请10个int型空间</pre>
	形式2: 定义指针变量的同时申请空间 <pre>char *name = (char *)malloc(10*sizeof(char)); char *name = (char *)calloc(10, sizeof(char));</pre>	形式2: 定义指针变量的同时申请空间 <pre>char *name = new char[10]; //申请10个char</pre>
	<p>说明:</p> <p>虽然初次申请时也可以用</p> <pre>p = (int *)realloc(NULL, 10*sizeof(int));</pre> <p>但一般不用</p>	<p>形式3: 申请空间时赋初值</p> <ul style="list-style-type: none">● 动态申请的一维数组可以在申请时赋初值, 方法为后面跟 {}, {} 前不要加=, 且[]内必须有数, 其余规则同一维数组定义时初始化● 对于字符类型, Dev/Linux不支持字符串方式初始化 <p>例:</p> <pre>int *p; p = new int[5] {1, 2, 3, 4, 5}; //正确 p = new int[5] {1, 2, 3, 4, 5, 6}; //错误 p = new int[5] {1, 2}; //后面自动为0 p = new int[5]={1, 2, 3, 4, 5}; //错误 p = new int[] {1, 2, 3, 4, 5}; //VS正确+Dev/Linux错误</pre> <pre>char *s; s = new char[5] {'H', 'e', 'l', 'l', 'o'}; //正确 s = new char[5] {"Hello"}; //错误 s = new char[6] {"Hello"}; //VS正确+Dev/Linux错误</pre>



§ 13. 动态内存申请

3. 内存的动态申请与释放

申请对象	C的函数方式	C++的运算符
二维数组	形式1: 先定义指针变量, 再申请 <pre>int (*p)[4]; p = (int (*)[4])malloc(3*4*sizeof(int)); p = (int (*)[4])calloc(3*4, sizeof(int));</pre>	形式1: 先定义指针变量, 再申请 <pre>int *p; p = new int[3][4]; //申请3行4列, 错!!! int (*p)[4]; p = new int[3][4]; //申请3行4列, 对!!!</pre>
	形式2: 定义指针变量的同时申请空间 <pre>int (*p)[4] = (int (*)[4])malloc(3*4*sizeof(int)); int (*p)[4] = (int (*)[4])calloc(3*4, sizeof(int));</pre>	形式2: 定义指针变量的同时申请空间 <pre>float (*f)[4]=new float[3][4];</pre>
	说明: 虽然初次申请时也可以用 <pre>p = (int (*)[4])realloc(NULL, 3*4*sizeof(int));</pre> 但一般不用	形式3: 申请空间时赋初值 ● 动态申请的二维数组可以在申请时赋初值, 方法为后面跟 双层{} , {}前不要加=, 且[]内必须有数 , 其余规则同二维数组定义时初始化 ● 对于字符类型, Dev/Linux不支持字符串方式初始化 例: <pre>int (*p)[3]; p = new int[2][3] {1, 2, 3, 4, 5, 6}; //VS正确+Dev/Linux错误 p = new int[2][3] {{1, 2, 3}, {4, 5, 6}}; //正确 p = new int[2][3] {{1, 2}, {3, 4, 5, 6}}; //错误 p = new int[2][3] {1, 4}; //VS正确+Dev/Linux错误 p = new int[2][3] {{1}, {4}}; //正确</pre> 例: <pre>char (*p)[6]; p = new char[2][6] {'A', 'B', 'C'}; //VS正确+Dev/Linux错误 p = new char[2][6] {{'A'}, {'B', 'C'}}; //正确 p = new char[2][6] {"Hello", "China"}; //VS正确+Dev/Linux错误 p = new char[2][6] {"Hello1", "China"}; //错误</pre> 注: 字符型在使用字符串方式初始化时, VS允许一层{}



§ 13. 动态内存申请

3. 内存的动态申请与释放

释放对象	C的函数方式	C++的运算符
普通变量	<pre>int *p = (int *)malloc(sizeof(int)); int *p = (int *)calloc(1, sizeof(int)); free(p);</pre>	<pre>int *p = new int; delete p;</pre>
一维数组	<pre>int *p = (int *)malloc(10*sizeof(int)); int *p = (int *)calloc(10, sizeof(int)); free(p);</pre>	<pre>int *p = new int[10]; delete []p;</pre> <ul style="list-style-type: none">● 某些资料上说可以 <code>delete name</code>, 因为一维数组地址可理解为首元素地址, 不加[] (说法不准确, 必须加)● 对于int/char等基本类型的数组, 加不加均正确; 但对于用户自定义的class, 则必须加, 错误例子见后
二维数组	<pre>int (*p)[4] = (int (*)[4])malloc(3*4*sizeof(int)); int (*p)[4] = (int (*)[4])calloc(3*4, sizeof(int)); free(p);</pre>	<pre>int (*p)[4] = new int[3][4]; delete []p;</pre> <ul style="list-style-type: none">● 二维以上必须加一个[], 否则编译警告



§ 13. 动态内存申请

3. 内存的动态申请与释放

释放对象	C++的运算符
一维数组	<pre>int *p = new int[10]; delete []p;</pre> <ul style="list-style-type: none">● 某些资料上说可以 <code>delete name</code>, 因为一维数组地址可理解为 首元素地址, 不加[] (说法不准确, 必须加)● 对于int/char等基本类型的数组, 加不加均正确; 但对于用户自定义的class, 则必须加, 错误例子见后

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

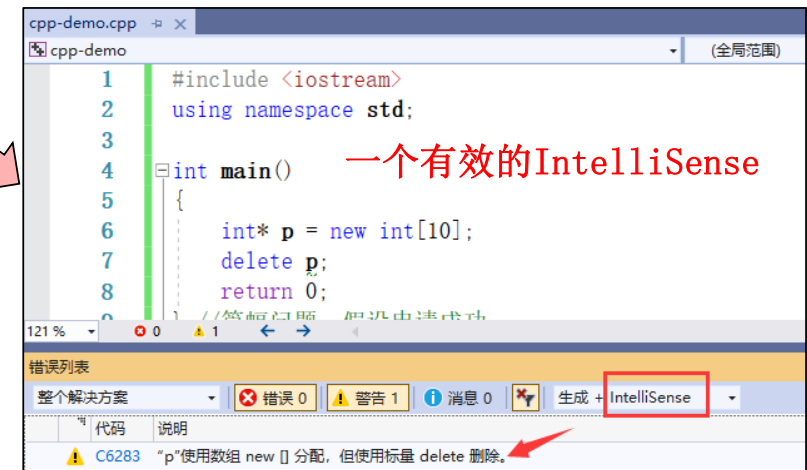
int main()
{
    int *p = new int[10];
    delete []p;

    return 0;
}
//篇幅问题, 假设申请成功
```

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

int main()
{
    int *p = new int[10];
    delete p;

    return 0;
}
//篇幅问题, 假设申请成功
```



§ 13. 动态内存申请

3. 内存的动态申请与释放

释放对象	C++的运算符
一维数组	<pre>int *p = new int[10]; delete []p;</pre> <ul style="list-style-type: none">某些资料上说可以 <code>delete name</code>, 因为一维数组地址可理解为首元素地址, 不加[] (说法不准确, 必须加)对于int/char等基本类型的数组, 加不加均正确; 但对于用户自定义的class, 则必须加, 错误例子见后

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

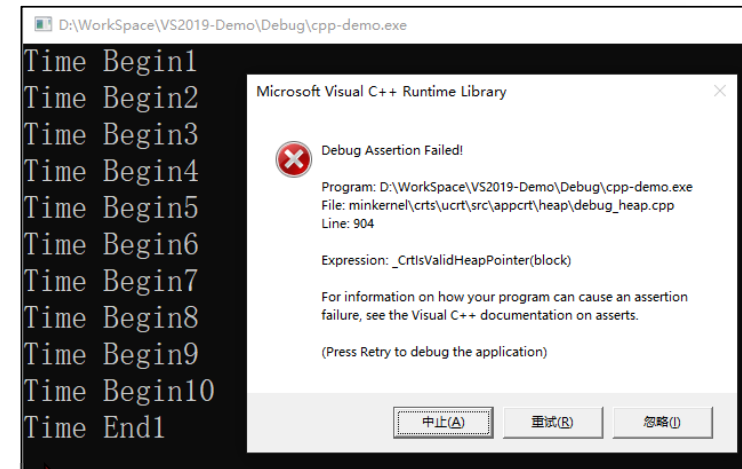
class Time {
private:
    int hour, minute, second;
public:
    Time(int h=0, int m=0, int s=0);
    ~Time();
};

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 = new Time[10] {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    delete t1;
    return 0;
} //篇幅问题, 假设申请成功
```

```
int main()
{
    Time *t1 = new Time[10] {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    delete []t1;
    return 0;
} //篇幅问题, 假设申请成功
```



```
Time Begin1
Time Begin2
Time Begin3
Time Begin4
Time Begin5
Time Begin6
Time Begin7
Time Begin8
Time Begin9
Time Begin10
Time End10
Time End9
Time End8
Time End7
Time End6
Time End5
Time End4
Time End3
Time End2
Time End1
```



§ 13. 动态内存申请

3. 内存的动态申请与释放

★ C : 可通过强制类型转换将void型的指针转为其它类型

★ C++ : 申请时自动确定类型

```
#include <iostream>
using namespace std;
int main()
{   int *p;
    p = (int *)malloc(10*sizeof(int));
    if (p==NULL) {
        cout << "No Memory" << endl;
        return -1;
    }
    cout << *p << endl; //观察运行结果, 是否进行了初始化
    free(p);
    return 0;
}
```

强制类型转换

申请10个int型的变量
空间可以直接写成
malloc(40), 但不建议,
因为适应型差

```
int main()
{   int *p;
    p = new(nothrow) int[10];
    if (p==NULL) {
        cout << "No Memory" << endl;
        return 0;
    }
    ...
    delete p;
    ...
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{   int *p;
    p = (int *)calloc(10, sizeof(int));
    if (p==NULL) {
        cout << "No Memory" << endl;
        return -1;
    }
    cout << *p << endl; //观察运行结果, 是否进行了初始化
    free(p);
    return 0;
}
```

强制类型转换

申请10个int型的变量
空间可以直接写成
calloc(10, 4), 但不建
议, 因为适应型差

malloc(10*sizeof(int))
calloc(10, sizeof(int))
realloc(NULL, 10*sizeof(int))
都表示申请连续的40字节空间,
结果一样, 只是表示方式有差别
以及是否初始化有差别



§ 13. 动态内存申请

3. 内存的动态申请与释放

★ 静态数据区、动态数据区、动态内存分配区(称为堆空间)的地址各不相同

例：观察下列程序的输出

```
#include <iostream>
#include <cstdlib>
using namespace std;
int a;
int main()
{
    int b;
    int *c;
    c = (int *)malloc(sizeof(int)); //一个int
    if (c==NULL) {
        cout << "申请int失败" << endl;
        return -1;
    }
    cout << &a <<endl;
    cout << &b <<endl;
    cout << &c << ' ' << c <<endl;
    free(c);

    return 0;
}
```

问：静态数据区/动态数据区/
堆空间的大小如何？如何
验证？



§ 13. 动态内存申请

3. 内存的动态申请与释放

★ 打开Windows的任务管理器，观察下列程序的运行结果，理解“动态申请与释放”的概念

例：观察下列程序的输出

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

int main()
{
    char *p;
    p = (char *)malloc(100 * 1024 * 1024 * sizeof(char)); //100MB
    if (p==NULL) {
        cout << "申请空间失败，请减少申请值后重试" << endl;
        return -1;
    }
    cout << "申请完成，请在任务管理器中观察内存占用" << endl;
    getchar(); //暂停，不释放内存

    free(p);
    cout << "释放完成，请在任务管理器中观察占用" << endl;
    getchar(); //暂停，不退出程序
    return 0;
}
```

如果是Linux下测试，则使用
top命令观察内存占用，具体
请自行查阅资料



§ 13. 动态内存申请

3. 内存的动态申请与释放

★ C/C++ : 动态申请返回的指针可以进行指针运算, 但释放时必须给出申请返回时的首地址, 否则释放时会出错

(以下几种情况均是编译不错执行错, 用多编译器观察运行结果)

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

```
int main()
{
    int i, *p;
    p = &i;
    free(p);
    return 0;
}
```

//p不是动态申请的空间

```
#include <iostream>
//不需要包含头文件stdlib
using namespace std;
```

```
int main()
{
    int i, *p;
    p = &i;
    delete p;
    return 0;
}
```

//p不是动态申请的空间

```
#include <iostream>
#include <stdlib.h>
using namespace std;
```

```
int main()
{
    int *p;
    p=(int*)malloc(sizeof(int)); //未判断
    p++;
    free(p);
    return 0;
}
```

//p已不指向动态申请的空间

```
#include <iostream>
//不需要包含头文件stdlib
using namespace std;
```

```
int main()
{
    int *p;
    p = new(nothrow) int; //未判断
    p++;
    delete p;
    return 0;
}
```

//p已不指向动态申请的空间

特别说明:

- 1、虽然申请一个int空间不可能申请失败, 但从程序规范角度出发, 要求每次申请后均需要判断申请是否成功
- 2、本例及后续课件中, 为了节约空间, 部分示例程序省略了是否成功的判断, 特此说明



§ 13. 动态内存申请

3. 内存的动态申请与释放

★ C/C++ : 动态内存申请的空间若不释放, 则会造成内存泄露, 这种情况不会导致即时错误, 但最终会耗尽内存

```
#include <iostream>
#include <cstdlib> //malloc系列函数用
using namespace std;
int main()
{
    char *p;
    int num = 0;
    while(1) {
        p = (char *)malloc(1024*1024*sizeof(char));
        if (p==NULL)
            break;
        num ++;
    }
    cout << num << " MB" << endl;
    return 0;
}
```

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

int main()
{
    char *p;
    int num = 0;
    while(1) {
        p = new(nothrow) char[1024*1024];
        if (p==NULL)
            break;
        num ++;
    }
    cout << num << " MB" << endl;
    return 0;
}
```

耗尽内存的例子:

- 1、每次申请1MB空间
- 2、申请完成后不释放, 且p不再指向, 导致内存泄露
- 3、循环1-2至内存耗尽



§ 13. 动态内存申请

3. 内存的动态申请与释放

★ C/C++ : 动态内存申请的空间若不释放, 则会造成**内存泄露**, 这种情况不会导致即时错误, 但最终会**耗尽内存**

```
#include <iostream>
using namespace std;
int main()
{
    char *p;
    int count = 0;
    while (1) {
        try {
            p = new char[1024 * 1024];
        }
        catch (const bad_alloc &mem_fail) {
            cout << mem_fail.what() << endl; //打印原因
            break;
        }
        count++;
    }
    cout << count << " MB" << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    char *p;
    int count = 0;
    try {
        while (1) {
            p = new char[1024 * 1024];
            count++;
        }
    }
    catch (const bad_alloc &mem_fail) {
        cout << mem_fail.what() << endl; //打印原因
    }
    cout << count << " MB" << endl;
    return 0;
}
```

耗尽内存的例子:

- 1、每次申请1MB空间
- 2、申请完成后不释放, 且p不再指向, 导致内存泄露
- 3、循环1-2至内存耗尽

在新版C++标准中, new申请失败会抛出异常bad_alloc, 需要使用try-catch来处理异常



§ 13. 动态内存申请

3. 内存的动态申请与释放

★ C/C++ : 动态内存申请的空间若不释放, 则会造成内存泄露, 这种情况不会导致即时错误, 但最终会耗尽内存
(坚决反对此种用法, 且不是所有的操作系统都支持)

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

```
int main()
{
    int *p;
    p=(int *)malloc(...);
    ...;
    return 0;
}
```



p所申请的空间在程序运行结束后由操作系统回收

```
#include <iostream>
//不需要包含头文件stdlib
using namespace std;
```

```
int main()
{
    int *p;
    p = new ...;
    ...;
    return 0;
}
```



p所申请的空间在程序运行结束后由操作系统回收

```
#include <iostream>
#include <stdlib.h>
using namespace std;
```

```
int main()
{
    int *p;
    /* 假设ATM机取款 */
    for(;;) {
        ...; //等待用户刷卡
        p=(int*)malloc(...);
        ...;
    }
    return 0;
}
```



会逐渐耗尽内存

```
#include <iostream>
//不需要包含头文件stdlib
using namespace std;
```

```
int main()
{
    int *p;
    /* 假设ATM机取款 */
    for(;;) {
        ...; //等待用户刷卡
        p = new ...;
        ...;
    }
    return 0;
}
```



会逐渐耗尽内存



§ 13. 动态内存申请

3. 内存的动态申请与释放

★ C : 对简单变量、一维/多维数组没有分别, 只算总大小

★ C++ : 不同情况申请方法不同

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

int main()
{
    int *p;
    p = (int *)malloc(sizeof(int)); //未判断
    *p = 10;
    cout << *p << endl;
    free(p); //记得释放
    return 0;
}
```

```
#include <iostream>

using namespace std;

int main()
{
    int *p;
    p = new(nothrow) int; //未判断
    *p = 10;
    cout << *p << endl;
    delete p; //记得释放
    return 0;
}
```

申请一个int型空间



§ 13. 动态内存申请

3. 内存的动态申请与释放

★ C : 对简单变量、一维/多维数组没有分别, 只算总大小

★ C++ : 不同情况申请方法不同

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

int main()
{
    int i, *p;
    p = (int *)malloc(10*sizeof(int));
    for(i=0; i<10; i++)
        p[i] = (i+1)*(i+1); //赋初值
    for(i=0; i<10; i++)
        cout << *(p+i) << endl; //打印
    free(p); //记得释放
    return 0;
} //未判断申请是否成功
```

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

int main()
{
    int i, *p;
    p = new(nothrow) int[10];
    for(i=0; i<10; i++)
        p[i] = (i+1)*(i+1); //赋初值
    for(i=0; i<10; i++)
        cout << *(p+i) << endl; //打印
    delete []p; //记得释放
    return 0;
} //未判断申请是否成功
```

申请10个int型空间,
当一维数组用
指针法/下标法均可



§ 13. 动态内存申请

3. 内存的动态申请与释放

★ C : 对简单变量、一维/多维数组没有分别, 只算总大小

★ C++ : 不同情况申请方法不同

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

int main()
{
    int i, *p, *head;
    p = (int *)malloc(10*sizeof(int));
    head = p;
    for(i=0; i<10; i++)
        *p++ = (i+1)*(i+1); //赋初值
    for(p=head; p-head<10; p++)
        cout << *p << endl; //打印
    free(head); //记得释放
    return 0;
} //未判断申请是否成功
```

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

int main()
{
    int i, *p, *head;
    p = new(nothrow) int[10];
    head = p;
    for(i=0; i<10; i++)
        *p++ = (i+1)*(i+1); //赋初值
    for(p=head; p-head<10; p++)
        cout << *p << endl; //打印
    delete []head; //记得释放
    return 0;
} //未判断申请是否成功
```

申请10个int当一维数组用
用head记住申请的首地址,
便于复位和释放, p可++/--



§ 13. 动态内存申请

3. 内存的动态申请与释放

★ C : 对简单变量、一维/多维数组没有分别, 只算总大小

★ C++ : 不同情况申请方法不同

申请12个int型空间
当做二维数组使用
指针法/下标法均可

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

int main()
{
    int i, j, (*p)[4];
    p=(int (*)[4]) malloc(3*4*sizeof(int));
    for(i=0; i<3; i++)
        for(j=0; j<4; j++)
            p[i][j] = i*4+j; //赋初值
    for(i=0; i<3; i++) {
        for(j=0; j<4; j++)
            cout << *((p+i)+j) << ' ';
        cout << endl; //每行加回车
    }
    free(p); //记得释放
    return 0;
} //未判断申请是否成功
```

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

int main()
{
    int i, j, (*p)[4];
    p = new(nothrow) int[3][4];
    for(i=0; i<3; i++)
        for(j=0; j<4; j++)
            p[i][j] = i*4+j; //赋初值
    for(i=0; i<3; i++) {
        for(j=0; j<4; j++)
            cout << *((p+i)+j) << ' ';
        cout << endl; //每行加回车
    }
    delete []p; //记得释放
    return 0;
} //未判断申请是否成功
```



§ 13. 动态内存申请

3. 内存的动态申请与释放

★ C : 对简单变量、一维/多维数组没有分别, 只算总大小

★ C++ : 不同情况申请方法不同

申请12个int当二维使用
p为行指针, p_element为
元素指针, head记住首址

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

int main()
{
    int i, j, (*p)[4], (*head)[4], *p_element;
    head = p = (int (*)[4]) malloc(3*4*sizeof(int));
    for(i=0; i<3; i++)
        for(j=0; j<4; j++)
            p[i][j] = i*4+j; //赋初值
    for(p=head; p-head<3; p++) {
        for(p_element=*p; p_element-*p<4; p_element++)
            cout << *p_element << ' ';
        cout << endl; //每行加回车
    }
    free(head); //记得释放
    return 0;
} //未判断申请是否成功
```

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

int main()
{
    int i, j, (*p)[4], (*head)[4], *p_element;
    head = p = new(nothrow) int[3][4];
    for(i=0; i<3; i++)
        for(j=0; j<4; j++)
            p[i][j] = i*4+j; //赋初值
    for(p=head; p-head<3; p++) {
        for(p_element=*p; p_element-*p<4; p_element++)
            cout << *p_element << ' ';
        cout << endl; //每行加回车
    }
    delete []head; //记得释放
    return 0;
} //未判断申请是否成功
```



§ 13. 动态内存申请

3. 内存的动态申请与释放

★ C/C++ : 动态申请的内存, 只能通过首指针释放一次, 若重复释放, 则会导致运行出错

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

int main()
{
    int *p;
    p = (int *)malloc(sizeof(int)); //未判断
    *p = 10;
    cout << *p << endl;
    free(p); //释放
    free(p); //再次释放, 致运行出错

    return 0;
}
```

```
#include <iostream>

using namespace std;

int main()
{
    int *p;
    p = new(nothrow) int; //未判断
    *p = 10;
    cout << *p << endl;
    delete p; //释放
    delete p; //再次释放, 致运行出错

    return 0;
}
```

重复释放导致错误



§ 13. 动态内存申请

3. 内存的动态申请与释放

★ C/C++ : 如果出现需要嵌套进行动态内存申请的情况, 则按从外到内的顺序进行申请, 反序进行释放

嵌套申请 先student变量, 再name成员
<pre>#include <iostream> #include <cstdlib> using namespace std; struct student { int num; char *name; }; int main() { student *s1; s1 = (student *)malloc(sizeof(student)); //申请8字节 s1->name = (char *)malloc(6*sizeof(char)); //申请6字节 s1->num = 1001; strcpy(s1->name, "zhang"); cout << s1->num << ":" << s1->name << endl; free(s1->name); //释放6字节 free(s1); //释放8字节 return 0; } //为节约篇幅, 未判断申请是否成功</pre>

嵌套申请 先student变量, 再name成员
<pre>#include <iostream> using namespace std; struct student { int num; char *name; }; int main() { student *s1; s1 = new(nothrow) student; //申请8字节 s1->name = new(nothrow) char[6]; //申请6字节 s1->num = 1001; strcpy(s1->name, "zhang"); cout << s1->num << ":" << s1->name << endl; delete []s1->name; //释放6字节 delete s1; //释放8字节 return 0; } //为节约篇幅, 未判断申请是否成功</pre>



★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
struct student {  
    int num;  
    char *name;  
};  
  
int main()  
{  
    student *s1;  
    s1 = (student *)malloc(sizeof(student)); //申请8字节  
    s1->name = (char *)malloc(6*sizeof(char)); //申请6字节  
    s1->num = 1001;  
    strcpy(s1->name, "zhang");  
    cout << s1->num << ":" << s1->name << endl;  
    free(s1->name); //释放6字节  
    free(s1); //释放8字节  
    return 0;  
} //为节约篇幅，未判断申请是否成功
```



★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
struct student {  
    int num;  
    char *name;  
};  
  
int main()  
{ student *s1;  
    s1 = (student *)malloc(sizeof(student)); //申请8字节  
    s1->name = (char *)malloc(6*sizeof(char)); //申请6字节  
    s1->num = 1001;  
    strcpy(s1->name, "zhang");  
    cout << s1->num << ":" << s1->name << endl;  
    free(s1->name); //释放6字节  
    free(s1); //释放8字节  
    return 0;  
} //为节约篇幅，未判断申请是否成功
```

s1	2000 2003	???
----	--------------	-----



★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
struct student {
```

```
    int num;
```

```
    char *name;
```

```
};
```

```
int main()
```

```
{    student *s1;
```

```
    s1 = (student *)malloc(sizeof(student)); //申请8字节
```

```
    s1->name = (char *)malloc(6*sizeof(char)); //申请6字节
```

```
    s1->num = 1001;
```

```
    strcpy(s1->name, "zhang");
```

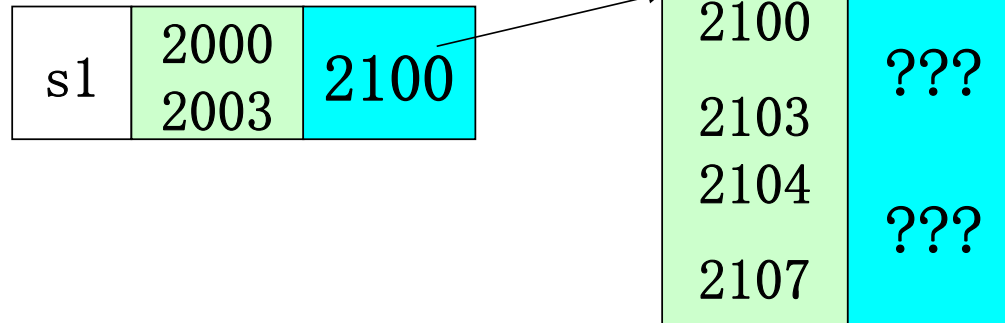
```
    cout << s1->num << ":" << s1->name << endl;
```

```
    free(s1->name); //释放6字节
```

```
    free(s1); //释放8字节
```

```
    return 0;
```

```
} //为节约篇幅，未判断申请是否成功
```





★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
struct student {  
    int num;  
    char *name;  
};
```

```
int main()
```

```
{ student *s1;
```

```
  s1 = (student *)malloc(sizeof(student)); //申请8字节
```

```
  s1->name = (char *)malloc(6*sizeof(char)); //申请6字节
```

```
  s1->num = 1001;
```

```
  strcpy(s1->name, "zhang");
```

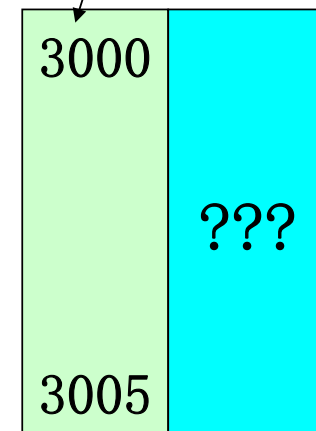
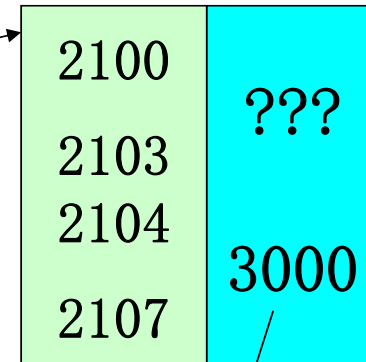
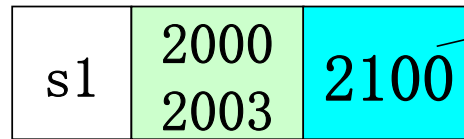
```
  cout << s1->num << ":" << s1->name << endl;
```

```
  free(s1->name); //释放6字节
```

```
  free(s1); //释放8字节
```

```
  return 0;
```

```
} //为节约篇幅，未判断申请是否成功
```





★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
struct student {
```

```
    int num;
```

```
    char *name;
```

```
};
```

```
int main()
```

```
{    student *s1;
```

```
    s1 = (student *)malloc(sizeof(student)); //申请8字节
```

```
    s1->name = (char *)malloc(6*sizeof(char)); //申请6字节
```

```
    s1->num = 1001;
```

```
    strcpy(s1->name, "zhang");
```

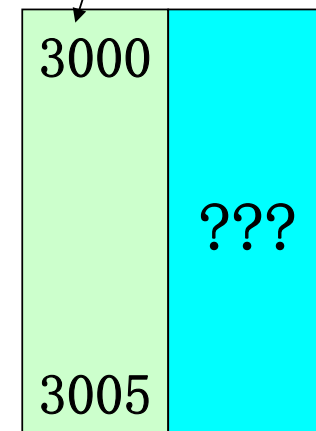
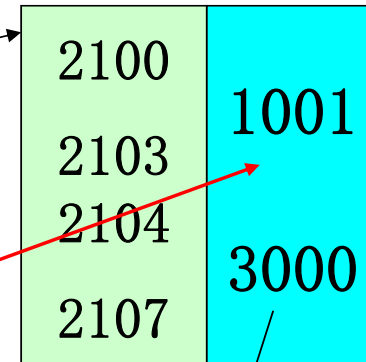
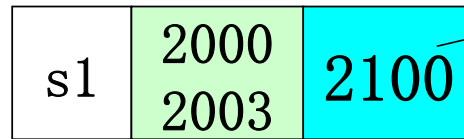
```
    cout << s1->num << ":" << s1->name << endl;
```

```
    free(s1->name); //释放6字节
```

```
    free(s1); //释放8字节
```

```
    return 0;
```

```
} //为节约篇幅，未判断申请是否成功
```





★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
struct student {  
    int num;  
    char *name;  
};
```

```
int main()
```

```
{ student *s1;
```

```
s1 = (student *)malloc(sizeof(student)); //申请8字节
```

```
s1->name = (char *)malloc(6*sizeof(char)); //申请6字节
```

```
s1->num = 1001;
```

```
strcpy(s1->name, "zhang");
```

```
cout << s1->num << ":" << s1->name << endl;
```

```
free(s1->name); //释放6字节
```

```
free(s1); //释放8字节
```

```
return 0;
```

```
} //为节约篇幅，未判断申请是否成功
```

s1	2000 2003	2100
----	--------------	------

2100	1001
2103	
2104	
2107	3000

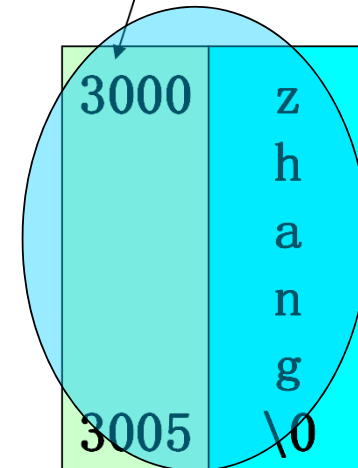
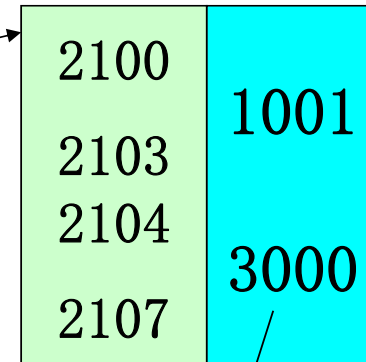
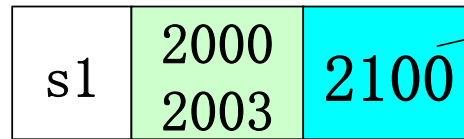
3000	z
	h
	a
	n
	g
3005	\0



★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
struct student {  
    int num;  
    char *name;  
};
```

```
int main()  
{  
    student *s1;  
    s1 = (student *)malloc(sizeof(student)); //申请8字节  
    s1->name = (char *)malloc(6*sizeof(char)); //申请6字节  
    s1->num = 1001;  
    strcpy(s1->name, "zhang");  
    cout << s1->num << ":" << s1->name << endl;  
    free(s1->name); //释放6字节  
    free(s1); //释放8字节  
    return 0;  
} //为节约篇幅，未判断申请是否成功
```

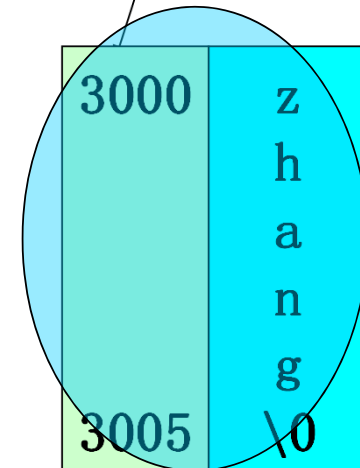
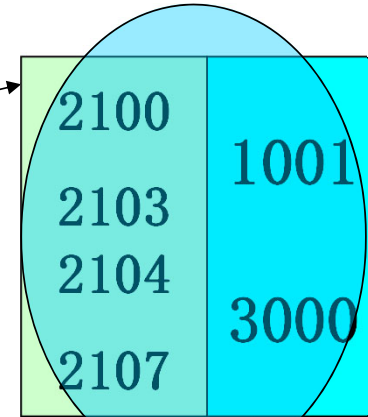
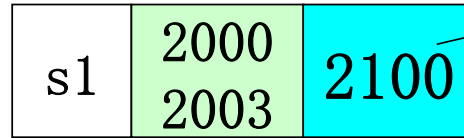




★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
struct student {  
    int num;  
    char *name;  
};
```

```
int main()  
{  
    student *s1;  
    s1 = (student *)malloc(sizeof(student)); //申请8字节  
    s1->name = (char *)malloc(6*sizeof(char)); //申请6字节  
    s1->num = 1001;  
    strcpy(s1->name, "zhang");  
    cout << s1->num << ":" << s1->name << endl;  
    free(s1->name); //释放6字节  
    free(s1); //释放8字节  
    return 0;  
} //为节约篇幅，未判断申请是否成功
```





★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
struct student {  
    int num;  
    char *name;  
};
```

```
int main()  
{ student *s1;
```

s1自身所占4字节
由操作系统回收

```
s1 = (student *)malloc(sizeof(student)); //申请8字节
```

```
s1->name = (char *)malloc(6*sizeof(char)); //申请6字节
```

```
s1->num = 1001;
```

```
strcpy(s1->name, "zhang");
```

```
cout << s1->num << ":" << s1->name << endl;
```

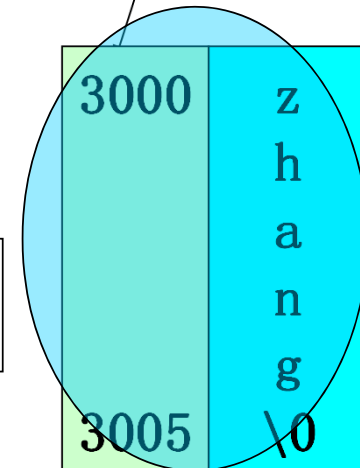
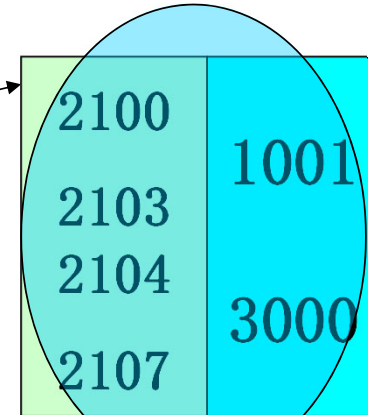
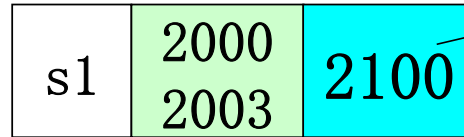
```
free(s1->name); //释放6字节
```

```
free(s1); //释放8字节
```

free的顺序不能反

```
return 0;
```

```
} //为节约篇幅，未判断申请是否成功
```





§ 13. 动态内存申请

3. 内存的动态申请与释放

★ realloc专题讨论

函数形式:

```
void *realloc(void *ptr, unsigned newsize);
```

- 表示为指针ptr重新申请newsize大小的空间
- ptr必须是malloc/calloc/realloc返回的指针
- 如果ptr为NULL, 则等同于malloc
- 如果ptr非NULL, newsize为0, 则等同于free, 并返回NULL
- 新老空间可重合, 也可能不重合, 若不重合, 原空间原有内容会被复制到新空间, 再释放原空间
- 对申请到的空间不做初始化操作
- 若申请不到, 则返回NULL (此时已有指针ptr不释放)



§ 13. 动态内存申请

3. 内存的动态申请与释放

★ realloc专题讨论

- 如果ptr为NULL，则等同于malloc
- 对申请到的空间不做初始化操作

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

int main()
{
    int *p;
    p = (int *)realloc(NULL, 10 * sizeof(int));
    if (p==NULL) {
        cout << "No Memory" << endl;
        return -1;
    }

    for(int i=0; i<10; i++)
        cout << p[i] << endl;
    free(p);
    return 0;
}
```

强制类型转换

等价于 malloc(10 * sizeof(int))

观察运行结果，是否进行了初始化



§ 13. 动态内存申请

3. 内存的动态申请与释放

★ realloc专题讨论

- 表示为指针ptr重新申请newsize大小的空间
- ptr必须是malloc/calloc/realloc返回的指针
- 新老空间可重合，也可能不重合，若不重合，原空间原有内容会被复制到新空间，再释放原空间

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{   int i, *p, *q;
    p = (int *)malloc(10 * sizeof(int)); //省略了是否申请成功的判断
    cout << p << endl; //地址
    for (i=0; i<10; i++)
        p[i] = i*i; //为10个数赋初值

    q = (int *)realloc(p, 20 * sizeof(int)); //省略了是否申请成功的判断
    cout << p << ' ' << q << endl; //观察地址是否相同
    for (i=0; i<20; i++)
        cout << p[i] << ' '; //观察前10个和后10个数
    cout << endl;
    free(q);
    return 0;
}
```

此处换成 ++p / p+1等形式，
多编译器观察程序的运行结果



§ 13. 动态内存申请

3. 内存的动态申请与释放

★ realloc专题讨论

- 新老空间可重合，也可能不重合，若不重合，原空间原有内容会被复制到新空间，再释放原空间

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

int main()
{
    int *p, *q;
    p = (int *)malloc(10 * sizeof(int)); //省略了是否申请成功的判断
    cout << p << endl;

    q = (int *)realloc(p, 20 * sizeof(int)); //省略了是否申请成功的判断
    cout << p << ' ' << q << endl;

    free(p);
    free(q);

    return 0;
}
```

- | | |
|----------|-------------------------|
| free(p); | 1、多编译器观察程序的运行结果 |
| free(q); | 2、注释掉free(p)，再观察结果 |
| | 3、此处换成5(小于原大小即可)，再重复1、2 |



§ 13. 动态内存申请

3. 内存的动态申请与释放

★ realloc专题讨论

- 如果ptr非NULL, newsize为0, 则等同于free, 并返回NULL

//先打开Windows的任务管理器, 再观察程序的运行

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

int main()
{
    char *p, *q;

    p = (char *)malloc(100 * 1024 * 1024 * sizeof(char)); //100MB, 此处要保证成功
    if (p == NULL) {
        cout << "申请空间失败, 请减少申请值后重试" << endl;
        return -1;
    }
    cout << "申请完成, 请在任务管理器中观察内存占用" << endl;
    getchar(); //暂停, 不释放内存

    q = (char *)realloc(p, 0); //0字节
    cout << (q==NULL ? "NULL" : q) << endl; //NULL不能直接打印
    cout << "realloc 0字节完成, 请在任务管理器中观察内存占用" << endl;
    getchar(); //暂停, 不退出程序

    return 0;
}
```



§ 13. 动态内存申请

3. 内存的动态申请与释放

★ realloc专题讨论

- 若申请不到，则返回NULL（此时已有指针ptr不释放）

//先打开Windows的任务管理器，再观察程序的运行

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

```
int main()
{
```

```
    char *p, *q;
```

```
    p = (char *)malloc(100 * 1024 * 1024 * sizeof(char)); //100MB，此处要保证成功
```

```
    if (p == NULL) {
        cout << "申请空间失败，请减少申请值后重试" << endl;
        return -1;
    }
```

```
    cout << "申请完成，请观察内存占用" << endl;
    getchar(); //暂停，不释放内存
```

问题：为什么加U？

```
    q = (char *)realloc(p, 2048U * 1024 * 1024 * sizeof(char)); //2GB，此处要保证失败，如果不失败，继续增大值
```

```
    if (q==NULL) //如果不提示失败，2048U继续增大
        cout << "realloc失败，请观察内存占用" << endl;
    getchar(); //暂停，不退出程序
```

```
    free(p);
    return 0;
```

```
}
```

realloc的不正确用法(网上常见):
传入指针和返回指针用同一个时，
一旦申请失败，原内存就丢失了!!!!



§ 13. 动态内存申请

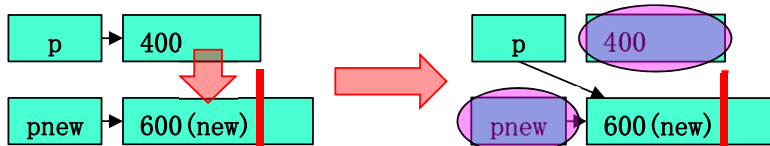
3. 内存的动态申请与释放

★ realloc专题讨论

★ C++中没有类似于realloc的renew，如果需要扩大/缩小原动态申请空间，要自己处理

```
int *renew(int *p, int oldsize, int newsize)
{
    int *pnew;
    pnew = new(nothrow) int[newsize]; //申请新
    for(i=0; i<oldsize; i++) //原内容=>新
        pnew[i] = p[i];
    delete p; //释放原空间(main中的new int [100])
    p = pnew; //原指针p指向新空间
    return p; //pnew已无用，但不能delete
}
```

```
main中: int *p = new int[100];
...
p = renew(p, 100, 150);
...
delete []p; //释放空间(renew中的new int[150])
```



重要提示:

由renew这个例子可看出，C/C++的动态内存申请和释放，可能会在不同函数间进行；因此，在大型程序中要做到无内存泄露是一件很困难的事情，这也是C/C++相比较于其它语言的难点所在!!!

左侧是一个不完整的renew，缺以下情况：

- newsize<oldsize的情况
- 申请失败的情况
- 其它基类型的指针(可重载/模板解决)



§ 13. 动态内存申请

3. 内存的动态申请与释放

★ 在C/C++的动态申请混合使用时，可能会出现问题

```
//例: C++的动态申请, 内嵌string类 (C++特有)
#include <iostream>
#include <string>    //C++特有的string类需要
using namespace std;
```

```
struct student {
    string name; //C++特有的string类
    int num;
    char sex;
};
```

```
int main()
{
    student *p;
    p = new(nothrow) student;
    if (p==NULL)    //加判断保证程序的正确性
        return -1;
    p->name = "Wang Fun";
    p->num = 10123;
    p->sex = 'm';
    cout << p->name << endl
         << p->num << endl
         << p->sex << endl;
    delete p;
    return 0;
}
```

本例运行正确

```
//思考: 下面的例子说明了什么?
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string s1;
    cout << sizeof(s1) << endl;
    s1 = "***"; //此处长度超sizeof
    cout << sizeof(s1) << endl;
    return 0;
}
```

```
//例: 变化, C方式的动态申请, 内嵌string类 (C++特有)
#include <iostream>
#include <string>    //C++特有的string类需要
#include <stdlib.h>
using namespace std;
```

```
struct student {
    string name; //C++特有的string类
    int num;
    char sex;
};
```

```
int main()
{
    student *p;
    p = (student *)malloc(sizeof(student));
    if (p==NULL)    //加判断保证程序的正确性
        return -1;
    p->name = "Wang Fun";
    p->num = 10123;
    p->sex = 'm';
    cout << p->name << endl
         << p->num << endl
         << p->sex << endl;
    free(p);
    return 0;
}
```

多编译器运行, 哪些编译器下运行错误? 表现是什么?
为什么?
(提示: 和左侧的思考题、下面的建议结合在一起思考)

建议:
C++下的动态内存申请不建议
采用C函数方式, 不要为了
realloc的便捷性而降级



§ 13. 动态内存申请

3. 内存的动态申请与释放

例：用动态内存申请方式建立一个有5个结点的链表，学生的基本信息从键盘进行输入

假设键盘输入为

Zhang 1001 m

Li 1002 f

Wang 1003 m

Zhao 1004 m

Qian 1005 f

```
struct student {  
    string name;  
    int num;  
    char sex;  
    struct student *next; //指向结构体自身的指针(下个结点)  
};
```

成员类型不允许是自身的结构体类型，
但可以是自身结构体类型的指针
(因为指针占用空间已知)



例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

初始状态

```
for(i=0; i<5; i++) {
```

```
    if (i>0)
```

```
        q=p;
```

```
    p = new(nothrow) student;
```

```
    if (p==NULL)
```

```
        return -1;
```

```
    if (i==0)
```

```
        head = p; //head指向第1个结点
```

```
    else
```

```
        q->next = p;
```

```
    cout << "请输入第" << i+1 << "个人的基本信息" << endl;
```

```
    cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
```

```
    p->next = NULL;
```

```
}
```

head	2000	???
	2003	
p	2100	???
	2103	
q	2200	???
	2203	



例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
  for(i=0; i<5; i++) {
```

```
    if (i>0)
```

```
        q=p;
```

```
    p = new(nothrow) student;
```

```
    if (p==NULL)
```

```
        return -1;
```

```
    if (i==0)
```

```
        head = p; //head指向第1个结点
```

```
    else
```

```
        q->next = p;
```

```
    cout << "请输入第" << i+1 << "个人的基本信息" << endl;
```

```
    cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
```

```
    p->next = NULL;
```

```
}
```

i=0的循环

head	2000 2003	???
p	2100 2103	3000
q	2200 2203	???

3000	???
------	-----



例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
  for(i=0; i<5; i++) {
```

```
    if (i>0)
```

```
      q=p;
```

```
    p = new(nothrow) student;
```

```
    if (p==NULL)
```

```
      return -1;
```

```
    if (i==0)
```

```
      head = p; //head指向第1个结点
```

```
    else
```

```
      q->next = p;
```

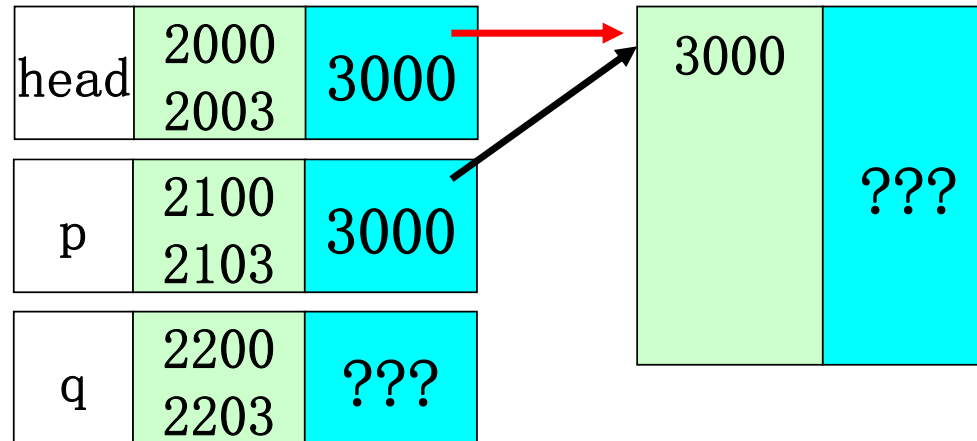
```
    cout << "请输入第" << i+1 << "个人的基本信息" << endl;
```

```
    cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
```

```
    p->next = NULL;
```

```
  }
```

i=0的循环





例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
  for(i=0; i<5; i++) {
```

```
    if (i>0)
```

```
      q=p;
```

```
    p = new(nothrow) student;
```

```
    if (p==NULL)
```

```
      return -1;
```

```
    if (i==0)
```

```
      head = p; //head指向第1个结点
```

```
    else
```

```
      q->next = p;
```

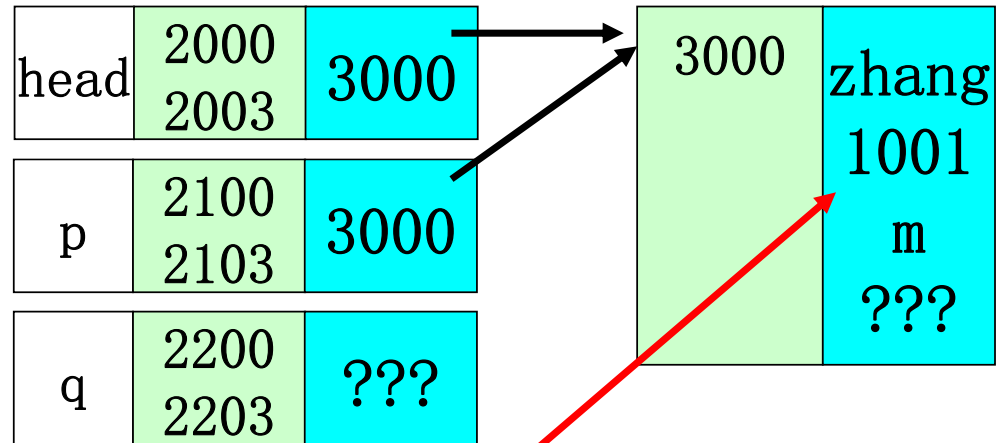
```
    cout << "请输入第" << i+1 << "个人的基本信息" << endl;
```

```
    cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
```

```
    p->next = NULL;
```

```
  }
```

i=0的循环





例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
  for(i=0; i<5; i++) {
```

```
    if (i>0)
```

```
      q=p;
```

```
    p = new(nothrow) student;
```

```
    if (p==NULL)
```

```
      return -1;
```

```
    if (i==0)
```

```
      head = p; //head指向第1个结点
```

```
    else
```

```
      q->next = p;
```

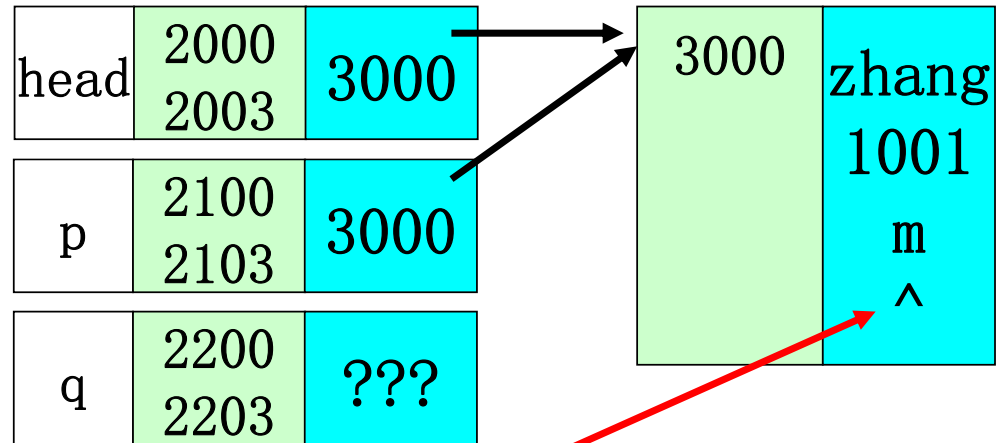
```
    cout << "请输入第" << i+1 << "个人的基本信息" << endl;
```

```
    cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
```

```
    p->next = NULL;
```

```
  }
```

i=0的循环





例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
  for(i=0; i<5; i++) {
```

```
    if (i>0)
```

```
        q=p;
```

```
    p = new(nothrow) student;
```

```
    if (p==NULL)
```

```
        return -1;
```

```
    if (i==0)
```

```
        head = p; //head指向第1个结点
```

```
    else
```

```
        q->next = p;
```

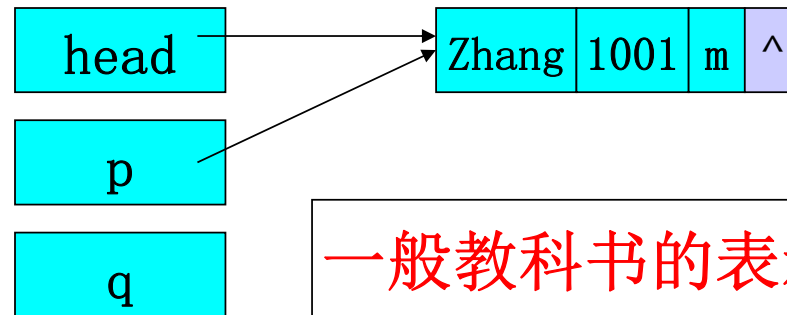
```
    cout << "请输入第" << i+1 << "个人的基本信息" << endl;
```

```
    cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
```

```
    p->next = NULL;
```

```
}
```

i=0的循环



一般教科书的表示



例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
  for(i=0; i<5; i++) {
```

```
    if (i>0)
```

```
        q=p;
```

```
    p = new(nothrow) student;
```

```
    if (p==NULL)
```

```
        return -1;
```

```
    if (i==0)
```

```
        head = p; //head指向第1个结点
```

```
    else
```

```
        q->next = p;
```

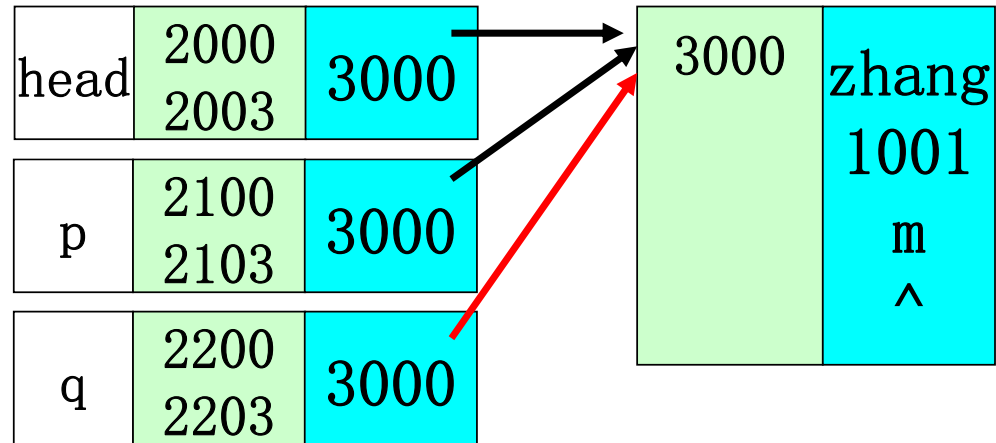
```
    cout << "请输入第" << i+1 << "个人的基本信息" << endl;
```

```
    cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
```

```
    p->next = NULL;
```

```
}
```

i=1的循环





例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
  for(i=0; i<5; i++) {
```

```
    if (i>0)
```

```
      q=p;
```

```
      p = new(nothrow) student;
```

```
      if (p==NULL)
```

```
        return -1;
```

```
      if (i==0)
```

```
        head = p; //head指向第1个结点
```

```
      else
```

```
        q->next = p;
```

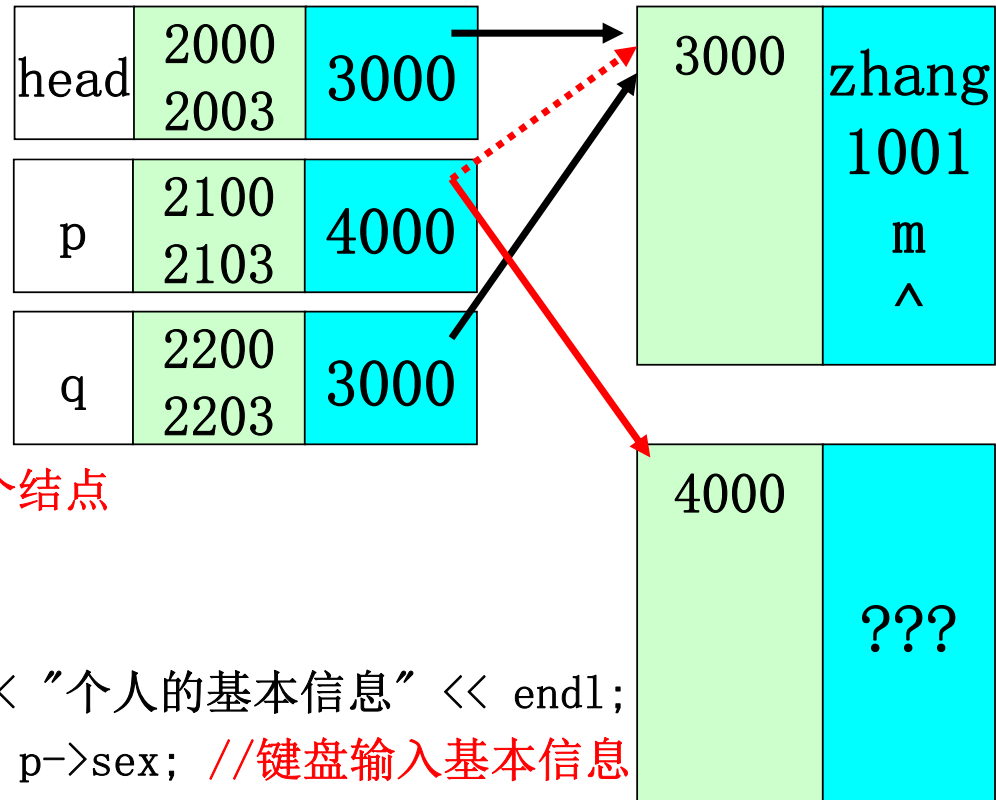
```
        cout << "请输入第" << i+1 << "个人的基本信息" << endl;
```

```
        cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
```

```
        p->next = NULL;
```

```
    }
```

i=1的循环





例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
for(i=0; i<5; i++) {
```

```
    if (i>0)
```

```
        q=p;
```

```
    p = new(nothrow) student;
```

```
    if (p==NULL)
```

```
        return -1;
```

```
    if (i==0)
```

```
        head = p; //head指向第1个结点
```

```
    else
```

```
        q->next = p;
```

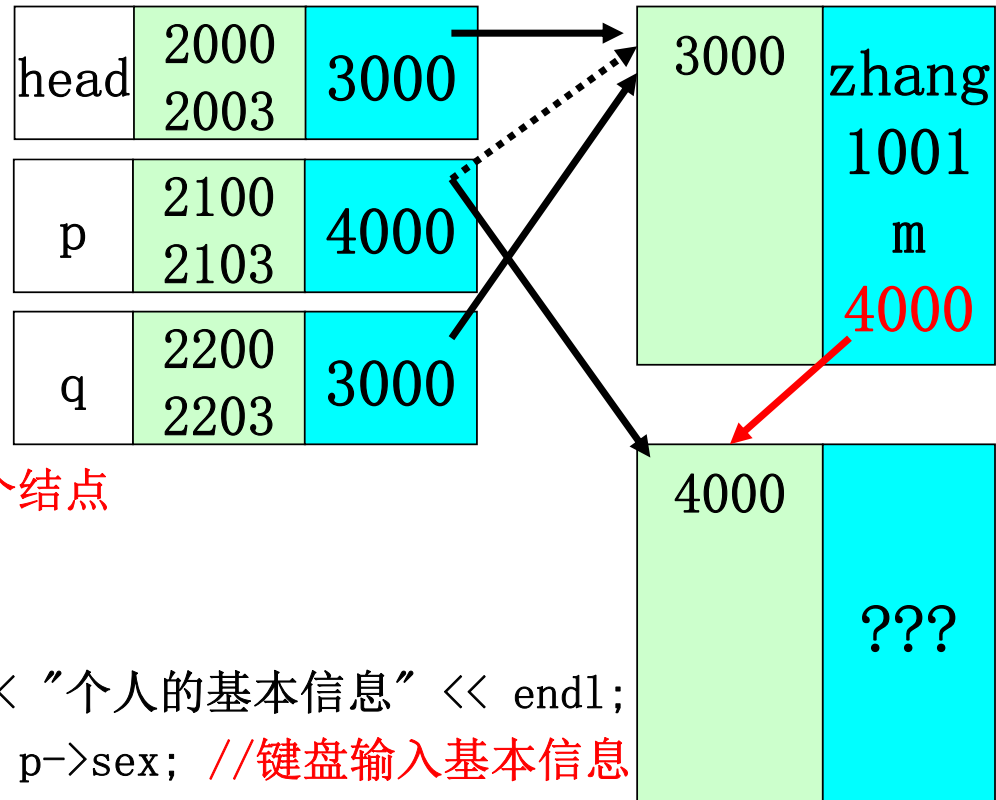
```
    cout << "请输入第" << i+1 << "个人的基本信息" << endl;
```

```
    cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
```

```
    p->next = NULL;
```

```
}
```

i=1的循环





例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
  for(i=0; i<5; i++) {
```

```
    if (i>0)
```

```
      q=p;
```

```
    p = new(nothrow) student;
```

```
    if (p==NULL)
```

```
      return -1;
```

```
    if (i==0)
```

```
      head = p; //head指向第1个结点
```

```
    else
```

```
      q->next = p;
```

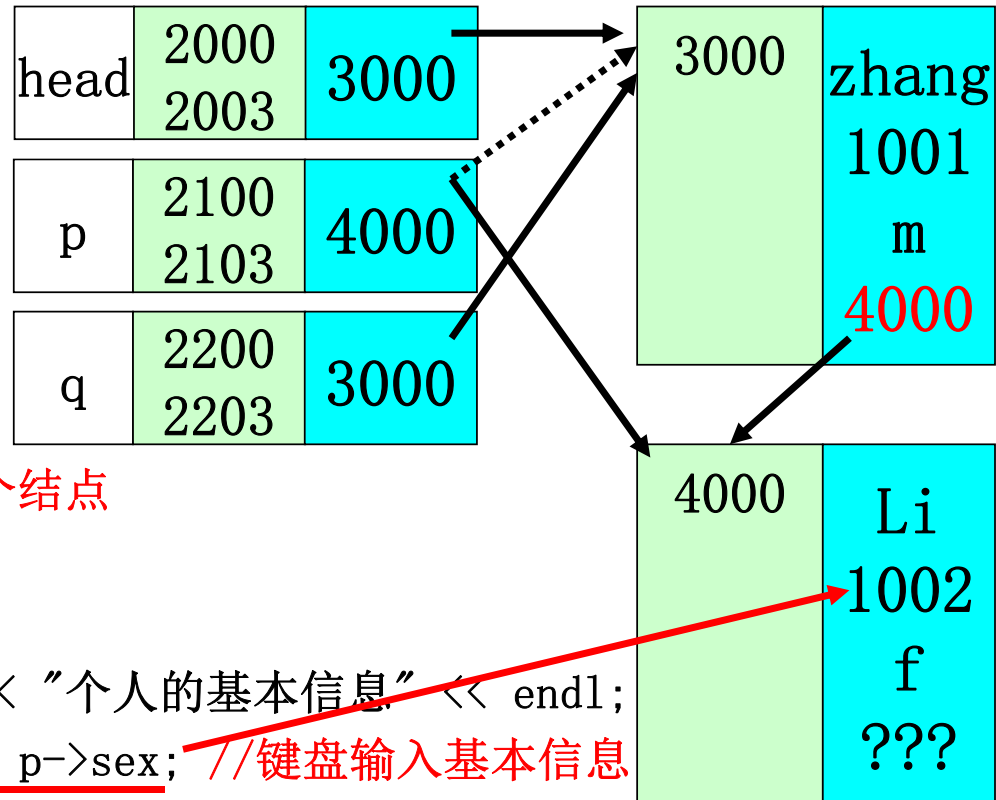
```
    cout << "请输入第" << i+1 << "个人的基本信息" << endl;
```

```
    cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
```

```
    p->next = NULL;
```

```
  }
```

i=1的循环





例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
  for(i=0; i<5; i++) {
```

```
    if (i>0)
```

```
      q=p;
```

```
    p = new(nothrow) student;
```

```
    if (p==NULL)
```

```
      return -1;
```

```
    if (i==0)
```

```
      head = p; //head指向第1个结点
```

```
    else
```

```
      q->next = p;
```

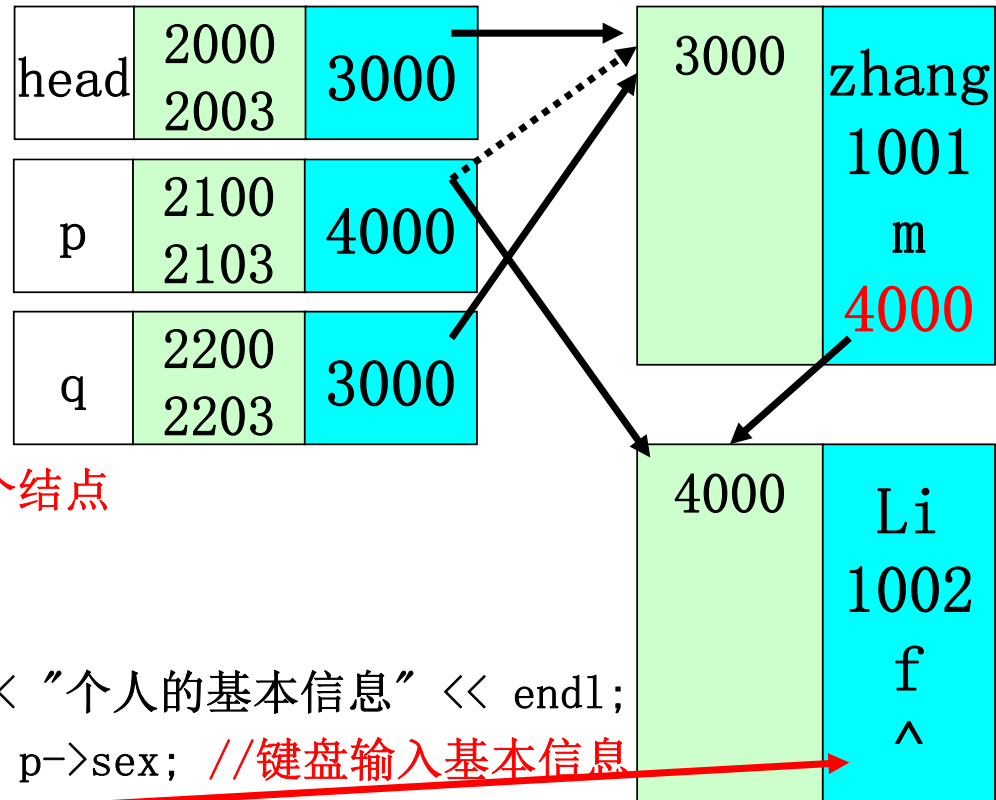
```
    cout << "请输入第" << i+1 << "个人的基本信息" << endl;
```

```
    cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
```

```
    p->next = NULL;
```

```
  }
```

i=1的循环





例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
  for(i=0; i<5; i++) {
```

```
    if (i>0)
```

```
        q=p;
```

```
    p = new(nothrow) student;
```

```
    if (p==NULL)
```

```
        return -1;
```

```
    if (i==0)
```

```
        head = p; //head指向第1个结点
```

```
    else
```

```
        q->next = p;
```

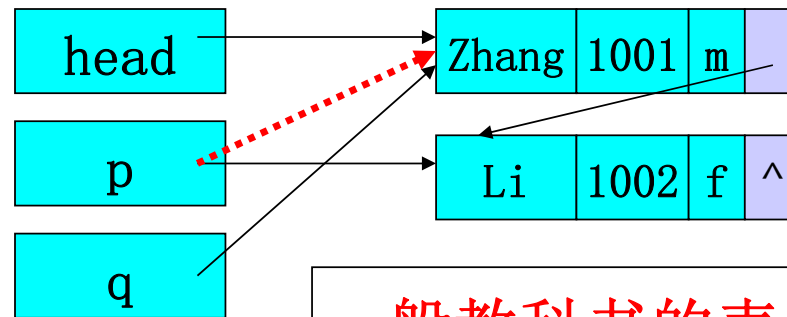
```
    cout << "请输入第" << i+1 << "个人的基本信息" << endl;
```

```
    cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
```

```
    p->next = NULL;
```

```
}
```

i=1的循环



一般教科书的表示



例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
  for(i=0; i<5; i++) {
```

```
    if (i>0)
```

```
        q=p;
```

```
    p = new(nothrow) student;
```

```
    if (p==NULL)
```

```
        return -1;
```

```
    if (i==0)
```

```
        head = p; //head指向第1个结点
```

```
    else
```

```
        q->next = p;
```

```
    cout << "请输入第" << i+1 << "个人的基本信息" << endl;
```

```
    cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
```

```
    p->next = NULL;
```

```
}
```

i=2-4自行画图



例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

i=4的循环结束

```
for(i=0; i<5; i++) {
```

```
    if (i>0)
```

```
        q=p;
```

```
    p = new(nothrow) student;
```

```
    if (p==NULL)
```

```
        return -1;
```

```
    if (i==0)
```

```
        head = p; //head指向第1个结点
```

```
    else
```

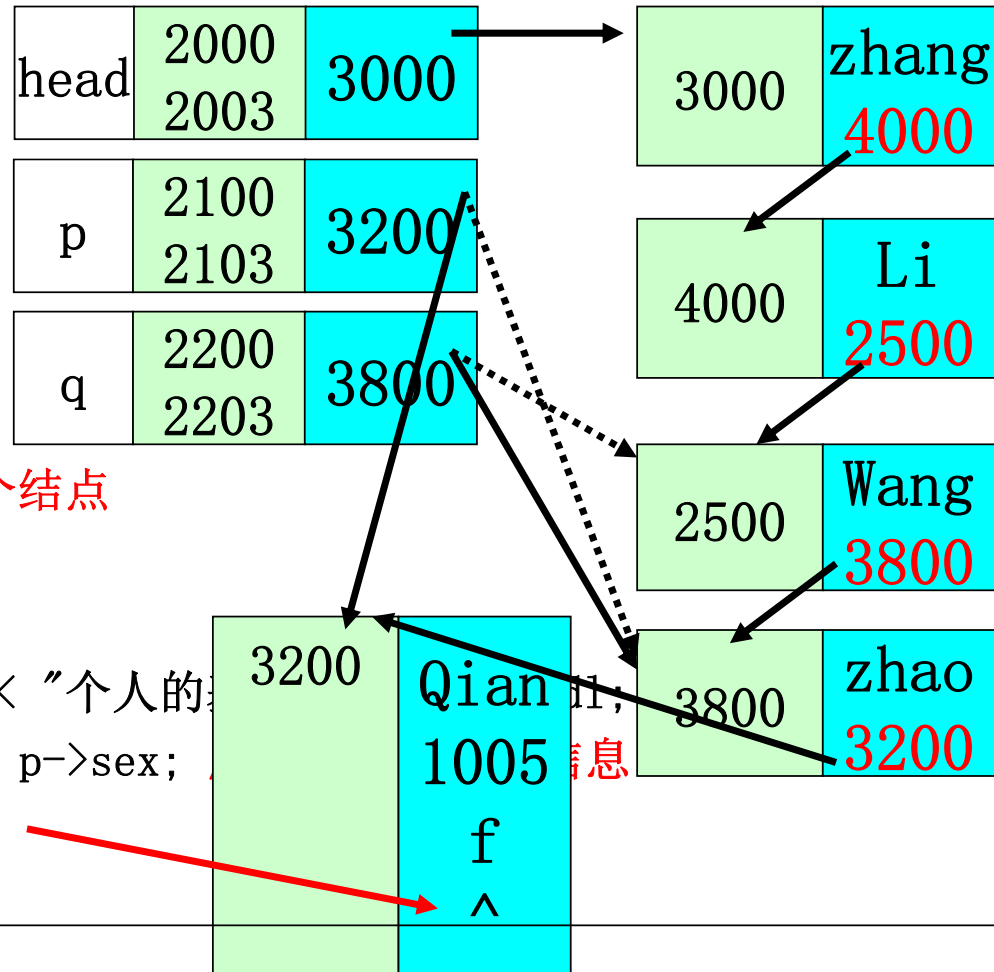
```
        q->next = p;
```

```
    cout << "请输入第" << i+1 << "个人的";
```

```
    cin >> p->name >> p->num >> p->sex;
```

```
    p->next = NULL;
```

```
}
```





例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
  for(i=0; i<5; i++) {
```

```
    if (i>0)
```

```
        q=p;
```

```
    p = new(nothrow) student;
```

```
    if (p==NULL)
```

```
        return -1;
```

```
    if (i==0)
```

```
        head = p; //head指向第1个结点
```

```
    else
```

```
        q->next = p;
```

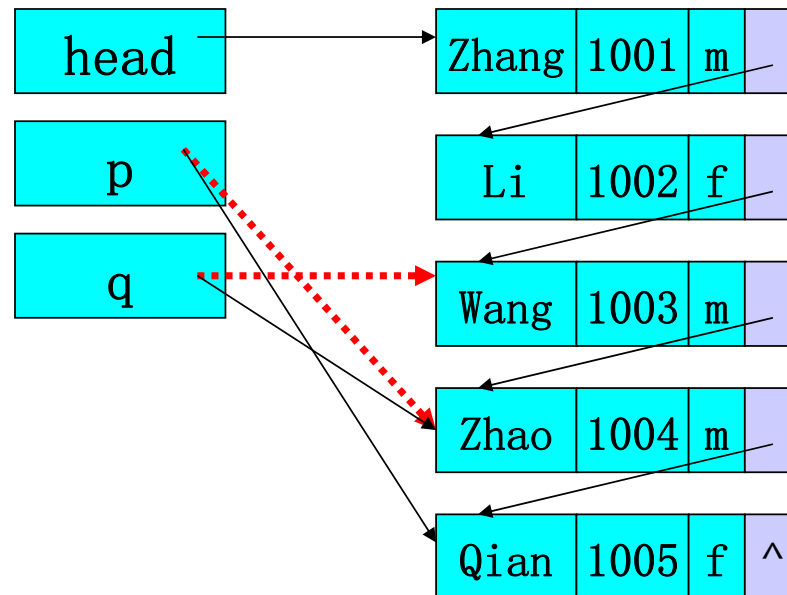
```
    cout << "请输入第" << i+1 << "个人的基本信息" << endl;
```

```
    cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
```

```
    p->next = NULL;
```

```
}
```

i=4的循环结束



一般教科书的表示



例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
  for(i=0; i<5; i++) {
```

```
    if (i>0)
```

```
        q=p;
```

```
    p = new(nothrow) student;
```

```
    if (p==NULL)
```

```
        return -1;
```

```
    if (i==0)
```

```
        head = p; //head指向第1个结点
```

```
    else
```

```
        q->next = p;
```

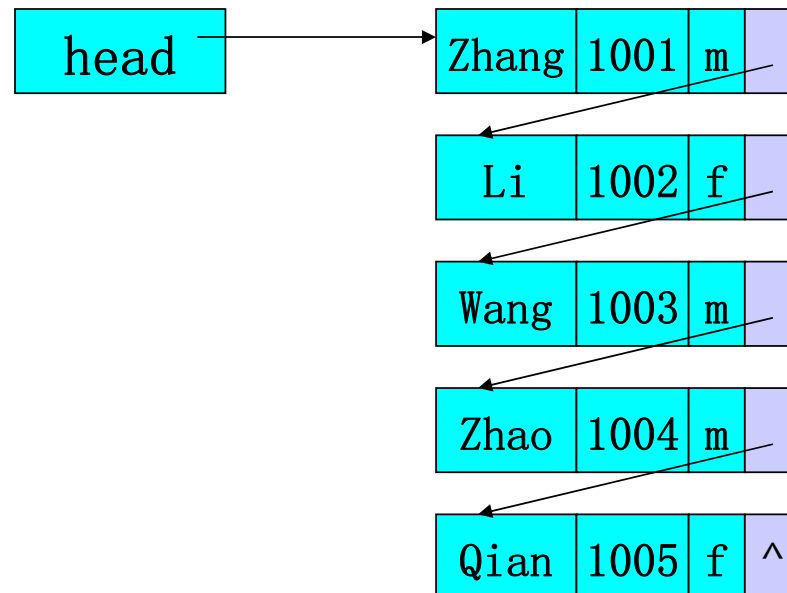
```
    cout << "请输入第" << i+1 << "个人的基本信息" << endl;
```

```
    cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
```

```
    p->next = NULL;
```

```
}
```

循环完成





例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
    for(i=0; i<5; i++) {  
        }
```

刚才建立链表的循环

```
    p=head; //p复位，指向第1个结点
```

```
    while(p!=NULL) { //循环进行输出
```

```
        cout << p->name << " " << p->num << " " << p->sex << endl;
```

```
        p=p->next;
```

```
    }
```

```
    p=head; //p复位，指向第1个结点
```

```
    while(p) { //循环进行各结点释放
```

```
        q = p->next;
```

```
        delete p;
```

```
        p = q;
```

```
    }
```

注意：不能用free

```
    return 0;
```

```
}
```

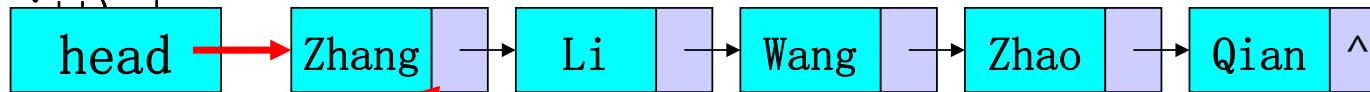


例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

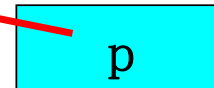
```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
for(i=0; i<5; i++) {
```



```
    p=head; //p复位，指向第1个结点
```

```
    while(p!=NULL) { //循环进行输出
```



```
        cout << p->name << " " << p->num << " " << p->sex << endl;
```

```
        p=p->next;
```

```
    }
```

```
    p=head; //p复位，指向第1个结点
```

```
    while(p) { //循环进行各结点释放
```

```
        q = p->next;
```

```
        delete p;
```

```
        p = q;
```

```
    }
```

```
    return 0;
```

```
}
```

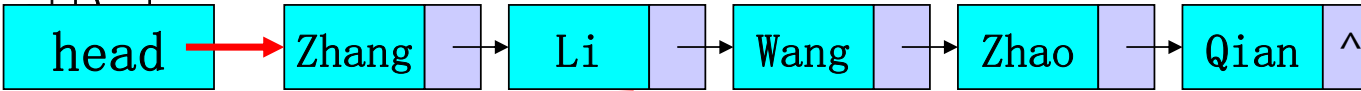



例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
for(i=0; i<5; i++) {
```



```
  }
```

```
p=head; //p复位，指向第1个结点
```

```
while(p!=NULL) { //循环进行输出
```

```
    cout << p->name << " " << p->num << " " << p->sex << endl;
```

```
    p=p->next;
```

```
}
```

Zhang 1001 m

```
p=head; //p复位，指向第1个结点
```

```
while(p) { //循环进行各结点释放
```

```
    q = p->next;
```

```
    delete p;
```

```
    p = q;
```

```
}
```

```
return 0;
```

```
}
```

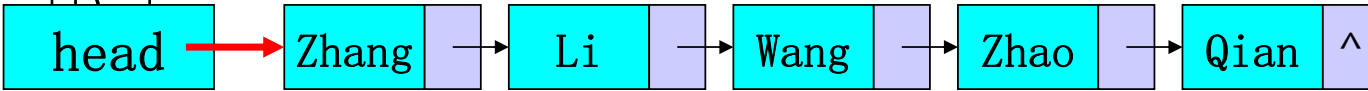


例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
for(i=0; i<5; i++) {
```



```
}
```

```
p=head; //p复位，指向第1个结点
```

```
while(p!=NULL) { //循环进行输出
```

```
    cout << p->name << " " << p->num << " " << p->sex << endl;
```

```
    p=p->next;
```

```
}
```

```
p=head; //p复位，指向第1个结点
```

```
while(p) { //循环进行各结点释放
```

```
    q = p->next;
```

```
    delete p;
```

```
    p = q;
```

```
}
```

```
return 0;
```

```
}
```

p

后续输出自行画图理解

Zhang 1001 m

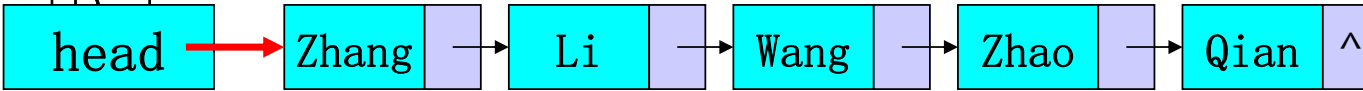


例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
for(i=0; i<5; i++) {
```



```
}
```

```
p=head; //p复位，指向第1个结点
```

```
while(p!=NULL) { //循环进行输出
```

```
    cout << p->name << " " << p->num << " " << p->sex << endl;
```

```
    p=p->next;
```

```
}
```

p: ^

最后一个结点输出

循环结束

Zhang 1001 m

Li 1002 f

Wang 1003 m

Zhao 1004 m

Qian 1005 f

```
p=head; //p复位，指向第1个结点
```

```
while(p) { //循环进行各结点释放
```

```
    q = p->next;
```

```
    delete p;
```

```
    p = q;
```

```
}
```

```
return 0;
```

```
}
```



例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
  for(i=0; i<5; i++) {  
    }
```

```
  p=head; //p复位，指向第1个结点
```

```
  while(p!=NULL) { //循环进行输出
```

```
    cout << p->name << " " << p->num << " " << p->sex << endl;
```

```
    p=p->next;
```

```
  }
```

```
graph LR; head[head] --> Zhang[Zhang]; Zhang --> Li[Li]; Li --> Wang[Wang]; Wang --> Zhao[Zhao]; Zhao --> Qian[Qian]; Qian --> null(^);
```

```
  p=head; //p复位，指向第1个结点
```

```
  while(p) { //循环进行各结点释放
```

```
    q = p->next;
```

```
    delete p;
```

```
    p = q;
```

```
  }
```

```
graph LR; p[p] --> Zhang[Zhang];
```

```
  return 0;
```

```
}
```



例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
  for(i=0; i<5; i++) {  
    }
```

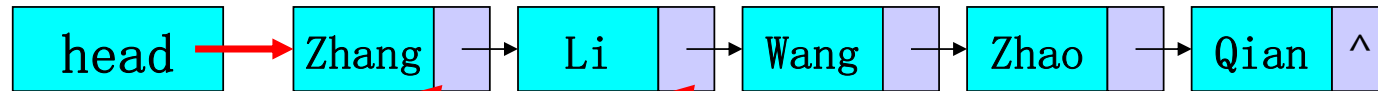
```
  p=head; //p复位，指向第1个结点
```

```
  while(p!=NULL) { //循环进行输出
```

```
    cout << p->name << " " << p->num << " " << p->sex << endl;
```

```
    p=p->next;
```

```
  }
```



```
  p=head; //p复位，指向第1个结点
```

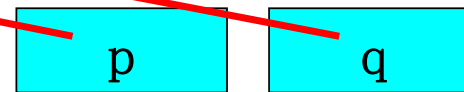
```
  while(p) { //循环进行各结点释放
```

```
    q = p->next;
```

```
    delete p;
```

```
    p = q;
```

```
  }
```



```
  return 0;
```

```
}
```



例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
  for(i=0; i<5; i++) {  
    }
```

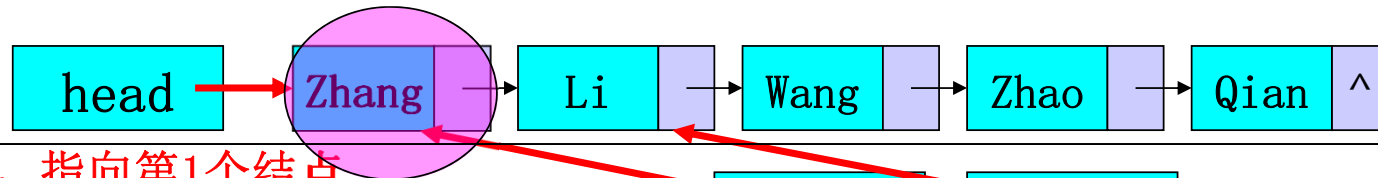
```
  p=head; //p复位，指向第1个结点
```

```
  while(p!=NULL) { //循环进行输出
```

```
    cout << p->name << " " << p->num << " " << p->sex << endl;
```

```
    p=p->next;
```

```
  }
```



```
  p=head; //p复位，指向第1个结点
```

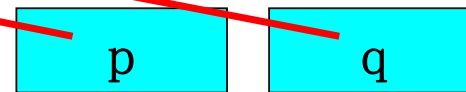
```
  while(p) { //循环进行各结点释放
```

```
    q = p->next;
```

```
    delete p;
```

```
    p = q;
```

```
  }
```



```
  return 0;
```

```
}
```



例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
  for(i=0; i<5; i++) {  
    }
```

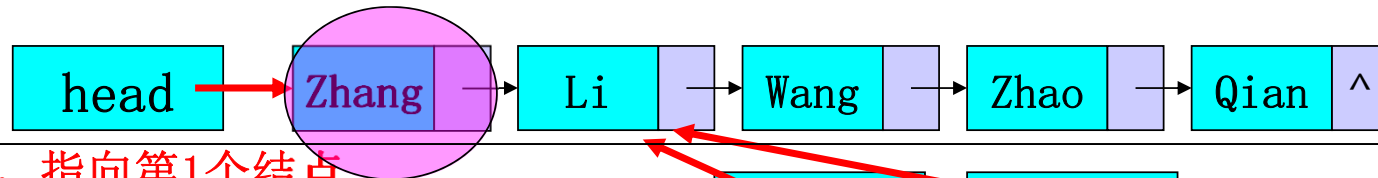
```
  p=head; //p复位，指向第1个结点
```

```
  while(p!=NULL) { //循环进行输出
```

```
    cout << p->name << " " << p->num << " " << p->sex << endl;
```

```
    p=p->next;
```

```
  }
```



```
  p=head; //p复位，指向第1个结点
```

```
  while(p) { //循环进行各结点释放
```

```
    q = p->next;
```

```
    delete p;
```

```
    p = q;
```

```
  }
```

```
  return 0;
```

```
}
```

后续结点释放自行画图理解



例：用动态内存申请方式建立一个有5个结点的链表

```
int main()
```

```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

```
  for(i=0; i<5; i++) {  
    }
```

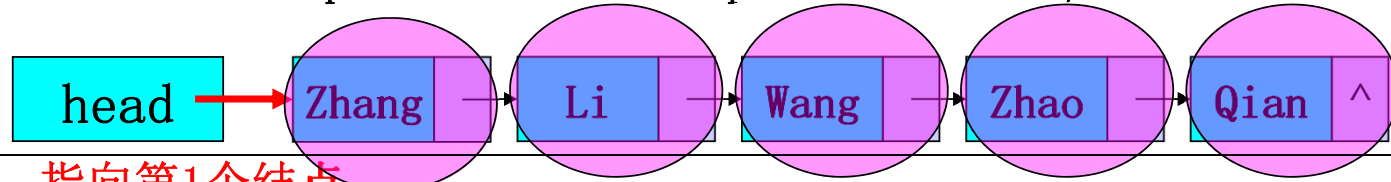
```
  p=head; //p复位，指向第1个结点
```

```
  while(p!=NULL) { //循环进行输出
```

```
    cout << p->name << " " << p->num << " " << p->sex << endl;
```

```
    p=p->next;
```

```
  }
```



```
  p=head; //p复位，指向第1个结点
```

```
  while(p) { //循环进行各结点释放
```

```
    q = p->next;
```

```
    delete p;
```

```
    p = q;
```

```
  }
```

p: ^ q: ^

最后一个结点被释放后

循环结束，new申请的5个空间已被释放，指针变量head/p/q自身不是动态申请空间，由操作系统回收

```
  return 0;
```

```
}
```




§ 13. 动态内存申请

4. 含动态内存申请内存的类和对象

4.1. 对象的动态建立和释放

C语言方法: `Time *p;`
申请: `p = (Time *)malloc(sizeof(Time));`
`p = (Time *)malloc(10*sizeof(Time));`
`if (p==NULL) { ... }`
释放: `free(p);` //统一方法释放 单Time/Time数组

★ 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(见4.2例)
- 前例中, struct中有string类对象, 则 malloc/free 会出错

C++方法: `Time *p;`
申请: `p = new(nothrow) Time;`
`p = new(nothrow) Time[10];`
`if (p==NULL) { ... }`
释放: `delete p;` //释放单Time
`delete []p;` //释放Time数组

★ C++中delete时, 只要是数组, 必须加[]

- VS系列编译器会运行出错
- GNU系列(DevC++/Linux)虽然表面不出错, 但若含有动态内存申请, 则因为不调用析构函数, 仍会导致内存丢失



§ 13. 动态内存申请

4. 含动态内存申请内存的类和对象

4.1. 对象的动态建立和释放

4.2. 动态申请对象的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;
}
```

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

Microsoft Visual Studio 调试控制台

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

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

★ C++下采用C语言方式的动态内存申请，
不调用构造及析构函数

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

Microsoft Visual Studio 调试控制台

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



§ 13. 动态内存申请

4. 含动态内存申请内存的类和对象

4.1. 对象的动态建立和释放

4.2. 动态申请对象的构造函数与析构函数的调用时机

★ new时调用构造函数，delete时调用析构函数

★ C++下采用C语言方式的动态内存申请，不调用构造及析构函数(淘汰!)

§ 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~~ : 荣誉课内容(略)

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~~ : 荣誉课内容(略)

main开始前

main结束后

//例: 变化, C方式的动态申请, 内嵌string类(C++特有)

```
#include <iostream>
#include <string> //C++特有的string类需要
#include <stdlib.h>
using namespace std;
```

```
struct student {
    string name; //C++特有的string类
    int num;
    char sex;
};
```

```
int main()
{
    student *p;
    p = (student *)malloc(sizeof(student));
    if (p==NULL) //加判断保证程序的正确性
        return -1;
    p->name = "Wang Fun";
    p->num = 10123;
    p->sex = 'm';
    cout << p->name << endl
         << p->num << endl
         << p->sex << endl;
    free(p);

    return 0;
}
```

更进一步的解释:

★ malloc不激活student的构造函数(缺省的无参空体, 自然也不会激活成员string name的构造函数, 导致p->name访问时的内存错误)

前例:

多编译器运行, 哪些编译器下运行错误? 表现是什么? 为什么?

建议:

C++下的动态内存申请不建议采用C函数方式, 不要为了realloc的便捷性而降级



§ 13. 动态内存申请

4. 含动态内存申请内存的类和对象

4.1. 对象的动态建立和释放

4.2. 动态申请对象的构造函数与析构函数的调用时机

4.3. 在构造和析构函数中进行动态内存的申请与释放

§ 9. 类和对象基础

9.3. 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
- ★ 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)

```
class Time {  
    ...  
    public:  
        Time();  
        ~Time(); //声明  
};  
Time::Time()  
{ ...  
}  
Time::~~Time() //体外实现  
{ cout << "Time End" << endl;  
}
```



§ 13. 动态内存申请

4. 含动态内存申请内存的类和对象

4.3. 在构造和析构函数中进行动态内存的申请与释放

★ 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

★ 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!!!)

```
#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; //不需要显式调用构造，自动申请
    ...
} //不需要显式调用析构，自动释放
```

```
#include <iostream>
using namespace std;
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; //不需要显式调用构造，自动申请
    ...
    t1.Release();
} //必须显式调用Release()
```



§ 13. 动态内存申请

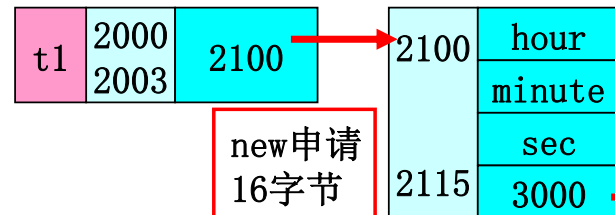
4. 含动态内存申请内存的类和对象

4.3. 在构造和析构函数中进行动态内存的申请与释放

★ 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

★ 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!!!)

★ 不要与对象的动态申请混淆

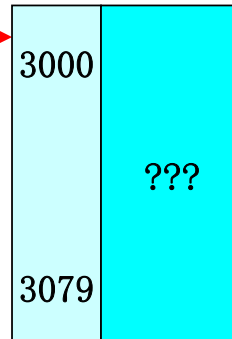


问: t1占的4个字节, 谁负责释放?

答: 操作系统

=>基本规则: 谁负责分配, 谁负责释放
(适合大程序的分工与组织)

delete释放
16字节



构造函数
申请80字节

析构函数
释放80字节

隐含但很明确的规则:
new Time和构造函数
中的new, 哪个在前?

隐含但很明确的规则:
delete t1和析构函数
中的delete, 哪个在前?

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

```
#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
}
```



§ 13. 动态内存申请

4. 含动态内存申请内存的类和对象

4.3. 在构造和析构函数中进行动态内存的申请与释放

★ 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

★ 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!!!)

★ 不要与对象的动态申请混淆

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

```
struct student {
    int num;
    char *name;
};
```

```
int main()
{
    student *s1;
    s1 = new(nothrow) student; //申请8字节
    s1->name = new(nothrow) char[6]; //申请6字节

    s1->num = 1001;
    strcpy(s1->name, "zhang");
    cout << s1->num << ":" << s1->name << endl;

    delete []s1->name; //释放6字节
    delete s1; //释放8字节

    return 0;
} //为节约篇幅，未判断申请是否成功
```

★ 如果出现需要嵌套进行动态内存申请的情况，
则按定义顺序进行申请，反序进行释放

前例改写，可有效
简化main的复杂度

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

```
struct student {
    int num;
    char *name;
    student()
    {
        name = new char[6]; //申请6字节
    }
    ~student()
    {
        delete []name; //释放6字节
    }
};

int main() //当main中多次对student申请/释放时
{
    student *s1;
    s1 = new(nothrow) student; //申请8字节
    s1->num = 1001;
    strcpy(s1->name, "zhang");
    cout << s1->num << ":" << s1->name << endl;
    delete s1; //释放8字节
    return 0;
}
```



§ 13. 动态内存申请

- 4. 含动态内存申请内存的类和对象
 - 4. 1. 对象的动态建立和释放
 - 4. 2. 动态申请对象的构造函数与析构函数的调用时机
 - 4. 3. 在构造和析构函数中进行动态内存的申请与释放
 - 4. 4. 含动态内存申请的对象赋值与复制

特别提醒:

整个4. 4节，不要仅仅模糊地记忆答案是正确/错误的，
要非常清晰地搞清楚每一个输出的由来，内在的原因是什么!!!



§ 13. 动态内存申请

- 4. 含动态内存申请内存的类和对象
- 4. 4. 含动态内存申请的对象的赋值与复制
- 4. 4. 1. 含动态内存申请的对象的赋值

§ 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**已存在**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

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

- ★ 两个对象属于同一个类，通过赋值语句实现(不能是定义时赋初值)
- ★ 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)

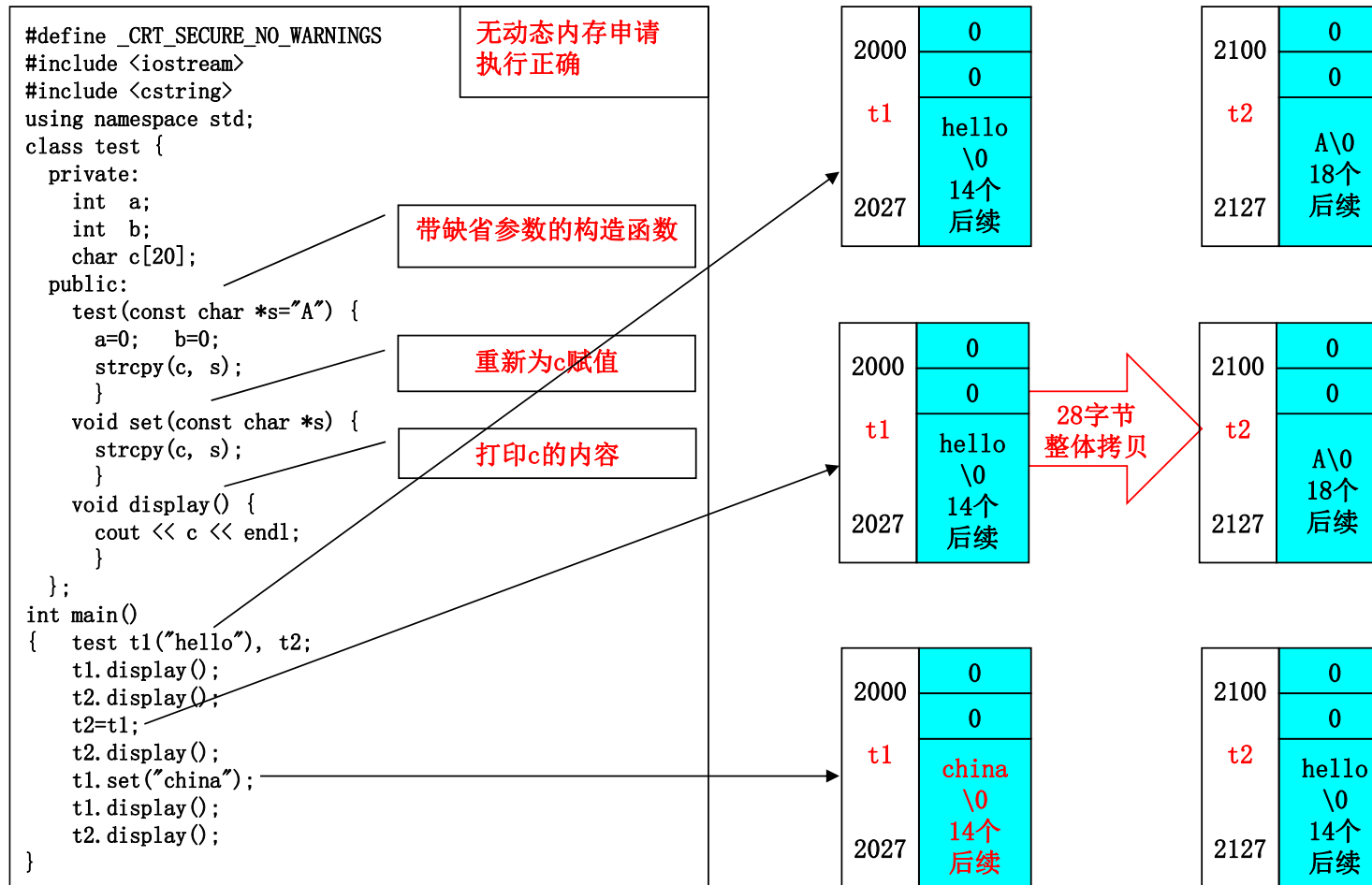


§ 13. 动态内存申请

4. 4. 含动态内存申请的对象的赋值与复制

4. 4. 1. 含动态内存申请的对象的赋值

★ 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





§ 13. 动态内存申请

4. 4. 含动态内存申请的对象的赋值与复制

4. 4. 1. 含动态内存申请的对象的赋值

★ 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

用多编译器分别
运行下面两个例子

```
#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(与期望不同)

注意:
1、篇幅问题, 假设申请成功
2、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(与期望不同)

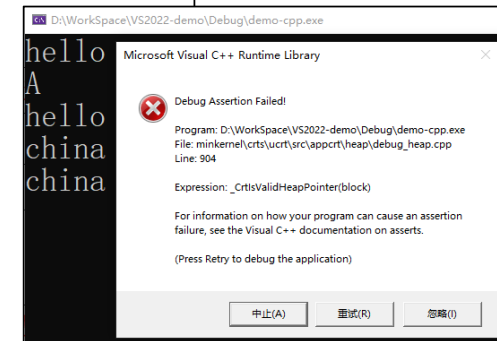
同左例, 加入析构函数后,
不但执行结果错, 而且
VS下会有错误弹窗,
GNU下虽然没有错误弹窗,
但仍然是错误的

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!!!

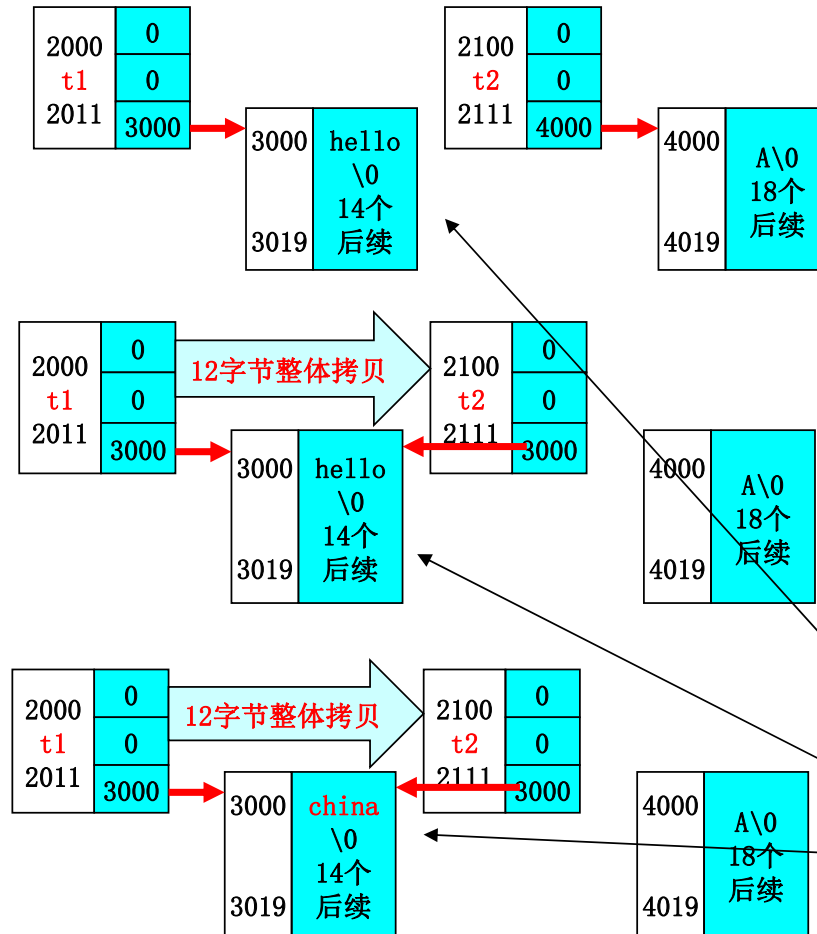
如何保证有动态内存时的赋值正确性?

后续模块 重载=运算符

4. 4. 含动态内存

4. 4. 1. 含动态内存

★ 若对象数据用



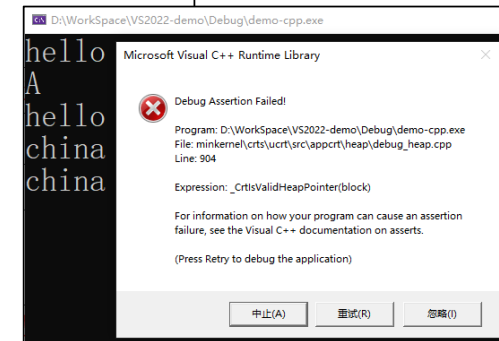
```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
class test {
private:
    int a;
    int b;
    char *c;
public:
    test(const char *s="A") {
        a=0; b=0;
        c = new char[20];
        strcpy(c, s);
    }
    ~test() {
        delete []c;
   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

同左例, 加入析构函数后,
不但执行结果错, 而且
VS下会有错误弹窗,
GNU下虽然没有错误弹窗,
但仍然是错误的

hello
A
hello
china
china





§ 13. 动态内存申请

- 4. 含动态内存申请内存的类和对象
- 4. 4. 含动态内存申请的对象的赋值与复制
- 4. 4. 1. 含动态内存申请的对象的赋值
- 4. 4. 2. 含动态内存申请的对象的复制

§ 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

形式：

类 对象名(已有对象名)	两种形式
类 对象名=已有对象名	本质一样

Time t1(14, 15, 23), t2(t1), t3=t1; //定义语句中

★ 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(已有对象t1 => 新对象t3)

t2 = t1; //赋值(已有对象t1 => 已有对象t2)

★ 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)



§ 13. 动态内存申请

4. 4. 含动态内存申请的对象赋值与复制

4. 4. 2. 含动态内存申请的对象复制

含义：建立一个**新**对象，其值与某个已有对象完全相同

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，则系统缺省定义的消失
- ★ 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



§ 13. 动态内存申请

4. 4. 含动态内存申请的对象的赋值与复制

4. 4. 2. 含动态内存申请的对象的复制

```
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 t0, t1(14, 15, 23), t2(t1), t3=t1;
    t0.display();    15:0:0
    t1.display();    14:15:23
    t2.display();    14:15:23
    t3.display();    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: $\text{Time } t0(15)$, 但类型不匹配!!!

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;
}
```



§ 13. 动态内存申请

4. 4. 含动态内存申请的对象赋值与复制

4. 4. 2. 含动态内存申请的对象复制

复制构造函数的调用时机:

- ★ (调用1) 用已有对象初始化一个新建立的对象时
- ★ (调用2) 函数形参为对象, 实参向形参进行单向传值时
- ★ (调用3) 函数的返回类型是对象时

★ (不调用1) 不包括形参为引用的情况 (引用为实参别名)

★ (不调用2) 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 (后续模块)

★ 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



§ 13. 动态内存申请

4. 4. 含动态内存申请的对象的赋值与复制

4. 4. 2. 含动态内存申请的对象复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

```
#include <iostream>
using namespace std;
class Time {
private:
    int hour, minute, sec;
public:
    Time(int h=0, int m=0, int s=0);
    Time(const Time &t);
    ~Time();
    void display() { cout << hour << ":" << minute << ":" << sec << endl; }
};

Time::Time(int h, int m, int s)
{
    hour = h;
    minute = m;
    sec = s;
    cout << "普通构造" << hour << endl;
}

Time::~Time()
{
    cout << "析构" << hour << endl;
}

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

★ (不调用1) 不包括形参为引用的情况 (引用为实参别名, 不调用)

- 用VS编译运行
 - 用Dev/Linux编译运行
- 多编译器一致

```
void fun(Time &t)
```

本例证明不调用1

```
{
    t.display();
}

int main()
{
    Time t1(14, 15, 23);
    fun(t1);
    return 0;
}
```

普通构造14
14:15:23
析构14



§ 13. 动态内存申请

4. 4. 含动态内存申请的对象的赋值与复制

4. 4. 2. 含动态内存申请的对象的复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

```
#include <iostream>
using namespace std;
class Time {
private:
    int hour, minute, sec;
public:
    Time(int h=0, int m=0, int s=0);
    Time(const Time &t);
    ~Time();
    void display() { cout << hour << ":" << minute << ":" << sec << endl; }
};
Time::Time(int h, int m, int s)
{
    hour = h;
    minute = m;
    sec = s;
    cout << "普通构造" << hour << endl;
}
Time::~Time()
{
    cout << "析构" << hour << endl;
}
Time::Time(const Time &t)
{
    hour = t.hour - 1;
    minute = t.minute - 1;
    sec = t.sec - 1;
    cout << "复制构造" << hour << endl;
}
```

★ (不调用1) 不包括形参为引用的情况 (引用为实参别名, 不调用)
★ (不调用2) 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 (后续内容)

- 用VS编译运行
 - 用Dev/Linux编译运行
- 多编译器一致

```
int main()
{
    Time t1(14, 15, 23), t2;
    t2 = t1;
    return 0;
}
```

本例证明不调用2

普通构造14
普通构造0
析构14
析构14



§ 13. 动态内存申请

4. 4. 含动态内存申请的对象的赋值与复制

4. 4. 2. 含动态内存申请的对象的复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

```
#include <iostream>
using namespace std;
class Time {
private:
    int hour, minute, sec;
public:
    Time(int h=0, int m=0, int s=0);
    Time(const Time &t);
    ~Time();
    void display() { cout << hour << ":" << minute << ":" << sec << endl; }
};

Time::Time(int h, int m, int s)
{
    hour = h;
    minute = m;
    sec = s;
    cout << "普通构造" << hour << endl;
}

Time::~Time()
{
    cout << "析构" << hour << endl;
}

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

★ (调用1)用已有对象初始化一个新建立的对象时

- 用VS编译运行
 - 用Dev/Linux编译运行
- 多编译器一致

```
int main()
{
    Time t1(14, 15, 23), t2(t1);
    t2.display();
    return 0;
}
```

本例证明调用1

普通构造14
复制构造13
13:14:22
析构13
析构14



§ 13. 动态内存申请

4. 4. 含动态内存申请的对象的赋值与复制

4. 4. 2. 含动态内存申请的对象的复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

```
#include <iostream>
using namespace std;
class Time {
private:
    int hour, minute, sec;
public:
    Time(int h=0, int m=0, int s=0);
    Time(const Time &t);
    ~Time();
    void display() { cout << hour << ":" << minute << ":" << sec << endl; }
};
Time::Time(int h, int m, int s)
{
    hour = h;
    minute = m;
    sec = s;
    cout << "普通构造" << hour << endl;
}
Time::~Time()
{
    cout << "析构" << hour << endl;
}
Time::Time(const Time &t)
{
    hour = t.hour - 1;
    minute = t.minute - 1;
    sec = t.sec - 1;
    cout << "复制构造" << hour << endl;
}
```

- ★ (调用1) 用已有对象初始化一个新建立的对象时
 - ★ (调用2) 函数形参为对象，实参向形参进行单向传值时
 - 用VS编译运行
 - 用Dev/Linux编译运行
- 多编译器一致

```
void fun(Time t)
{
    t.display();
}
int main()
{
    Time t1(14, 15, 23);
    fun(t1);
    return 0;
}
```

本例证明调用2

普通构造14
复制构造13
13:14:22
析构13
析构14



§ 13. 动态内存申请

4. 4. 含动态内存申请的对象的赋值与复制

4. 4. 2. 含动态内存申请的对象的复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

```
#include <iostream>
using namespace std;
class Time {
private:
    int hour, minute, sec;
public:
    Time(int h=0, int m=0, int s=0);
    Time(const Time &t);
    ~Time();
    void display() { cout << hour << ":" << minute << ":" << sec << endl; }
};

Time::Time(int h, int m, int s)
{
    hour = h;
    minute = m;
    sec = s;
    cout << "普通构造" << hour << endl;
}

Time::~Time()
{
    cout << "析构" << hour << endl;
}

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

- ★ (调用1) 用已有对象初始化一个新建立的对象时
 - ★ (调用2) 函数形参为对象，实参向形参进行单向传值时
 - ★ (调用3) 函数的返回类型是对象时
 - 用VS编译运行
 - 用Dev/Linux编译运行
- 返回自动变量时，多编译器有差异
返回静态局部变量时，多编译器一致

Time fun() { Time t1(14, 15, 23); return t1; }		本例证明调用3
int main() { Time t2 = fun(); t2.display(); }	//VS 普通构造14 复制构造13 13:14:22 析构13 析构14	//Dev+Linux 普通构造14 14:15:23 析构14

Time fun() { static Time t1(14, 15, 23); return t1; }		本例证明调用3
int main() { Time t2 = fun(); t2.display(); }	普通构造14 复制构造13 13:14:22 析构13 析构14	



§ 13. 动态内存申请

4. 4. 含动态内存申请的对象的赋值与复制

4. 4. 2. 含动态内存申请的对象的复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

NRV技术讨论：

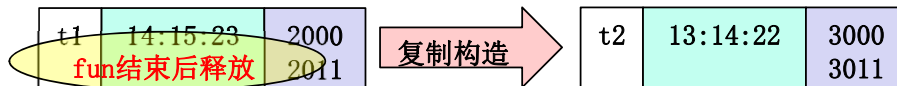
VS的解释：

return t1时，调用复制构造函数产生一份拷贝t2，再释放t1

GNU (Dev/Linux) 的解释：

采用NRV (NRV=Named Return value) 优化技术, 当函数返回类型为对象且被返回的是一个自动对象时，不调用复制构造函数而直接将原空间映射为新名称 (t2直接利用t1原空间)，从而提高运行速度 (少复制一次内存)

VS系列



Gnu系列

t1	14:15:23	2000
t2		2011

- ★ (调用1) 用已有对象初始化一个新建立的对象时
- ★ (调用2) 函数形参为对象，实参向形参进行单向传值时
- ★ (调用3) 函数的返回类型是对象时

- 用VS编译运行
- 用Dev/Linux编译运行

返回自动变量时，多编译器有差异
返回静态局部变量时，多编译器一致

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

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

本例证明调用3

//VS
普通构造14
复制构造13
析构13
析构14

//Dev+Linux
普通构造14
14:15:23
析构14



§ 13. 动态内存申请

4. 4. 含动态内存申请的对象的赋值与复制

4. 4. 2. 含动态内存申请的对象的复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

NRV技术讨论：

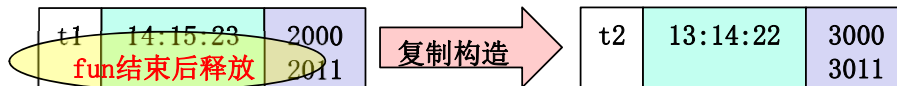
VS的解释：

return t1时，调用复制构造函数产生一份拷贝t2，再释放t1

GNU (Dev/Linux) 的解释：

采用NRV (NRV=Named Return value) 优化技术, 当函数返回类型为对象且被返回的是一个自动对象时，不调用复制构造函数而直接将原空间映射为新名称 (t2直接利用t1原空间)，从而提高运行速度 (少复制一次内存)

VS系列



Gnu系列

t1	14:15:23	2000
t2		2011

- ★ (调用1) 用已有对象初始化一个新建立的对象时
 - ★ (调用2) 函数形参为对象，实参向形参进行单向传值时
 - ★ (调用3) 函数的返回类型是对象时
 - 用VS编译运行
 - 用Dev/Linux编译运行
- 返回自动变量时，多编译器有差异
返回静态局部变量时，多编译器一致

Time fun()

```
{ Time t1(14, 15, 23);  
  cout << "&t1=" << &t1 << endl;  
  return t1;  
}
```

int main()

```
{ Time t2 = fun();  
  cout << "&t1=" << &t1 << endl;  
  t2.display();  
}
```

NRV技术讨论： (如何验证)

用不同编译器运行，观察t1和t2的地址是否相同，解释？

VS:

普通构造14

&t1=某地址

复制构造13

析构14

&t2=某地址 (与t1不同)

13:14:22

析构13

VS:

7行输出，观察两个地址

GNU:

5行输出，观察两个地址

GNU:

普通构造

&t1=某地址

&t2=某地址 (同t1)

14:15:23

析构14



§ 13. 动态内存申请

4. 4. 含动态内存申请的对象的赋值与复制

4. 4. 2. 含动态内存申请的对象的复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

NRV技术讨论：

VS的解释：

return t1时，调用复制构造函数产生一份拷贝t2，再释放t1

GNU (Dev/Linux) 的解释：

采用NRV (NRV=Named Return value) 优化技术, 当函数返回类型为对象且被返回的是一个自动对象时，不调用复制构造函数而直接将原空间映射为新名称 (t2直接利用t1原空间)，从而提高运行速度 (少复制一次内存)

★ “被返回值”不是自动对象时，不能进行NRV优化
(只有当“被返回值”在函数运行结束后被释放时才适用NRV优化)

- ★ (调用1) 用已有对象初始化一个新建立的对象时
- ★ (调用2) 函数形参为对象，实参向形参进行单向传值时
- ★ (调用3) 函数的返回类型是对象时
 - 用VS编译运行
 - 用Dev/Linux编译运行
 - 返回自动变量时，多编译器有差异
 - 返回静态局部变量时，多编译器一致

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

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

本例证明调用3

普通构造14
复制构造13
析构13
析构14



§ 13. 动态内存申请

4. 4. 含动态内存申请的对象的赋值与复制

4. 4. 2. 含动态内存申请的对象的复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

NRV技术讨论：

```
编译器选项
设定编译器配置
TDM-GCC 9.2.0 32-bit Debug
[root@vm-1234567 ~]# c++ -Wall -o t1 test.cpp
[root@vm-1234567 ~]# c++ -Wall -fno-elide-constructors -o t2 test.cpp
[root@vm-1234567 ~]# ./t1
普通构造14
14:15:23
析构14
[root@vm-1234567 ~]# ./t2
普通构造14
复制构造13
析构14
复制构造12
析构13
12:13:21
析构12
[root@vm-1234567 ~]#
```

★ 可能不满足某些特殊要求

(例：本测试样例中，希望复制构造函数将Time的三个成员hour/minute/sec各减1)

=> 合理推断：应有编译选项设置，可指定不采用NRV技术

-fno-elide-constructors

Dev：工具 - 编译选项 - 编译器卡片 - 加入命令

Linux：直接加入编译参数

- ★ (调用1)用已有对象初始化一个新建立的对象时
- ★ (调用2)函数形参为对象，实参向形参进行单向传值时
- ★ (调用3)函数的返回类型是对象时

- 用VS编译运行
- 用Dev/Linux编译运行

返回自动变量时，多编译器有差异
返回静态局部变量时，多编译器一致

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

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

本例证明调用3

//VS
普通构造14
复制构造13
13:14:22
析构13
析构14

//Dev+Linux
普通构造14
14:15:23
析构14

1. 本例程序，GNU下加编译选项是7行，不加是3行，为什么？
2. 为什么VS是5行？

//Dev+Linux(加选项)
普通构造14
复制构造13
析构14
复制构造12
析构13
12:13:21
析构12



§ 13. 动态内存申请

4. 4. 含动态内存申请的对象赋值与复制

4. 4. 2. 含动态内存申请的对象复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

NRV技术讨论：

NRV的进一步讨论：

在完全无NRV的情况下，①②两处应该调用两次复制构造函数

① t1 => 无名临时对象 13:14:22

② 无名临时对象 => t2 12:13:21

输出应为：

普通构造14

复制构造13 //①处复制构造，t1=>临时对象

析构14

复制构造12 //②处复制构造，临时对象=>t2

析构13 //临时对象析构

12:13:21

析构12

结论：

1、GNU加 `-fno-elide-constructors` 后输出符合预期

2、VS默认设置下，在①/②的某处仍然用了NRV（如何判断哪处？）

- ★（调用1）用已有对象初始化一个新建立的对象时
- ★（调用2）函数形参为对象，实参向形参进行单向传值时
- ★（调用3）函数的返回类型是对象时

● 用VS编译运行

● 用Dev/Linux编译运行

返回自动变量时，多编译器有差异

返回静态局部变量时，多编译器一致

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

本例证明调用3

//VS
普通构造14
复制构造13
13:14:22
析构13
析构14

//Dev+Linux
普通构造14
14:15:23
析构14

1. 本例程序，GNU下加编译选项是7行，不加是3行，为什么？
2. 为什么VS是5行？

//Dev+Linux(加选项)
普通构造14
复制构造13
析构14
复制构造12
析构13
12:13:21
析构12



§ 13. 动态内存申请

4. 4. 含动态内存申请的对象的赋值与复制

4. 4. 2. 含动态内存申请的对象的复制

本例仅用于证明复制构造函数的2种不调用时机和3种调用时机，无实际及具体含义

NRV技术讨论：

```
Time fun()
{   Time t1(14, 15, 23);
    return t1;
}           ①
int main()
{
    Time t2;
    ②
    t2 = fun(); //前例(不调用1)已证明②处不调用复制构造函数
    t2.display();
}
```

//VS
普通构造0
普通构造14
复制构造13
析构14
析构13
13:14:22
析构13

- ★ (调用1)用已有对象初始化一个新建立的对象时
- ★ (调用2)函数形参为对象，实参向形参进行单向传值时
- ★ (调用3)函数的返回类型是对象时

- 用VS编译运行
- 用Dev/Linux编译运行

返回自动变量时，多编译器有差异
返回静态局部变量时，多编译器一致

```
Time fun()
{   Time t1(14, 15, 23);
    return t1;
}           ①
int main()
{   Time t2 = fun(); ②
    t2.display();
}
```

本例证明调用3

//VS
普通构造14
复制构造13
13:14:22
析构13
析构14

NRV的进一步讨论：

在完全无NRV的情况下，①②两处应该调用两次复制构造函数

① t1 => 无名临时对象 13:14:22

② 无名临时对象 => t2 12:13:21

结论：

1、GNU加 `-fno-elide-constructors` 后输出符合预期

2、VS默认设置下，在①/②的某处仍然用了NRV (如何判断哪处?)

综合分析两处输出，可得到结论：

VS在_①_处调用了复制构造函数(未用NRV)

而在_②_处使用了NRV技术



§ 13. 动态内存申请

4. 4. 含动态内存申请的对象的赋值与复制

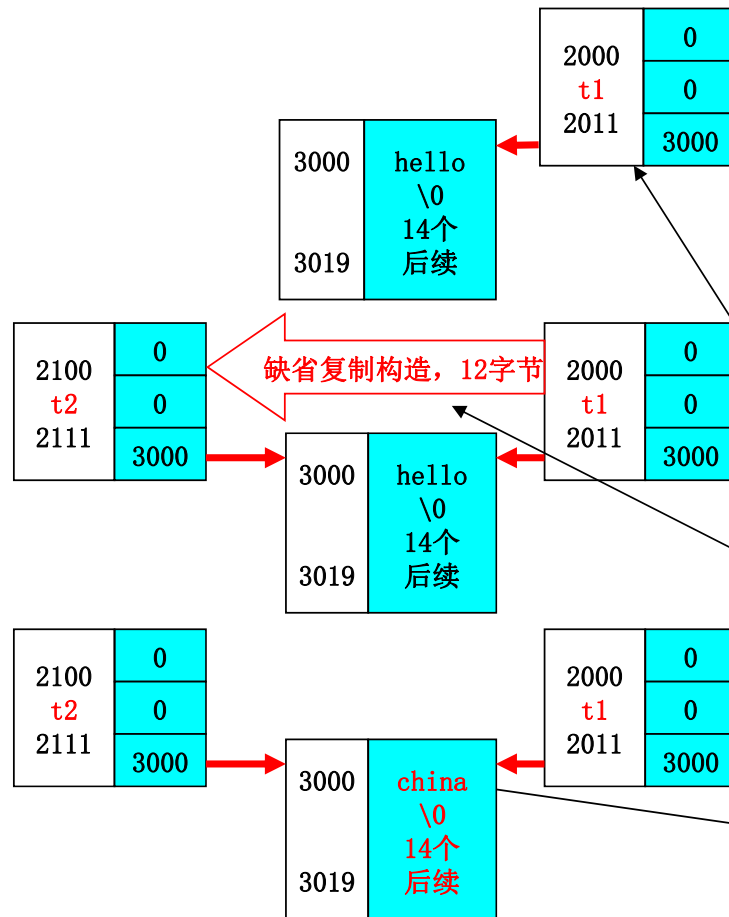
4. 4. 2. 含动态内存申请的对象的复制

★ 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

注意：
t2调用缺省的复制构造函数，
不再调用普通构造函数
(未申请20字节空间)，
因此本例中没有内存丢失

```
//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



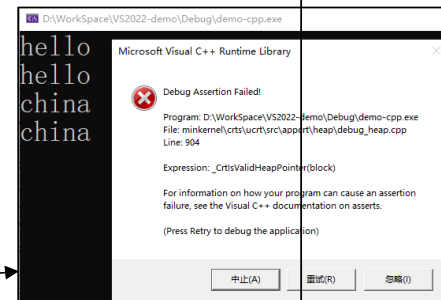
```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
```

```
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);
    } //带缺省参数的构造函数
    ~test() {
        delete []c;
    } //析构函数
    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();
}
```

复制错误示例

main完成后，
按t2, t1的顺序调用析构函数
导致3000-3019被重复释放



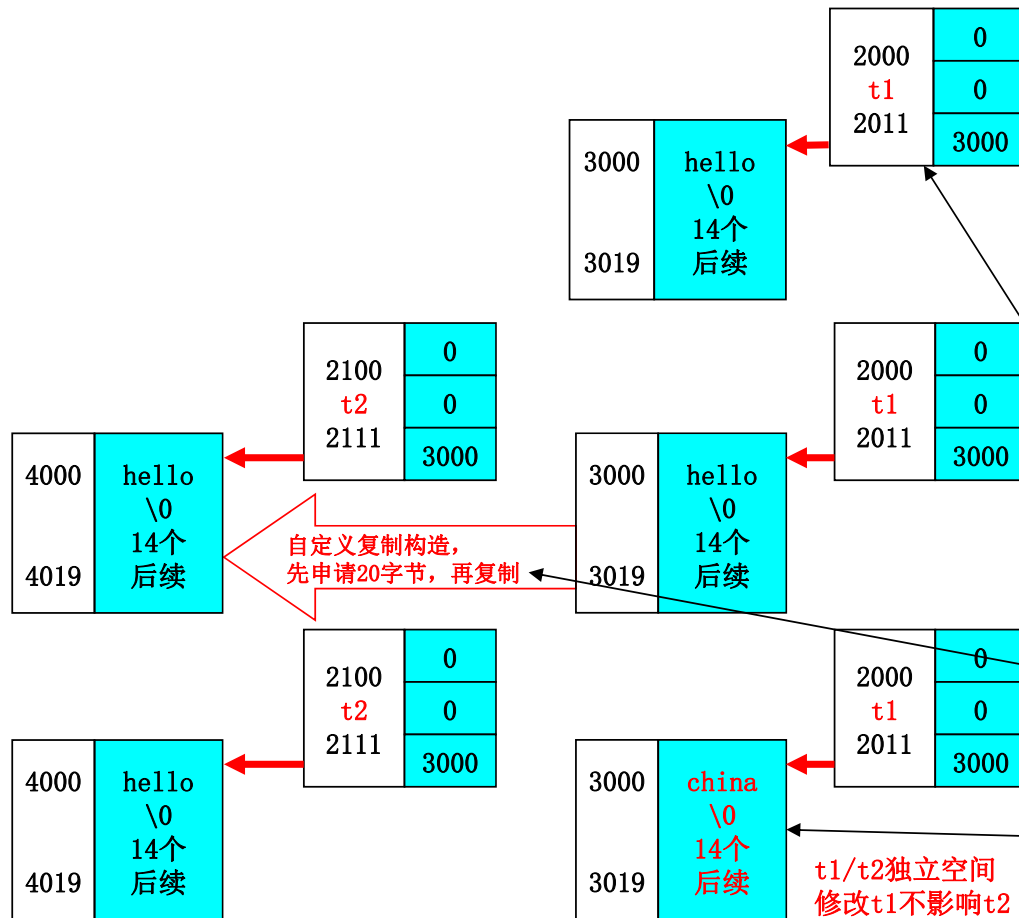


§ 13. 动态内存申请

4. 4. 含动态内存申请的对象的赋值与复制

4. 4. 2. 含动态内存申请的对象的复制

★ 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



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

复制正确示例

仔细体会两者
实现的区别

```
//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



§ 13. 动态内存申请

4. 4. 含动态内存申请的对象的赋值与复制

4. 4. 2. 含动态内存申请的对象的复制

★ 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

注意：虽然解决了定义时赋初值问题(`test t2(t1) / t2=t1;`)，但仍无法解决赋值问题(`t2 = t1;`)

赋值仍然会错，具体要用后续模块的=运算符重载来解决

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

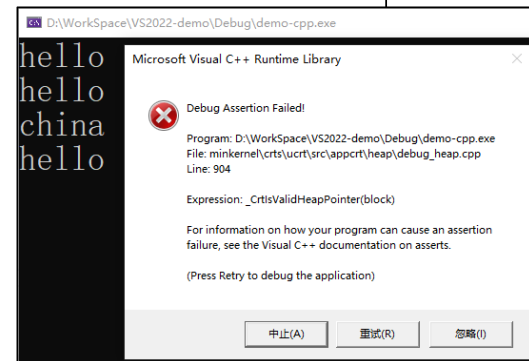
class test {
private:
    int a;
    int b;
    char *c;
public:
    test(const char *s="A") {
        a=0; b=0;
        c=new char[20];
        strcpy(c, s);
    } //带缺省参数的构造函数

    test(const test &t); //复制构造函数的声明
    ~test() { delete []c; } //析构函数
    void set(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()
{
    test t1("hello"), t2(t1);
    t1.display();      hello
    t2.display();      hello
    t1.set("china");
    t1.display();      china
    t2.display();      hello
    t2 = t1;
}
```

复制正确
仍无法解决
赋值错误示例





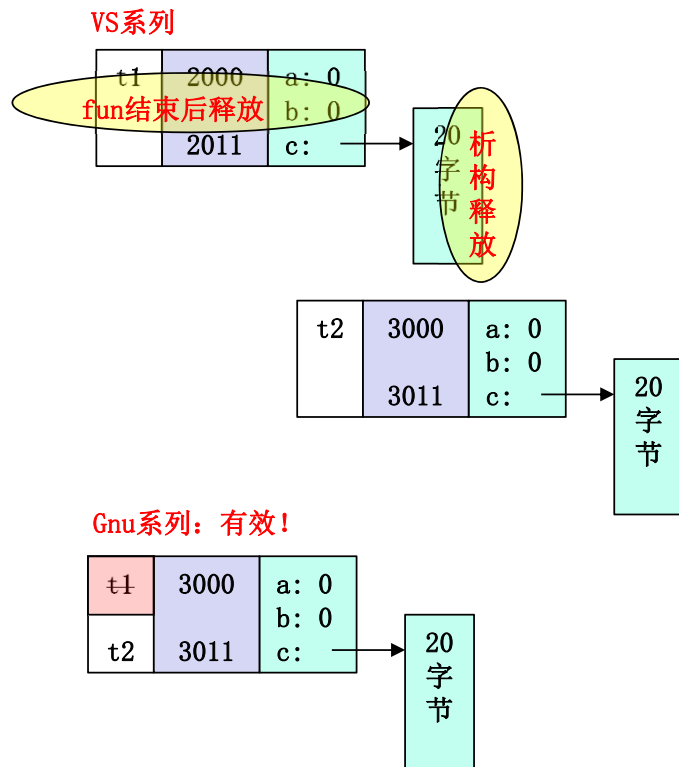
§ 13. 动态内存申请

4. 4. 含动态内存申请的对象的赋值与复制

4. 4. 2. 含动态内存申请的对象的复制

★ NRV技术的进一步讨论:

在GNU系列（Dev/Linux）中，如果含有二次申请，
且自定义了复制构造函数，NRV是否仍有效？



```
#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);
        cout << "普通构造" << (void *)c << endl;
    }
    test(const test &s);
    ~test() {
        cout << "析构" << (void *)c << endl;
        delete []c;
    }
    void set(const char *s) { strcpy(c, s); }
    void display() { cout << c << endl; }
};

test::test(const test &s)
{
    a=s.a;
    b=s.b;
    c=new char[20];
    strcpy(c, s.c);
    cout << "复制构造" << (void *)c << endl;
}

test fun()
{
    test t1("Hello");
    return t1;
}

int main()
{
    test t2 = fun();
    t2.display();
}
```

```
Microsoft Visual Studio
普通构造01304F08
复制构造01304F48
析构01304F08
Hello
析构01304F48

D:\Workspace\VS2022-
普通构造0xde0e48
Hello
析构0xde0e48
```

VS:
普通构造1
复制构造2
析构1
Hello
析构2

GNU:
普通构造1
Hello
析构1



§ 13. 动态内存申请

4. 含动态内存申请内存的类和对象

4.5. 浅拷贝与深拷贝

- ★ 浅拷贝 (Shallow Copy): 只复制对象的指针, 而不是复制对象自身;
新旧对象共享内存;
修改其中一个的值则另一个会随之改变;
两个对象是联动的
- ★ 深拷贝 (Deep Copy) : 复制对象自身;
新旧对象分别占用不同内存;
修改其中一个的值不会影响另一个;
两个对象是完全独立的

- 注: 1、网上浅拷贝/深拷贝的资料很多, 可读性/易懂程度/示例语言等各不相同, 但归根结底, 基本的原理就是本节的内存分析
- 2、某些无动态内存申请的语言, 其内部实现仍然是封装了动态内存申请, 只是不需要用户掌握而已
- 3、部分封装类, 可能不同函数分别浅/深拷贝
(例: 对象.assign() - 浅 / 对象.copy() - 深)
- 4、浅/深拷贝具体采用哪种, 根据实际需求决定