第三章 动态内存申请

模块3.2：C/C++方式的动态内存申请与释放

目录

- 动态内存的基本概念
- C中的相关函数
- C++中的相关运算符
- 内存的动态申请与释放
- 定位new运算符

3.2.1 动态内存的基本概念

- 编译器使用三块独立的内存
  - 静态变量
  - 自动变量
  - 动态存储
- 内存的分配
  - 静态存储区：编译时已分配好，内存在程序运行期间都存在（全局变量）
  - 栈：执行函数时，函数内局部变量的存储单元在栈上创建，函数执行结束时自动被释放，效率高，但内存容量有限（局部变量）
  - 堆：动态内存分配，可申请任意大小，用后自己负责释放。生存期自己决定，使用灵活，容易出错！

3.2.1 动态内存的基本概念

- 使用C函数malloc()等，或者C++运算符分配的内存
  - 不受作用域和链接性规则控制，因此可以在一个函数中分配动态内存，在另一个函数中将其释放
  - 动态内存不是LIFO，其分配和释放由new/delete或malloc()/free()完成

目录

- 动态内存的基本概念
- C中的相关函数
- C++中的相关运算符
- 内存的动态申请与释放
- 定位new运算符

3.2.2 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
  - 申请0字节内存，函数并不返回NULL，而是返回一个正常的内存地址。注意：无法使用这块大小为0的内存，而且此时无法用if（NULL！=p）校验
- 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

3.2.2 C中的相关函数

- 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
  - 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

3.2.2 C中的相关函数

- 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)

用malloc/calloc等申请的空间，用free释放

因为是系统库函数，需要包含头文件

```
#include <stdlib.h> //C方式
#include <cstdlib> //C++方式
```

目录

- 动态内存的基本概念
- C中的相关函数
- C++中的相关运算符
- 内存的动态申请与释放
- 定位new运算符

3.2.3 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，后面再介绍

- 用 delete 运算符释放空间

  用new申请的空间，用delete释放

  因为是运算符，不需要包含头文件

3.2.3 C++中的相关运算符

- 运算符new和new[]分别调用如下函数（分配函数）：

```
void * operator new(std::size_t); //used by new
void * operator new[](std::size_t); //used by new[]
```

- 运算符delete和delete[]分别调用如下函数（释放函数）：

```
void * operator delete(void *);
void * operator delete[](std::size_t);
```

- 使用运算符重载（后续内容），std::size_t是一个typedef，对应于合适的整型

3.2.3 C++中的相关运算符

- 举例：

  int *pi=new int; 实际被转换成int *pi=new(sizeof(int))

  int *pa=new int[40]; 实际被转换成int *pa=new(40*sizeof(int))

  delete pi; 实际被转换成delete (pi)函数调用

- 说明：

new和delete可以根据需要进行定制（因为是可替换函数）。例如：定义作用域为类的替换函数，满足该类的内存分配需求。代码实现时仍然使用new运算符，但实际调用自定义的new()函数

目录

- 动态内存的基本概念
- C中的相关函数
- C++中的相关运算符
- 内存的动态申请与释放
- 定位new运算符

3.2.4 内存的动态申请与释放

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

3.2.4 内存的动态申请与释放

<table>
<tr><th>申请对象</th><th>C的函数方式</th></tr>
<tr><td rowspan="3">一维数组</td><td>形式1：先定义指针变量，再申请
int *p;
p = (int *)malloc(10*sizeof(int));
p = (int *)calloc(10, sizeof(int));</td></tr>
<tr><td>形式2：定义指针变量的同时申请空间
char *name = (char *)malloc(10*sizeof(char));
char *name = (char *)calloc(10, sizeof(char));</td></tr>
<tr><td>说明：
虽然初次申请时也可以用
p = (int *)realloc(NULL, 10*sizeof(int)); 但一般不用</td></tr>
</table>

3.2.4 内存的动态申请与释放

<table>
<tr><th>申请对象</th><th>C++的运算符</th></tr>
<tr><td rowspan="3">一维数组</td><td>形式1：先定义指针变量，再申请
int *p;
p = new int[10]; //申请10个int型空间</td></tr>
<tr><td>形式2：定义指针变量的同时申请空间
char *name=new char[10]; //申请10个char</td></tr>
<tr><td>形式3：申请空间时赋初值
• 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</td></tr>
</table>

3.2.4 内存的动态申请与释放

- 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

```
例：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}; //错误
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"}; //正确
```

3.2.4 内存的动态申请与释放

<table>
<tr><th>申请对象</th><th>C的函数方式</th></tr>
<tr><td rowspan="3">二维数组</td><td>形式1：先定义指针变量，再申请
int (*p)[4];
p = (int (*)[4])malloc(3*4*sizeof(int));
p = (int (*)[4])calloc(3*4, sizeof(int));</td></tr>
<tr><td>形式2：定义指针变量的同时申请空间
int (*p)[4] = (int (*)[4])malloc(3*4*sizeof(int));
int (*p)[4] = (int (*)[4])calloc(3*4, sizeof(int));</td></tr>
<tr><td>说明：
虽然初次申请时也可以用
p = (int (*)[4])realloc(NULL, 3*4*sizeof(int));但一般不用</td></tr>
</table>

3.2.4 内存的动态申请与释放

<table>
<tr><th>申请对象</th><th>C++的运算符</th></tr>
<tr><td rowspan="3">二维数组</td><td>形式1：先定义指针变量，再申请
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</td></tr>
<tr><td>形式2：定义指针变量的同时申请空间
float (*p)[4]=new float[3][4];</td></tr>
<tr><td>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</td></tr>
</table>

3.2.4 内存的动态申请与释放

- 动态申请的二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，其余规则同二维数组定义时初始化

```
例：int (*p)[3];
        p = new int[2][3] {1,2,3,4,5,6};     //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};             //错误
        p = new int[2][3] {{1},{4}};         //正确
例：char (*p)[6];  //注：字符型在使用字符串方式初始化时，一层{}
    p = new char[2][6] {'A','B','C' };     //错误
    p = new char[2][6] {{'A'},{'B','C' }};//正确
    p = new char[2][6] {"Hello","China"}; //正确
    p = new char[2][6]{"Hello1","China"}; //错误
```

小结：内存的动态申请与释放（C方式）

释放对象	C的函数方式

普通变量	int *p = (int *)malloc(sizeof(int));
	int *p = (int *)calloc(1, sizeof(int));
	free(p);

一维数组	int *p = (int *)malloc(10*sizeof(int));
	int *p = (int *)calloc(10, sizeof(int));
	free(p);

二维数组	int (*p)[4] = (int (*)[4])malloc(3*4*sizeof(int));
	int (*p)[4] = (int (*)[4])calloc(3*4, sizeof(int));
	free(p);

小结：内存的动态申请与释放（C++方式）

释放对象	C++的运算符
普通变量	int *p = new int; delete p;
一维数组	int *p = new int[10]; delete []p; • 某些资料上说可以 delete name，因为一维数组地址可理解为首元素地址，不加[] • 对于int/char等基本类型的数组，加不加均正确
二维数组	int (*p)[4] = new int[3][4]; delete []p; • 二维以上必须加一个[]，否则编译警告

- 静态数据区、动态数据区、动态内存分配区(堆空间)的地址各不相同

```
#include <iostream>
#include <cstdlib>
using namespace std;
int a;
int main()
{    int b;
     int *c;
     c = (int *)malloc(sizeof(int));
     cout << &a <<endl;
     cout << &b <<endl;
     cout << &c << ' ' << c <<endl;
     free(c);
     return 0;
}
```

```
00BEC13C
0034F970
0034F964 006DB298
```

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、本课程课件中，为了节约空间，大部分示例程序省略了判断

实际写程序时要判断！！！

作业中会人工审核并扣分！！！

后续不再强调，自行补充所有ppt示例程序

- C：通过强制类型转换将void型的指针转为其它类型

```
int main()
{    int *p;
     p=(int *)malloc(10*sizeof(int));
       //或 p=(int *)calloc(10, sizeof(int));
     if (p==NULL) {
         cout << "No Memory" << endl;
         return 0;
     }
     ...
     free(p);
     ...
     return 0;
}
```

申请10个int型的变量空间不建议直接写成
malloc(40)/calloc(10,4)
因为程序的适应性差

- C++：申请时自动确定类型

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

- 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

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    int i, *p;
    p = &i;      //错误原因：p不是动态申请的空间
    free(p);
    return 0;
}
```

编译不错执行错

- 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

```
#include <iostream>
#include <stdlib.h>
using namespace std;
int main()
{   int *p;
    p = (int*)malloc(sizeof(int));
    if (p != NULL) {
        p++;      //错误原因：p已不指向动态申请的空间
        free(p);
    }
    return 0;
}
```

编译不错执行错

- 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

```
#include <iostream>
//不需要包含头文件stdlib
using namespace std;
int main()
{
    int i, *p;
    p = &i;     //错误原因：p不是动态申请的空间
    delete p;
    return 0;
}
```

编译不错执行错

- 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

```
#include <iostream>
//不需要包含头文件stdlib
using namespace std;
int main()
{
    int *p;
    p = new(nothrow) int;
    if (p != NULL)
    {   p++;   //错误原因：p已不指向动态申请的空间
        delete p;
    }
    return 0;
}
```

编译不错执行错

- new申请失败会抛出异常bad_alloc，需要使用 try-catch来处理异常

```
#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;
}
```

```
//或者：
try {
    while (1) {
    p = new char[1024 * 1024];
    count++;}
}
catch (const bad_alloc &mem_fail) {
    cout << mem_fail.what() << endl;
    break;
}
```

- 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 num = 0;
    while(1) {
        p = (char *)malloc(1024*1024*sizeof(char));
        if (p == NULL)
            break;
        num ++;
    }
    cout << num << " MB" << endl;
    return 0;
}
```

循环内：每次均申请空间，完成后不释放，且p不再指向，导致内存泄露。循环至内存耗尽

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

循环内：每次均申请空间，完成后不释放，且p不再指向，导致内存泄露。循环至内存耗尽

- 动态内存申请的空间若不释放，则程序退出时操作系统会自动回收

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

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

p所申请的空间可能：

1）在程序运行结束后由操作系统回收

2）逐渐耗尽内存

坚决反对此种用法，且不是所有的操作系统都支持，作业中重点审核！

- realloc申请时新老空间可重合，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，因此不需要再释放老空间

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    int *p;
    p = (int *)malloc(10 * sizeof(int));
    cout << p << endl;
    p = (int *)realloc(p, 20 * sizeof(int));
    cout << p << endl;
    free(p);
    return 0;
}
```

realloc的常规用法：传入指针和返回指针用同一个

//再次注意此例未判断申请是否成功，自行补充，后续例子类似

- realloc申请时新老空间可重合，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，因此不需要再释放老空间

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    int *p;
    p = (int *)malloc(10 * siz
    cout << p << endl;
    p = (int *)realloc(p, 20 *
    cout << p << endl;
    free(p);
    return 0;
}
```

```
//非常规用法
int *p, *q;
p  = (int *)malloc(10 * sizeof(int));
cout << p << endl;
q = (int *)realloc(p, 20 *sizeof(int));
cout << p << ' ' << q << endl;
free(p); //需注释掉此行才正确
free(q);
```

realloc的常规用法：传入指针和返回指针用同一个

//再次注意此例未判断申请是否成功，自行补充，后续例子类似

• realloc申请时，原空间原有内容会被复制到新空间

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{   int i, *p;
    p = (int *)malloc(10*sizeof(int));
    cout << "地址:" << p << endl;
    for(i=0; i<10; i++)
        p[i] = (i+1)*(i+1);     //[0-9]的值为1-10的平方

    p = (int *)realloc(p, 20*sizeof(int));
    cout << "新地址:" << p << endl;
    for(i=0; i<20; i++)
        cout << *(p+i) << endl;
    free(p);
    return 0;
}
```

```
地址:006CAEA8
新地址:006CB3C0
1
4
9
16
25
36
49
64
81
100
-842150451
-842150451
-842150451
-842150451
-842150451
-842150451
-842150451
-842150451
-842150451
-842150451
-842150451
```

//例1：申请一个int型空间：

```c
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
   int *p;
   p = (int *)malloc(sizeof(int));
   *p = 10;
   cout << *p << endl;
   free(p); //记得释放
   return 0;
}
```

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

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

//例2：申请10个int型空间当一维数组用：指针法/下标法均可

```
#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);
   return 0;
}
```

```
#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;
   return 0;
}
```

//例3：申请10个int型空间当一维数组用：
用head记住申请的首地址，便于复位和释放，工作指针p可++/--

```
#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);
   return 0;
}
```

```
#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;
   return 0;
}
```

//例4：申请12个int型当二维数组使用：
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);  return 0;
}
```

```
//换成C++方式程序其余相同
head = p = new(nothrow) int[3][4];
…
delete []head;
```

- 嵌套进行动态内存申请：按从外到内的顺序进行申请，反序进行释放

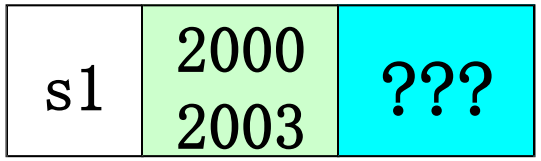
```
#include <iostream>
#include <cstdlib>
using namespace std;
struct student {
    int num;  char *name;
};
int main()
{  student *s1;
   s1 = (student *)malloc(sizeof(student)); //申请8字节
   s1->name = (char *)malloc(6*sizeof(char));//申请6字节
   s1->num = 1001;  strcpy(s1->name, "zhang");
   cout << s1->num << ":" << s1->name << endl;
   free(s1->name);//释放6字节
   free(s1);//释放8字节
   return 0;
}
```

申请：先student变量，再name成员
释放：先name变量，再student成员

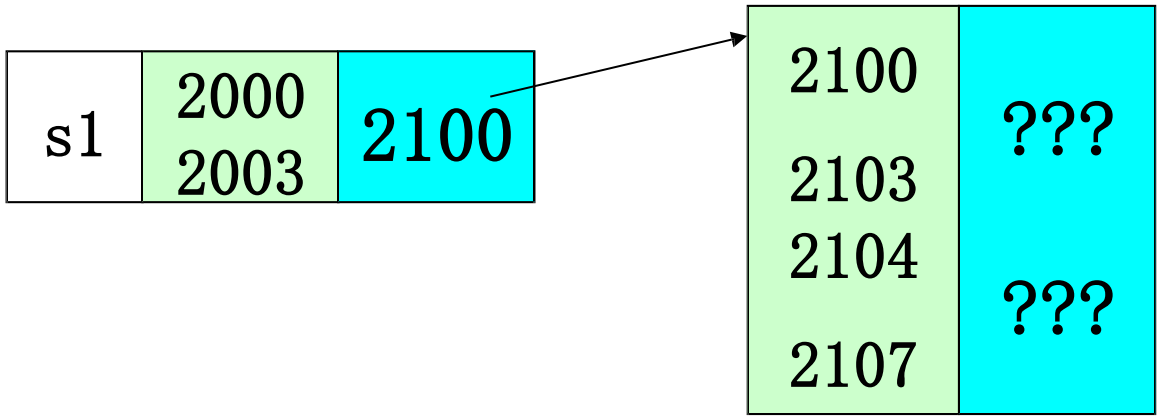
- 嵌套进行动态内存申请：按从外到内的顺序进行申请，反序进行释放

```
#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;
}
```

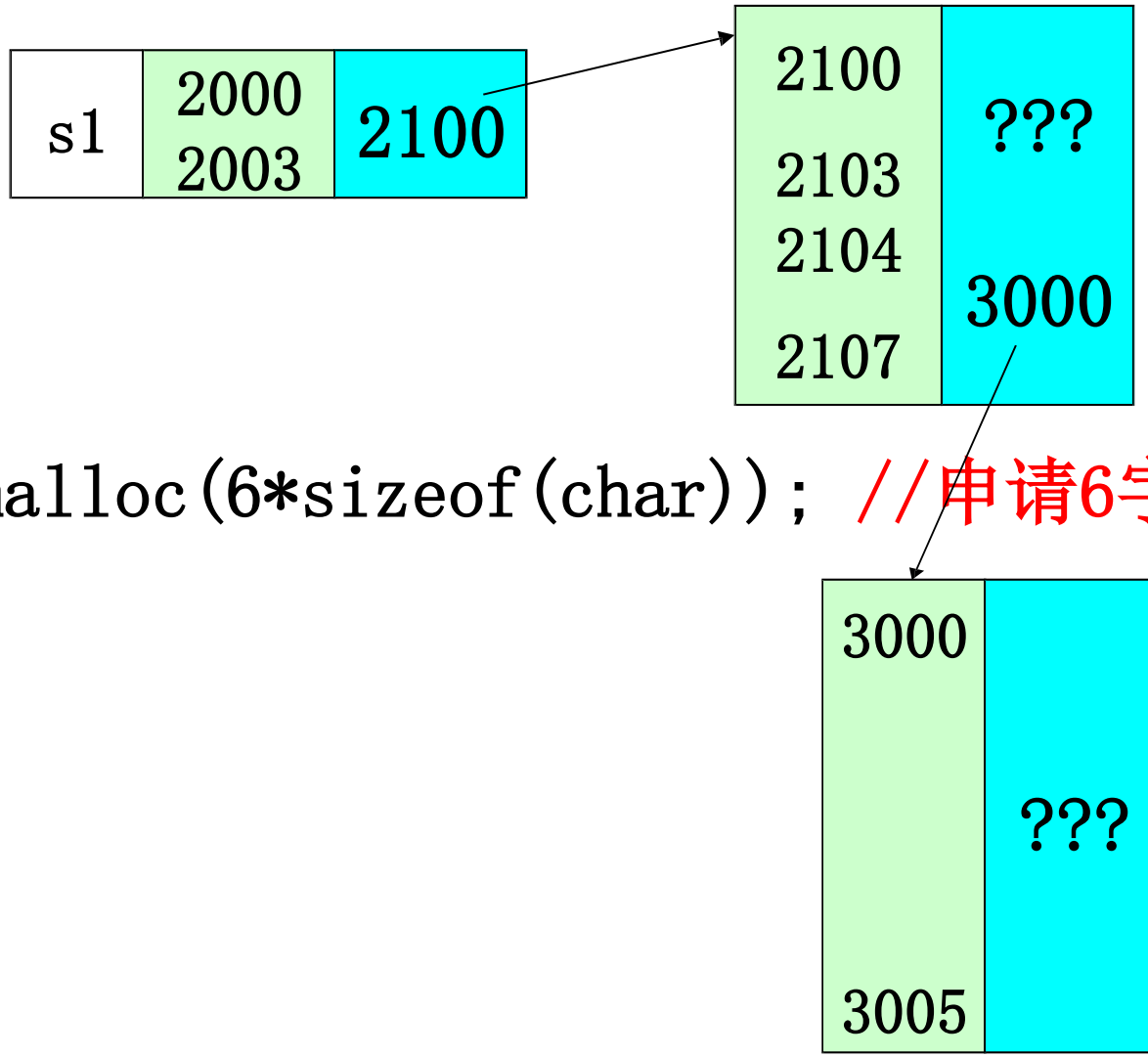
申请：先student变量，再name成员
释放：先name变量，再student成员



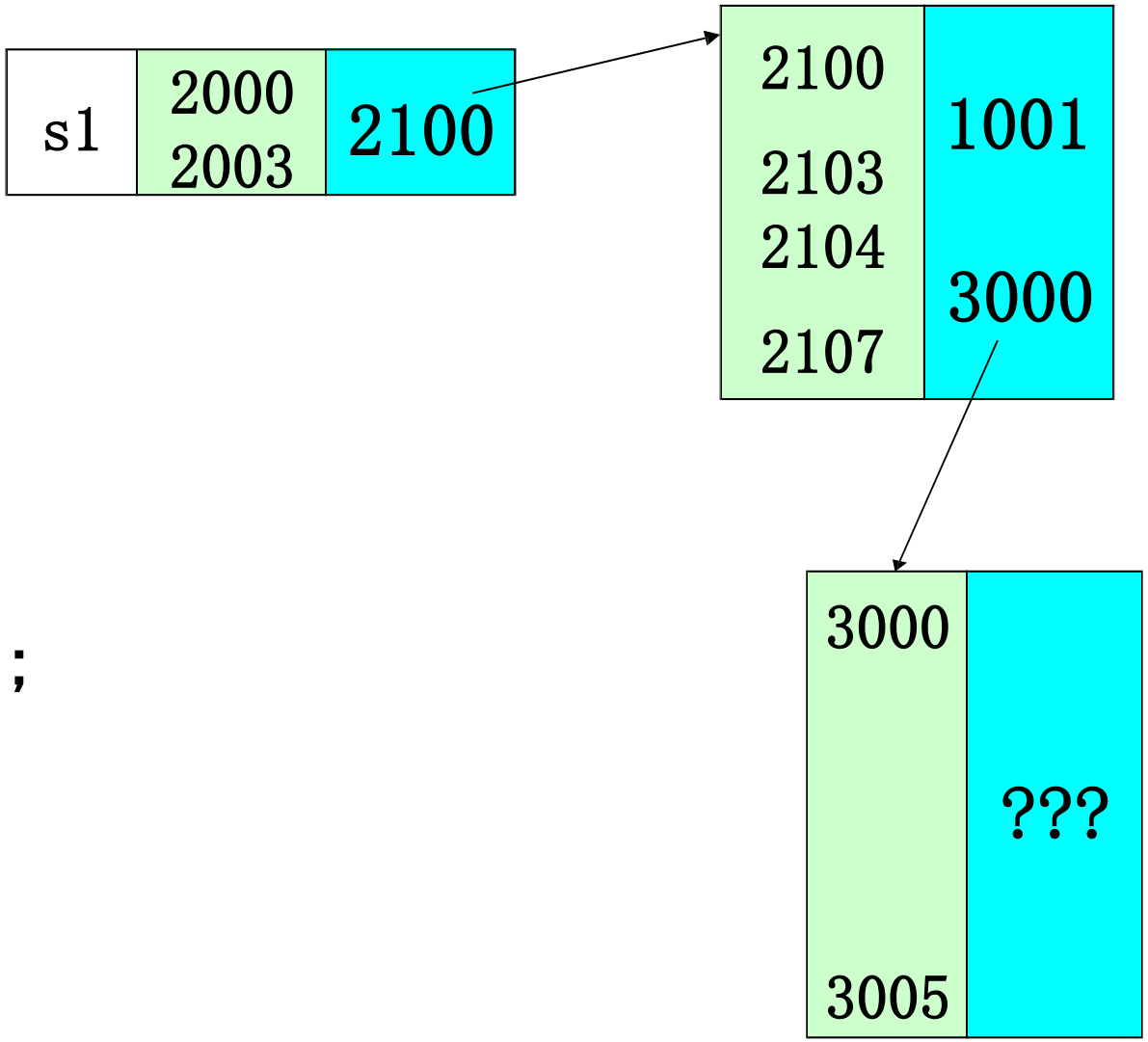

```
student *s1;
```



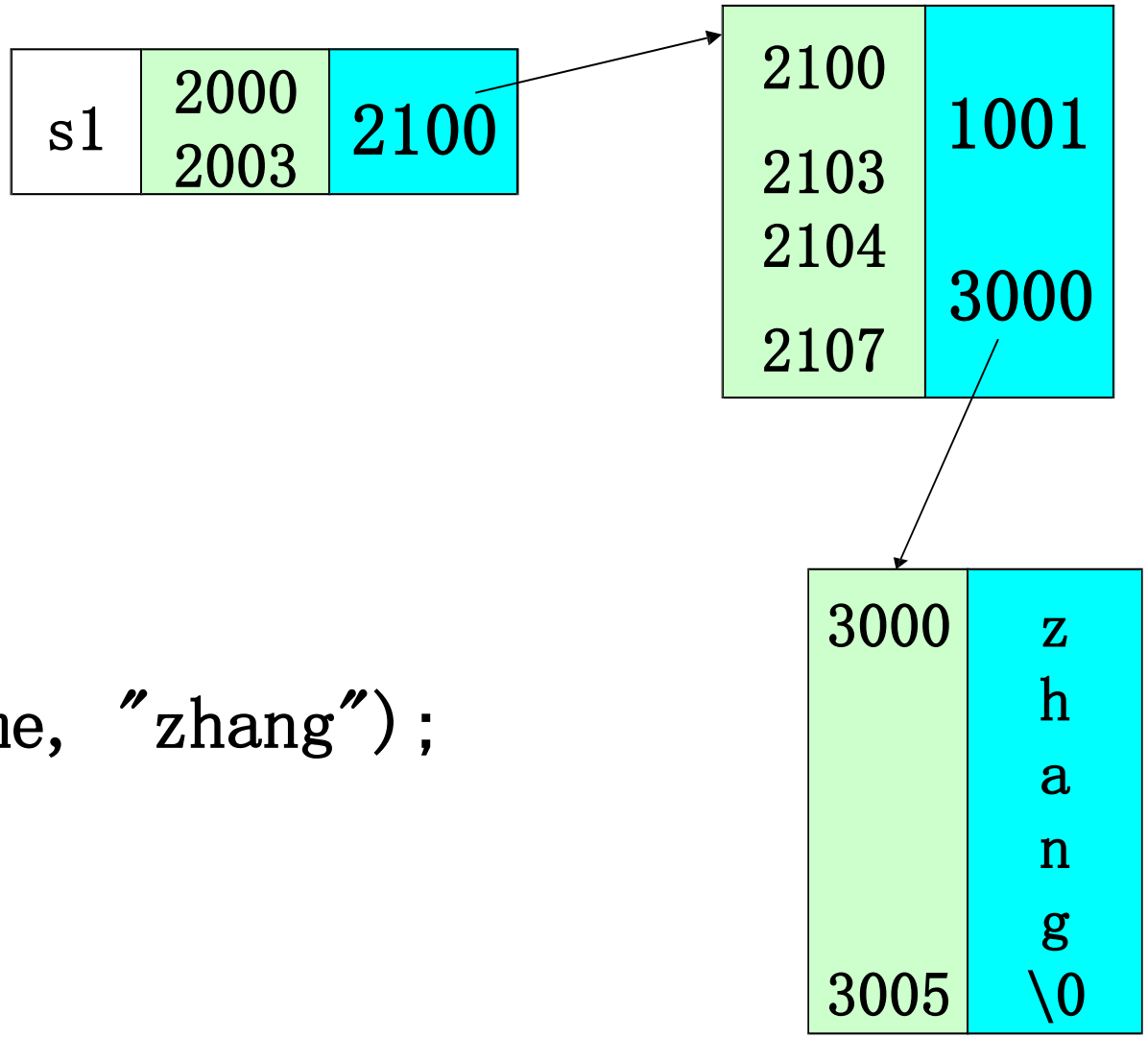

```
s1 = (student *)malloc(sizeof(student)); //申请8字节
```



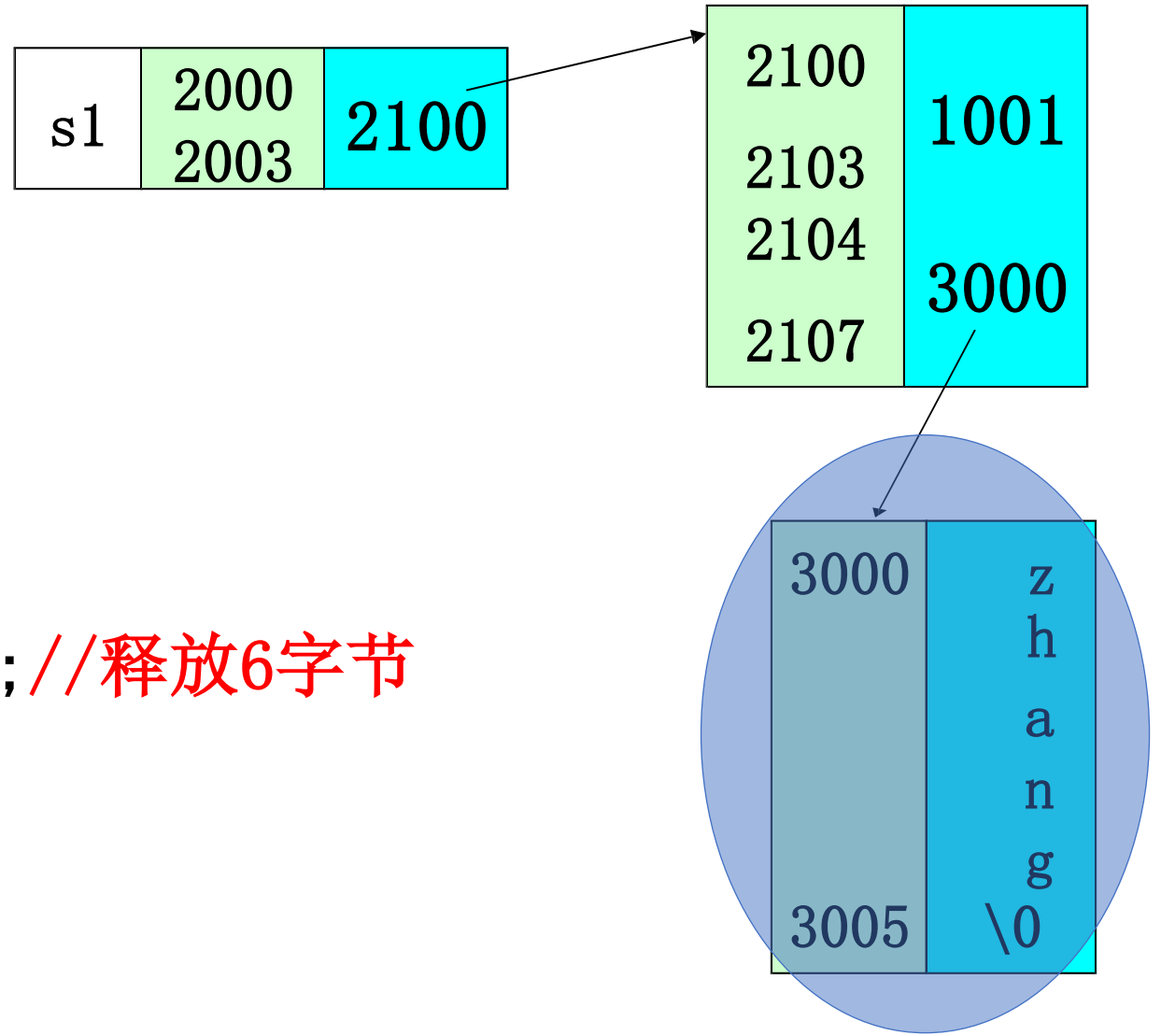

```
s1->name = (char *)malloc(6*sizeof(char)); //申请6字节
```



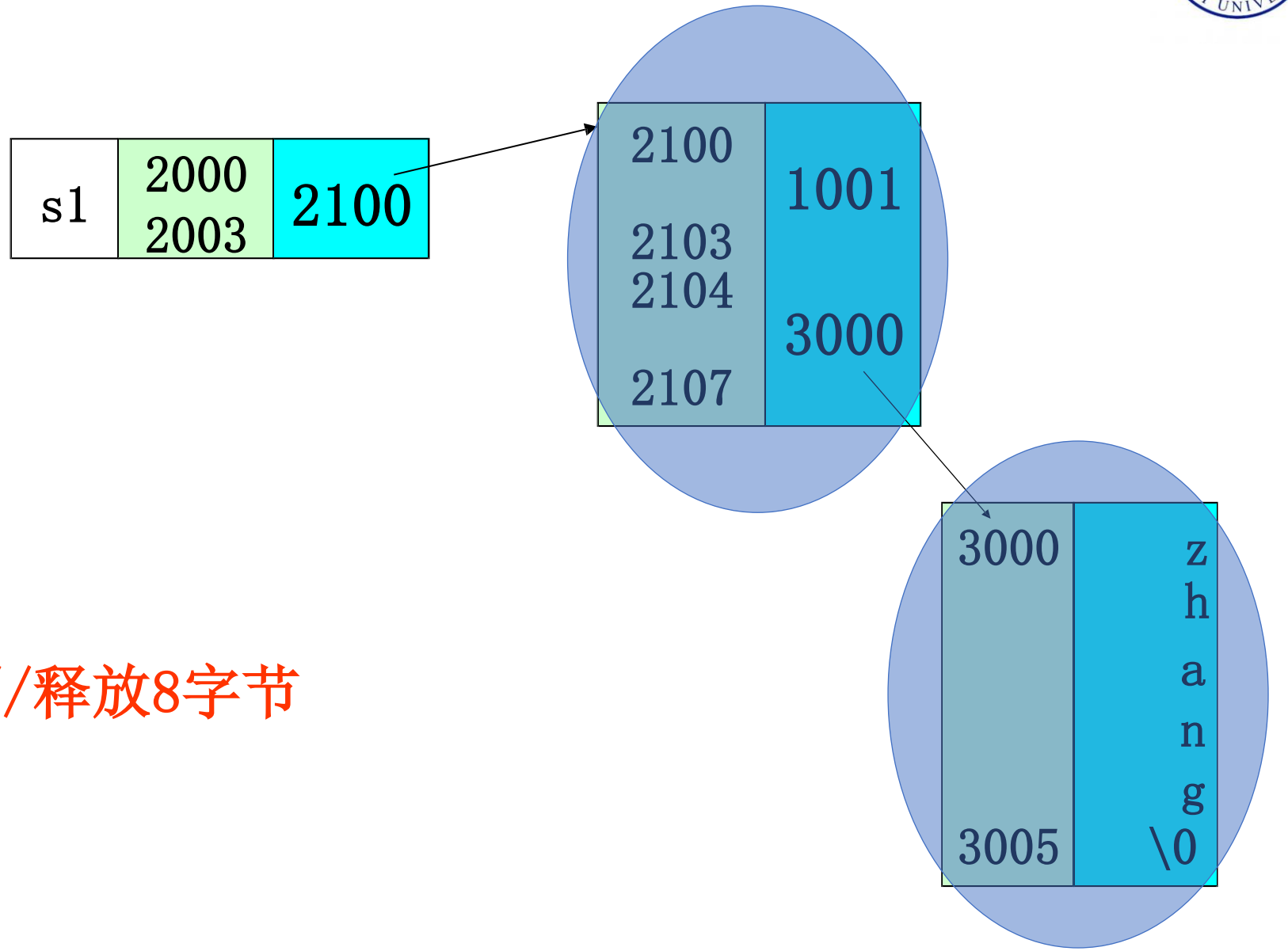
s1->num = 1001;



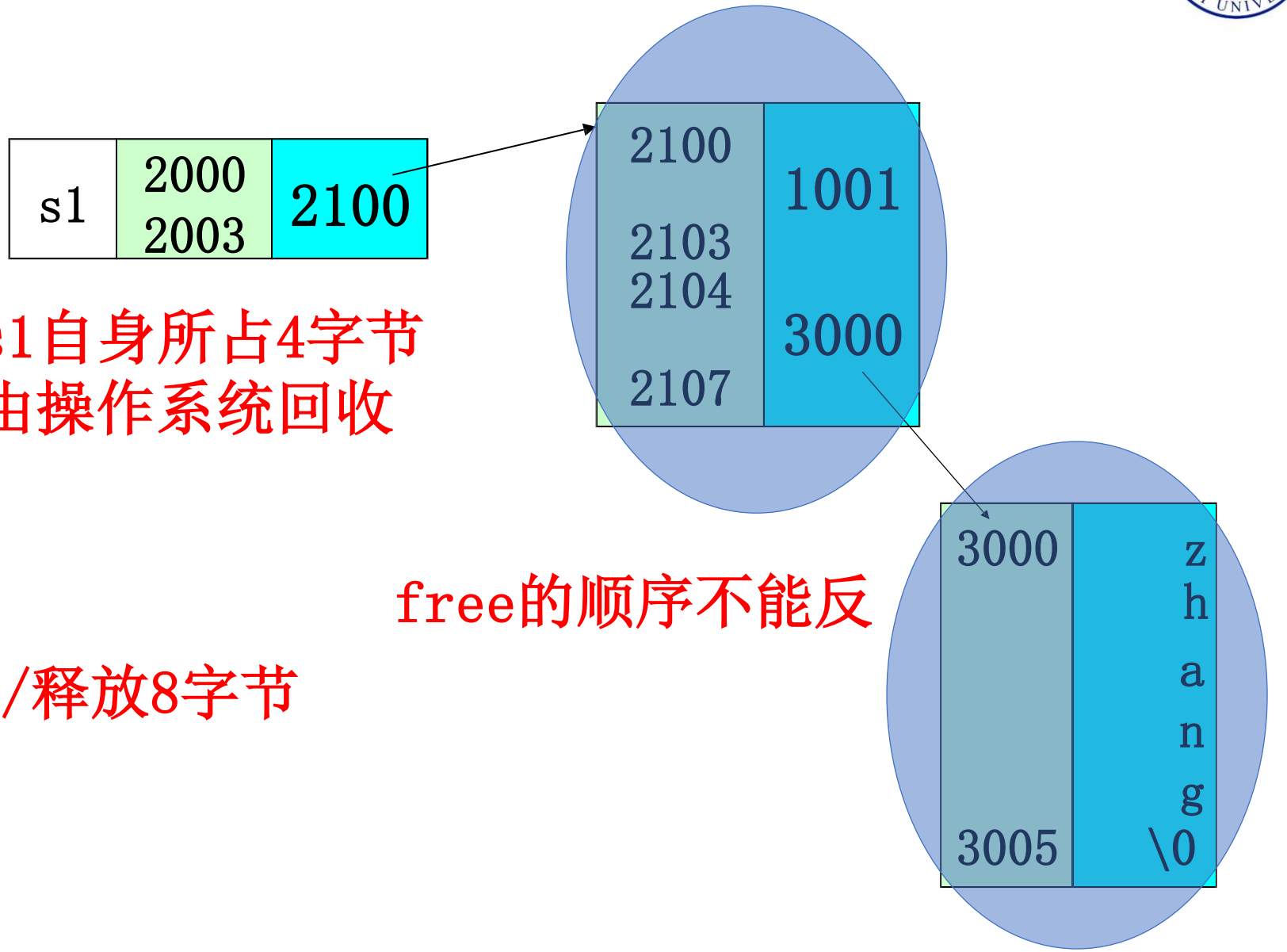
strcpy(s1->name, "zhang");



free(s1->name);//释放6字节



free(s1);//释放8字节



s1自身所占4字节
由操作系统回收

free的顺序不能反

free(s1);//释放8字节

- 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));
    *p = 10;
    cout << *p << endl;
    free(p); //释放
    free(p); //再次释放，致运行出错
}
```

```
#include <iostream>

using namespace std;
int main()
{
    int *p;
    p = new(nothrow) int;
    *p = 10;
    cout << *p << endl;
    delete p; //释放
    delete p; //再次释放，致运行出错
}
```

//例5：

```
#include <iostream>
#include <string>
using namespace std;
struct student
{
    string name;
    int num;
    char sex;
};
```

```
int main()                                          //错误
{
  student *p;
  p = (student *)malloc(sizeof(student));
  if (p == NULL)
     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;
}
```

对于非内部数据类型的对象而言，malloc/free无法满足动态对象的要求

malloc/free是库函数，不在编译器控制权限之内，不能执行构造函数和析构函数

//例5：

```
#include <iostream>
#include <string>
using namespace std;
struct student
{
    string name;
    int num;
    char sex;
};
```

```
int main()
{
  student *p;
  p = new(nothrow) student;
  if (p == NULL)
      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;
}
```

//正确

运算符new能完成动态内存分配和初始化工作
运算符delete能完成清理与释放内存的工作

//例6：malloc/free和new/delete的本质区别

```
#include <iostream>
using namespace std;
class Obj
{
public:
    Obj(){cout << "Initialization" << endl;}
    ~Obj(){cout << "Destroy" << endl;}
};
void UseMallocFree()
{
    cout << "in UseMallocFree()..." << endl;
    Obj* a = (Obj*)malloc(sizeof(Obj));
    free(a);
}
```

```
void UseNewDelete()
{
    cout << "in UseNewDelete()
             ..." << endl;
    Obj* a = new Obj;
    delete a;
}
int main()
{
    UseMallocFree();
    UseNewDelete();
    return 0;
}
```

```
in UseMallocFree()...
in UseNewDelete()...
Initialization
Destroy
```

非内部数据类型的对象而言：
malloc/free和new/delete有区别！！

- 例7：在例5的基础上建立一个有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：

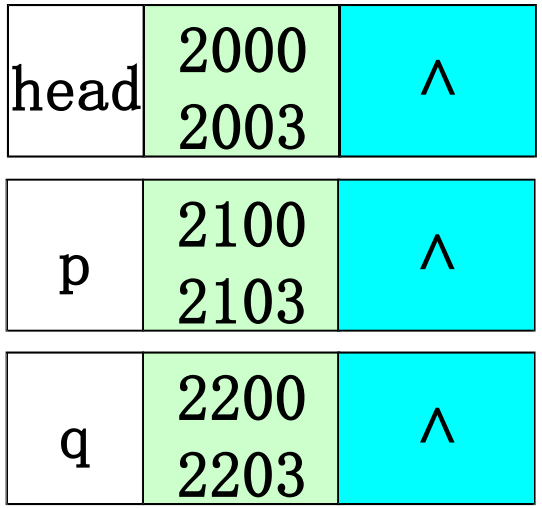
```
Zhang    1001    m
Li       1002    f
Wang     1003    m
Zhao     1004    m
Qian     1005    f
```

```
struct student {
    string name;
    int num;
    char sex;
    struct student *next;
};
```

```
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;
       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;
       p->next = NULL;
     }
}    //是否完整?
```

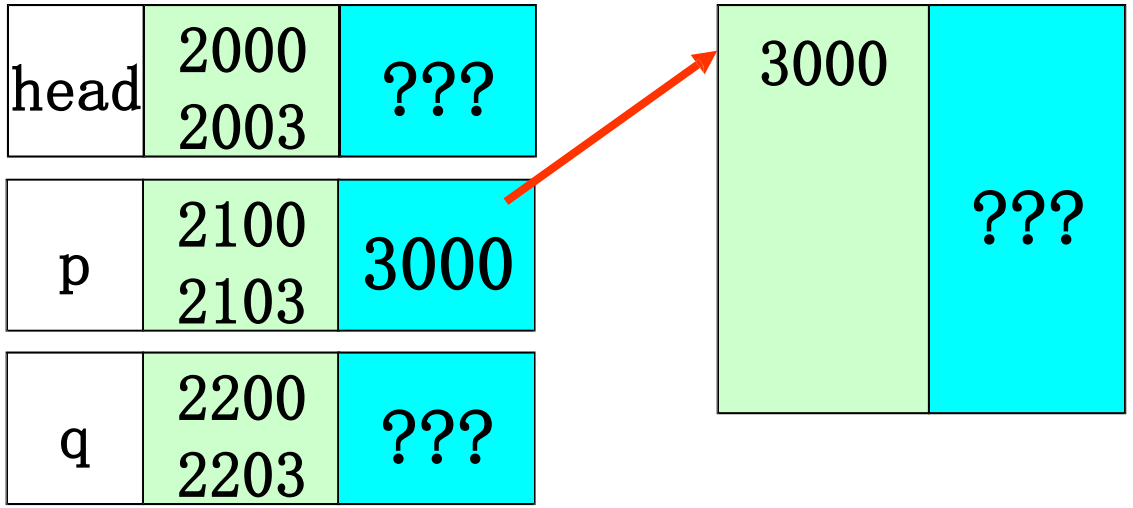
```
student *head=NULL, *p=NULL, *q=NULL; int i;
```

初始状态：



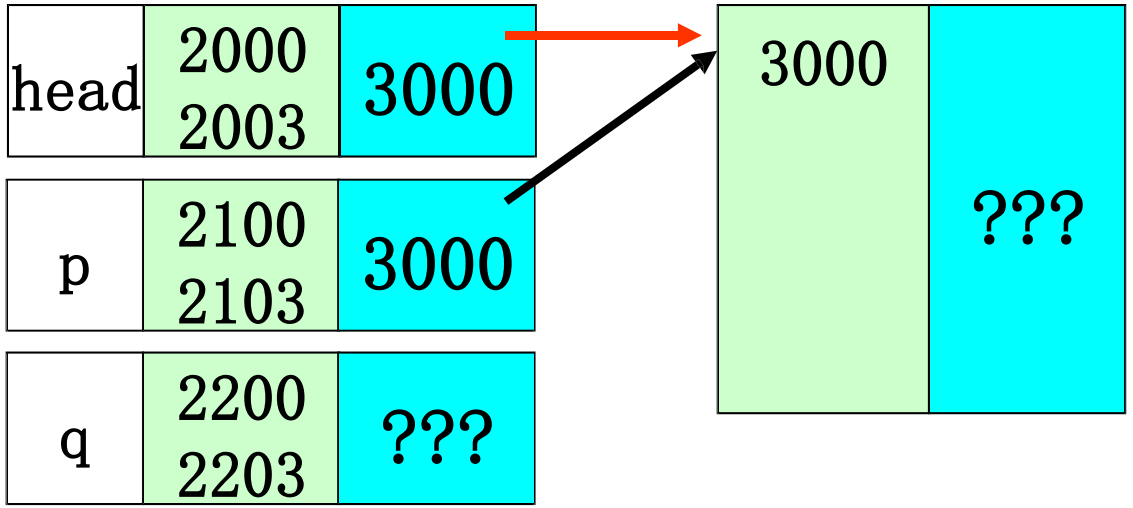
```
p = new(nothrow) student;
```

i=0的循环：



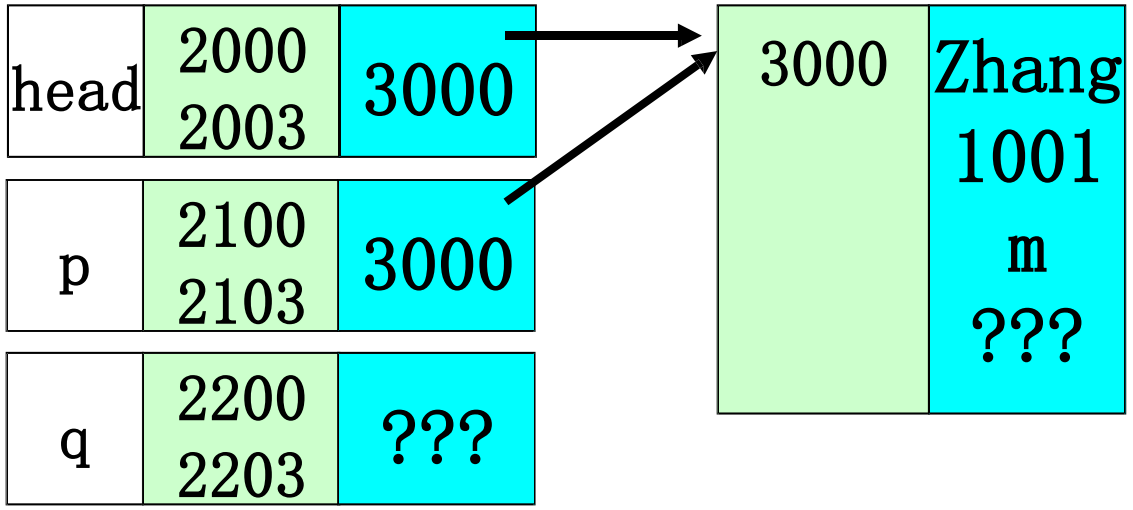
```
head = p; //head指向第1个结点
```

i=0的循环：



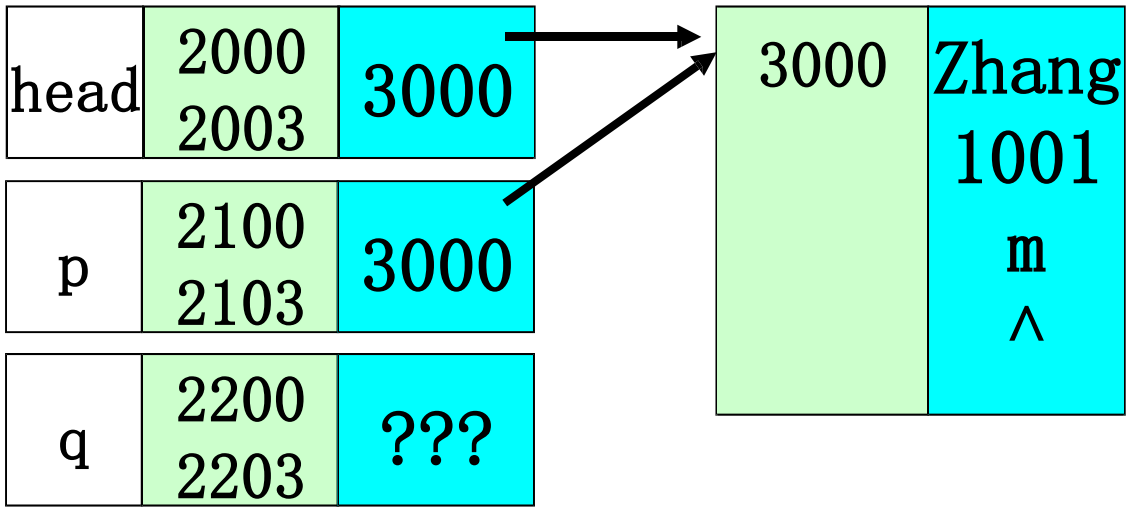
```
cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
```

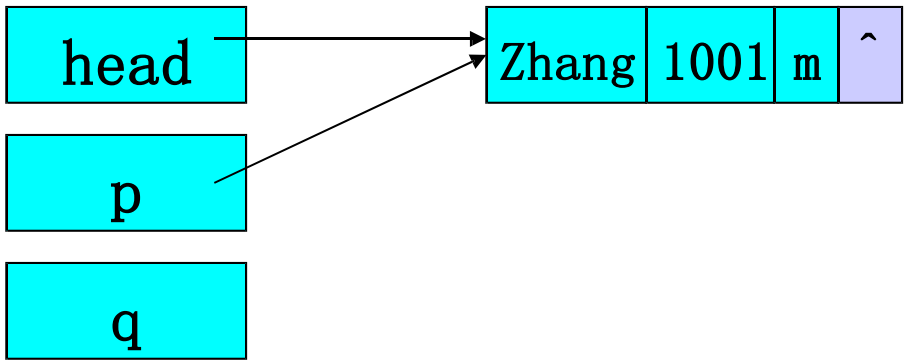
i=0的循环：



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

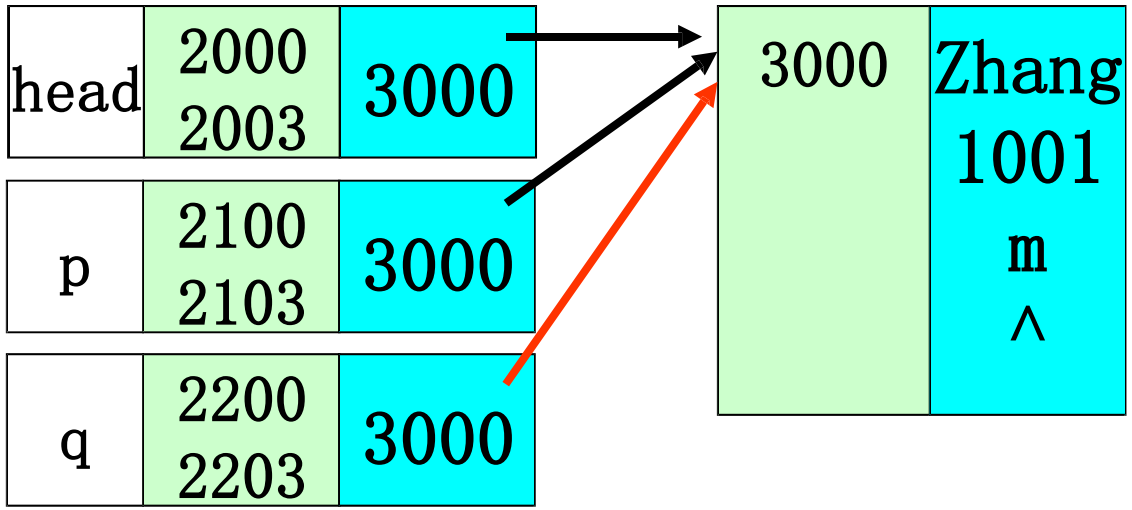
i=0的循环结束：





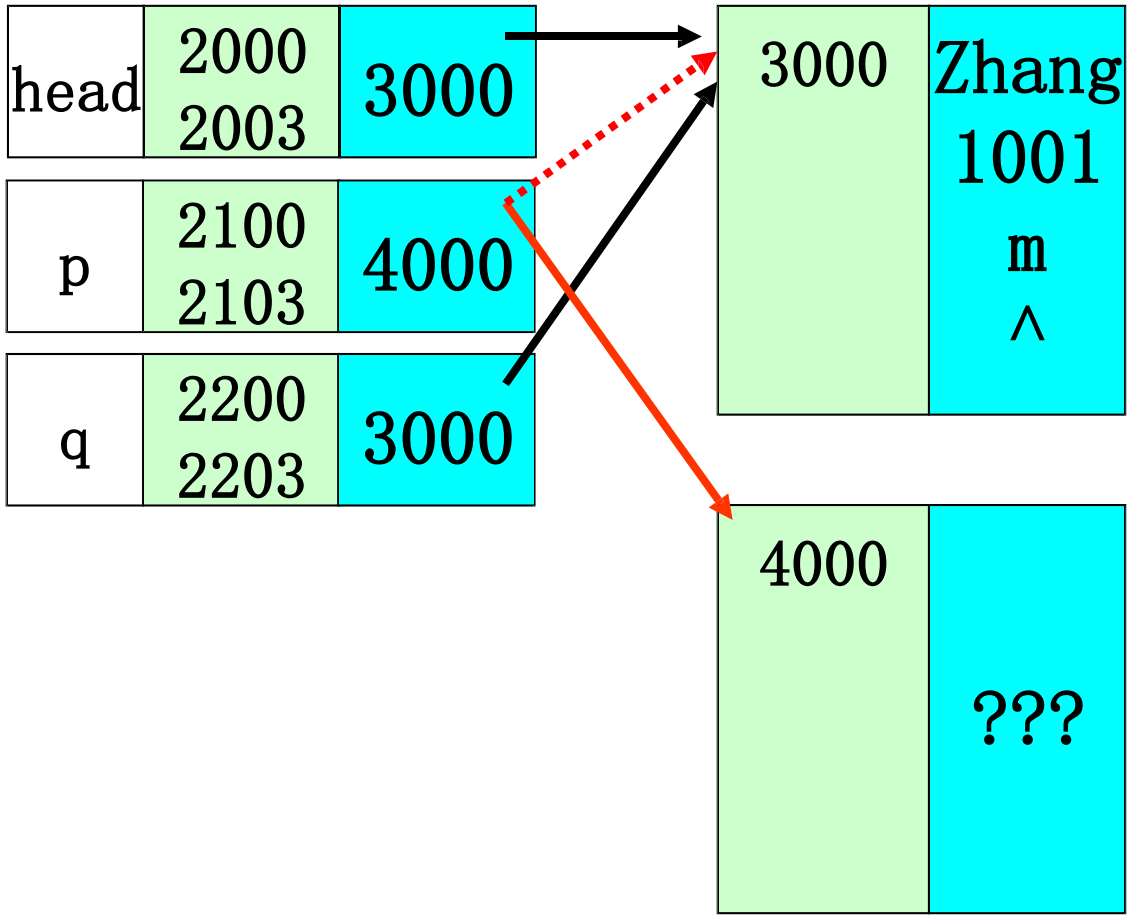
```
if (i > 0)   q = p;
```

i=1的循环：



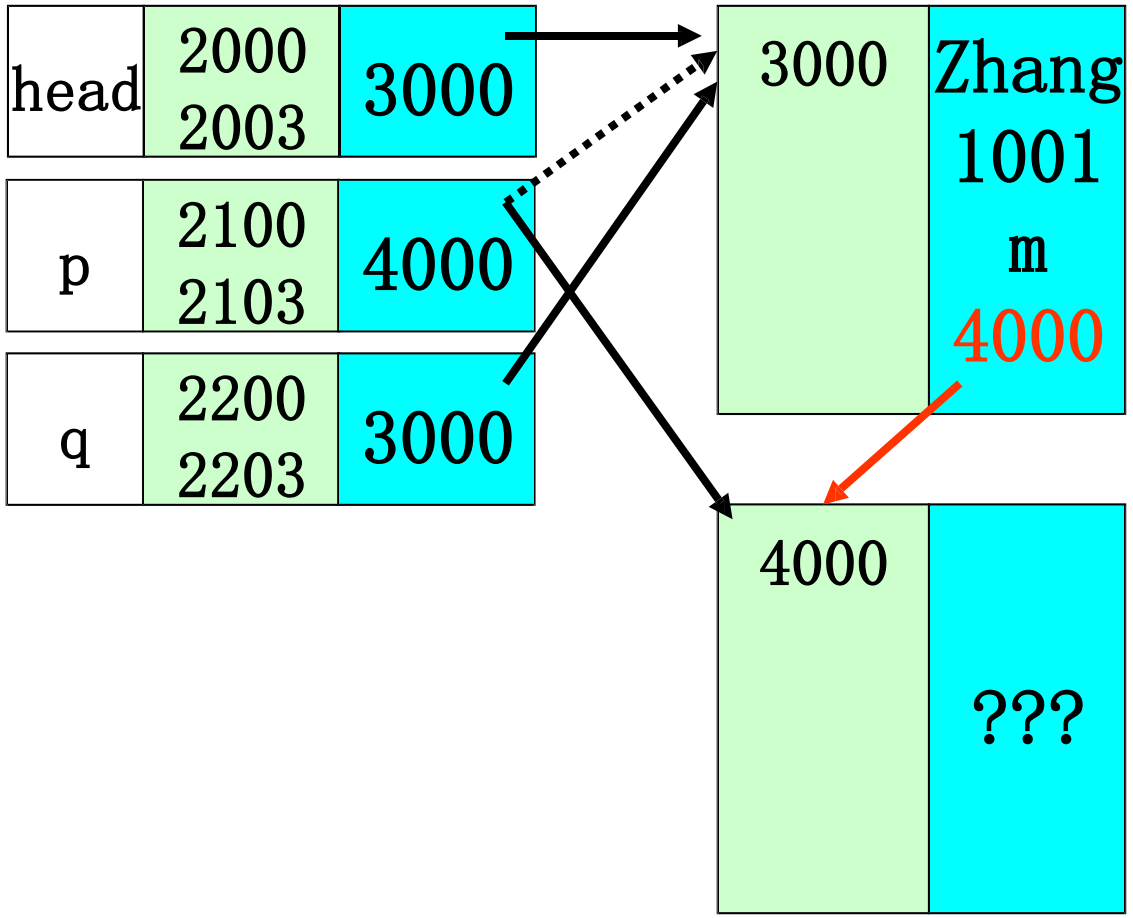
```
p = new(nothrow) student;
```

i=1的循环：



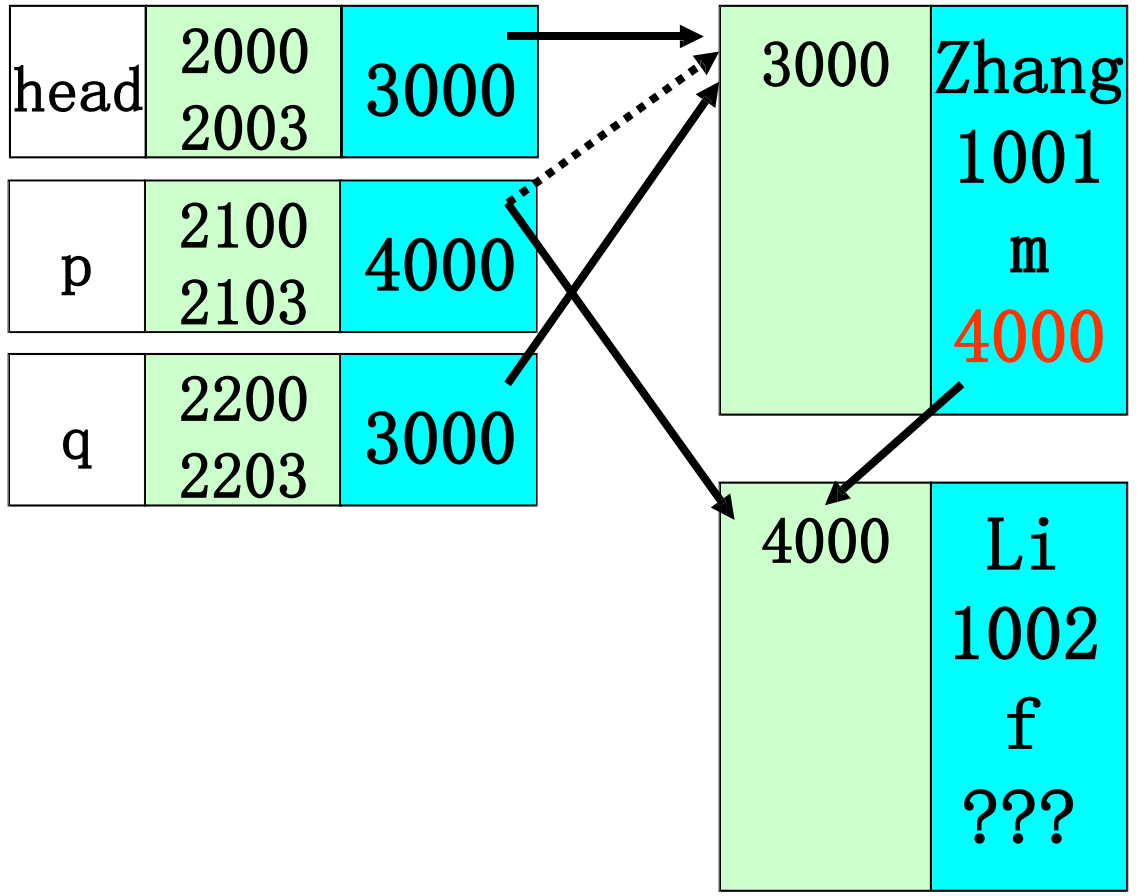
q->next = p;

i=1的循环：



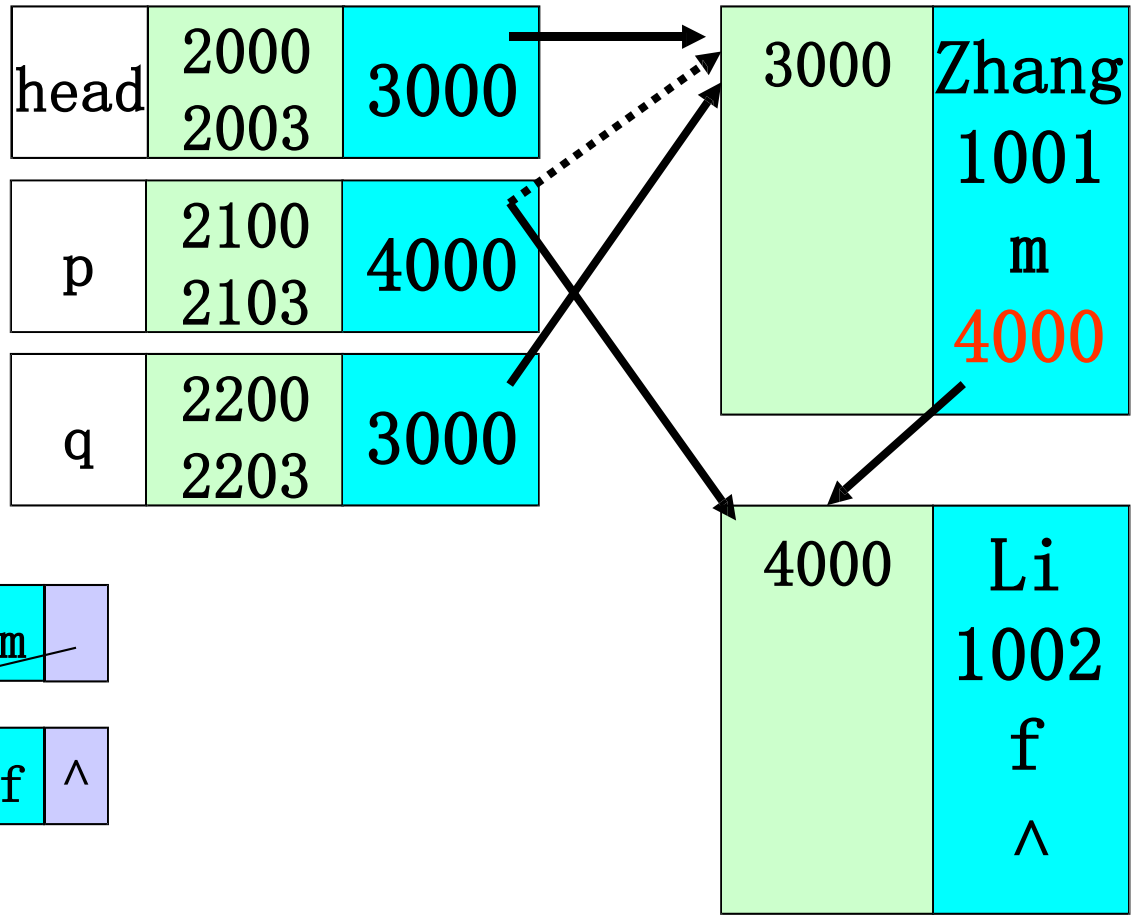
```
cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
```

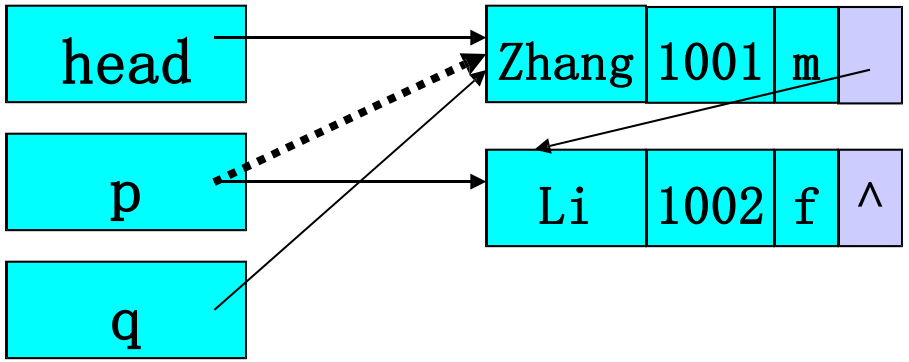
i=1的循环：



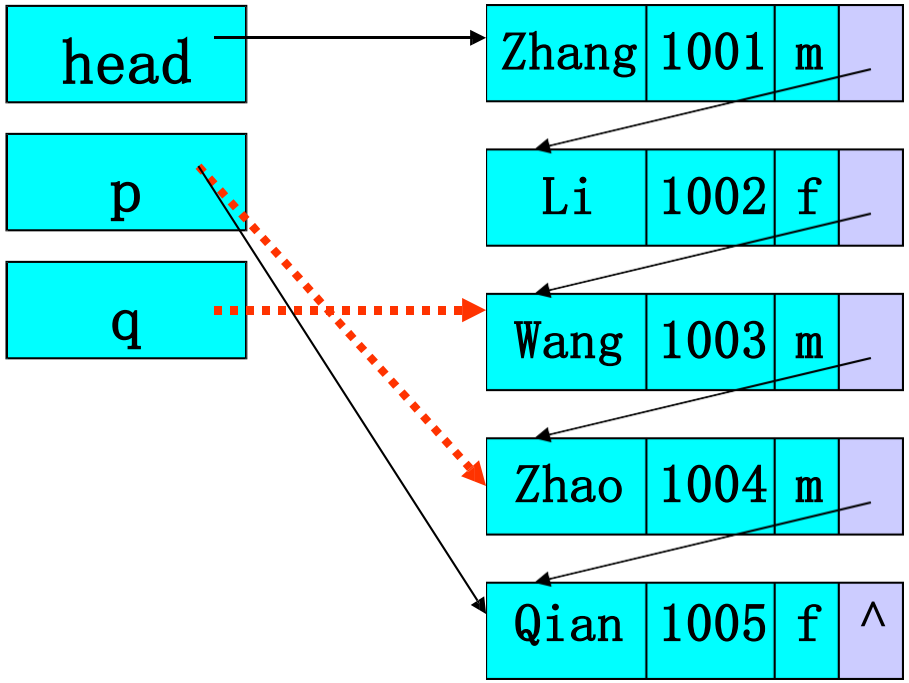
p->next = NULL;

i=1的循环结束：

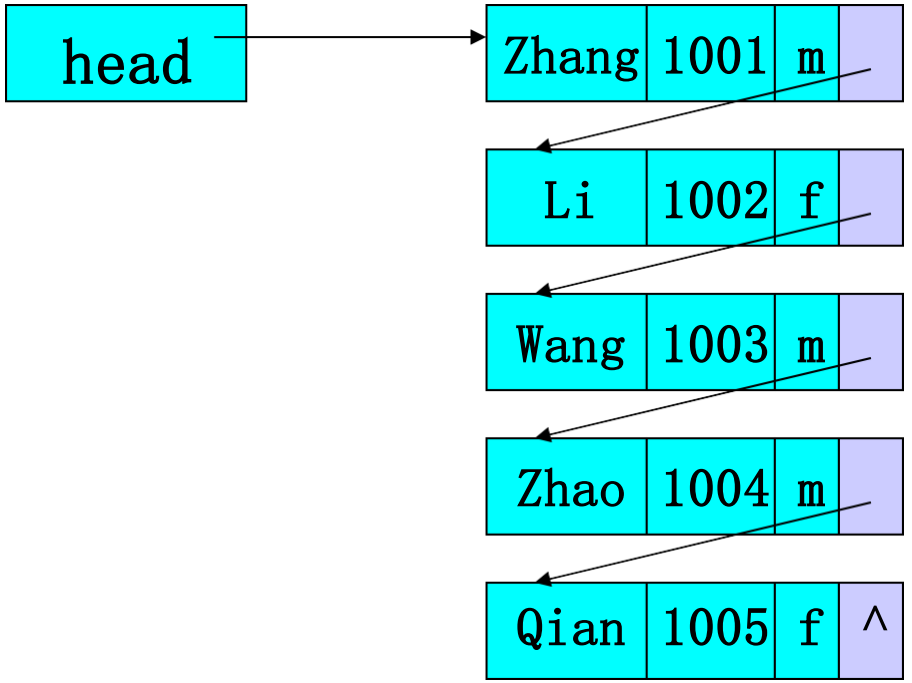




i=4的循环结束：



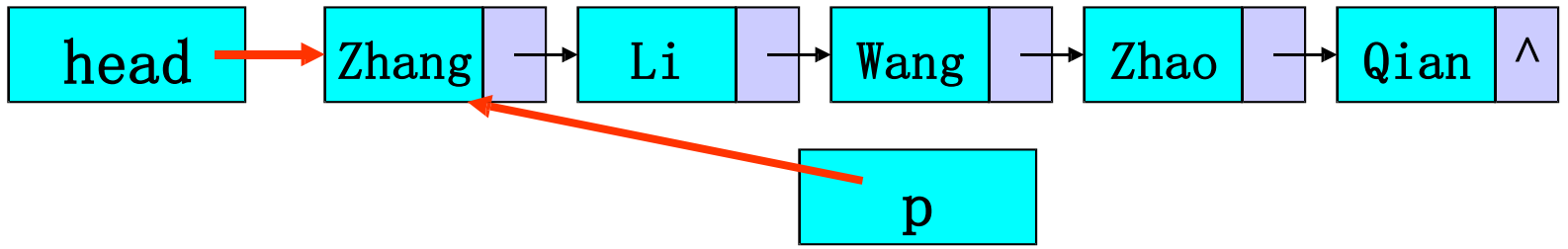
循环完成：

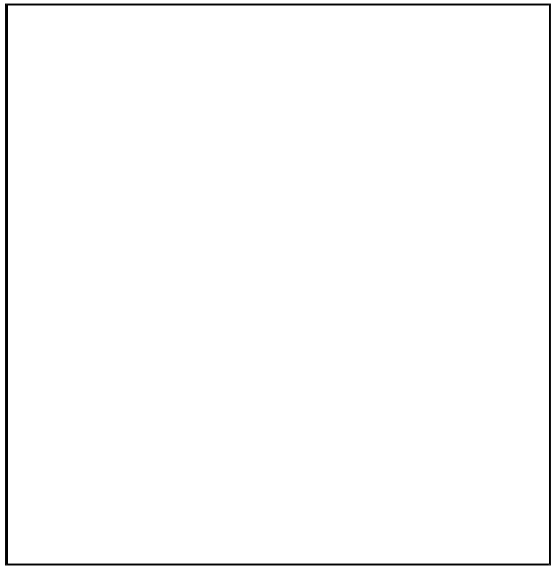


//例7：在例5的基础上建立一个有5个结点的链表

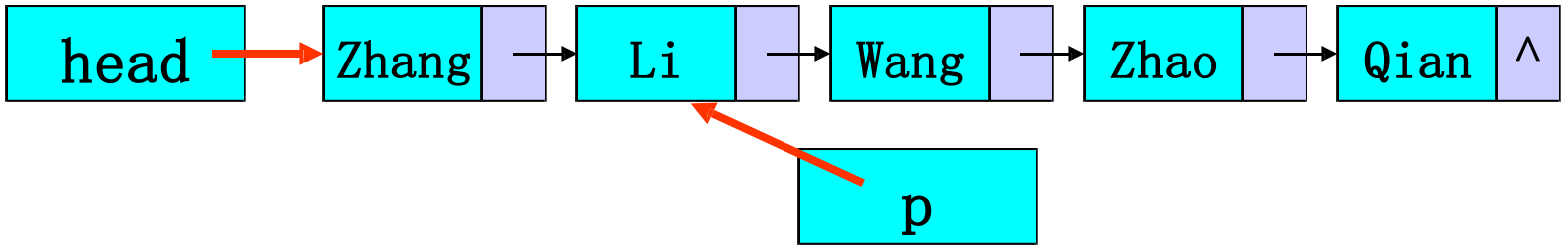
```
int main()
{   student *head = NULL, *p = NULL, *q = NULL; int i;
    for(i = 0; i < 5; i++) { …//刚才建立链表的循环}
    p = head; //p复位，指向第1个结点
    while(p != NULL) { //循环进行输出
        cout << p->name << " " << p->num << " " << p->sex << endl;
        p = p->next;
    }
    p = head; //p复位，指向第1个结点
    while(p) { //循环进行各结点释放
        q = p->next;
        delete p;
        p = q;
    }
    return 0;
}
```

p = head; //p复位，指向第1个结点





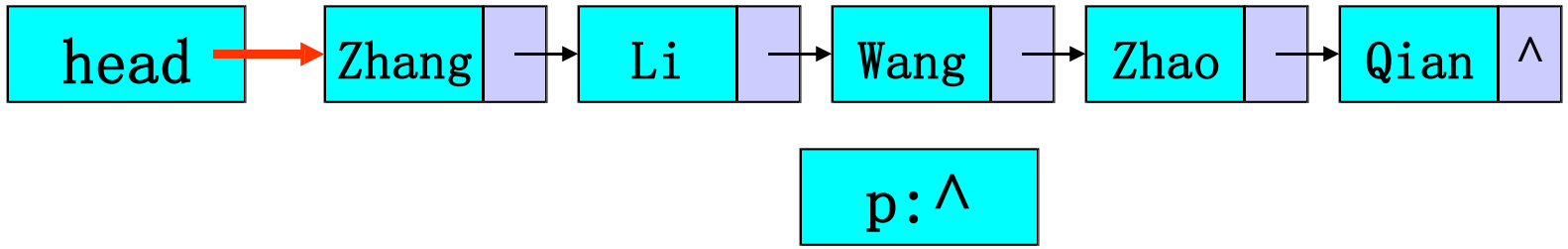
```
p = p->next;
```



Zhang	1001	m

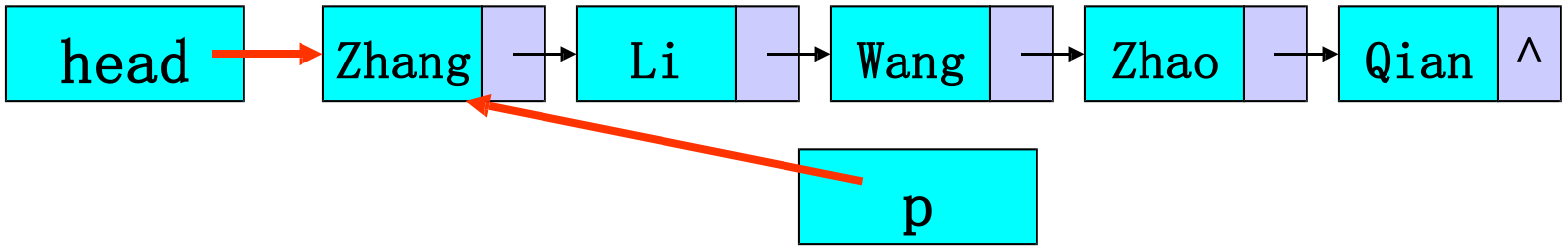
while(p!=NULL)

循环结束：

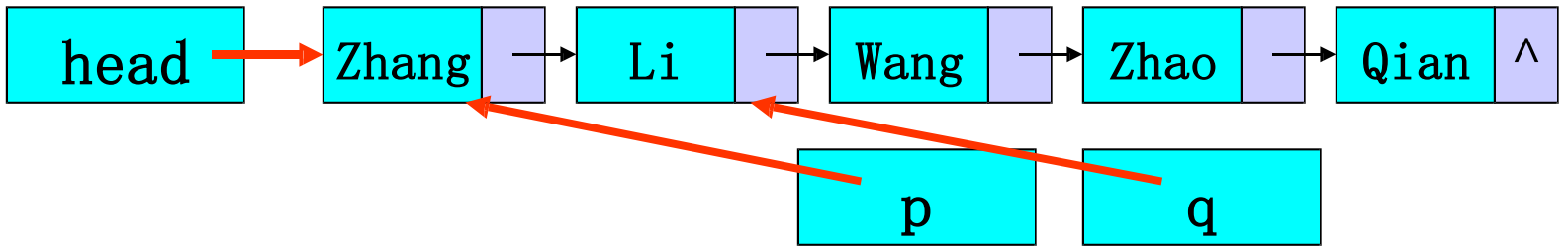



```
Zhang 1001 m
Li 1002 f
Wang 1003 m
Zhao 1004 m
Qian 1005 f
```

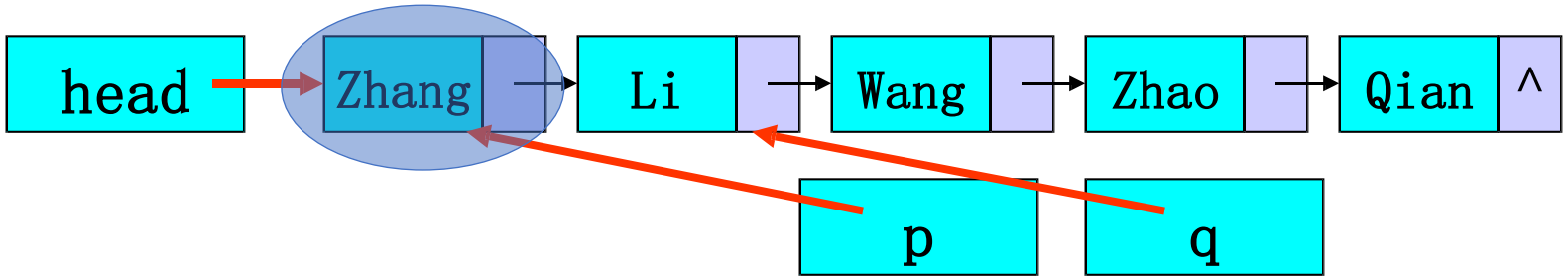
```
p = head; //p复位，指向第1个结点
```



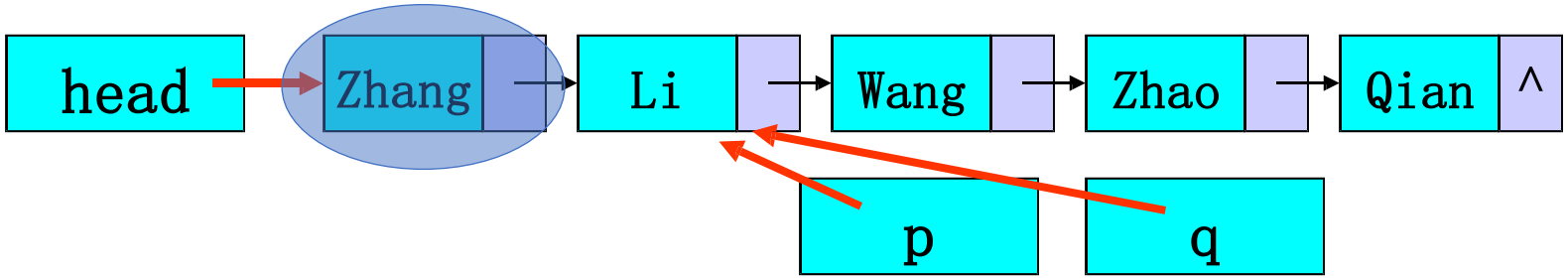
```
q = p->next;
```



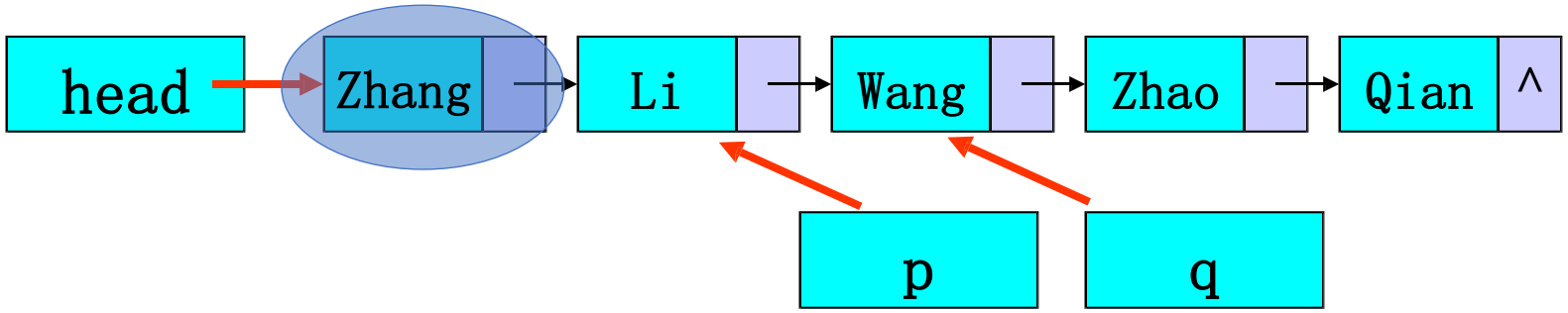
delete p;



```
p = q;
```



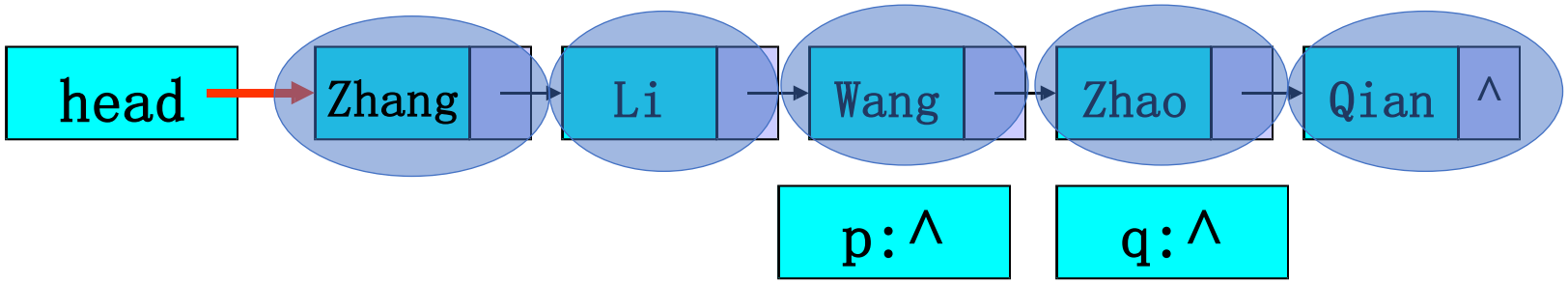
```
q = p->next;
```



```
while(p){…}
```

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



目录

- 动态内存的基本概念
- C中的相关函数
- C++中的相关运算符
- 内存的动态申请与释放
- 定位new运算符

3.2.5 定位new运算符

- new运算符的变体，指定要使用的位置
- 程序员可通过其设置内存管理规程，处理需要通过特定地址进行访问的硬件或在特定位置创建对象
- 需要包含头文件new

```
#include <new>
using namespace std;
struct chaff {
    char dross[20];
    int slag;
};
char buffer1[50];
char buffer2[500];
```

```
int main()
{ chaff *p1, *p2; int *p3, *p4;
  //the regular forms of new
    p1 = new chaff;   //place structure in heap
    p3 = new int[20]; //place int array in heap
  //the two forms of placement new
    p2 = new (buffer1) chaff;   //place structure in buffer1
    p4 = new (buffer2) int[20]; //place int array in buffer2
    …
}
```

//例8：书-程序清单9.10：三次内存分配以及其对应的地址的输出

```
#include <new>
const int BUF = 512;
char buffer[BUF]; // chunk of memory
…
int main()
{
  …
  pd1 = new double[N]; // use heap
  pd2 = new (buffer) double[N]; // use buffer array
  …
  for (i = 0; i < N; i++)
  {
      cout << pd1[i] << " at " << &pd1[i] << "; ";
      cout << pd2[i] << " at " << &pd2[i] << endl;
  }
  …接下页
```

- pd1使用的是常规的new，从堆中申请空间并返回首地址
- pd2使用的是定位new运算符，他们都是new但是参数数量上有区别，功能上也有区别，它会在指定空间（buffer）中分配N个double空间

```
Calling new and placement new:
Memory addresses:
 heap:00AACC10 static:0087D2D8
Memory contents:
1000 at 00AACC10; 1000 at 0087D2D8
1020 at 00AACC18; 1020 at 0087D2E0
1040 at 00AACC20; 1040 at 0087D2E8
1060 at 00AACC28; 1060 at 0087D2F0
1080 at 00AACC30; 1080 at 0087D2F8
```

```
…
pd3 = new double[N];          // find new address
pd4 = new (buffer) double[N];// overwrite old data
…//接下页
```

- pd3使用的是常规的new运算符，它负责从堆中申请一块空间，并返回首地址
- pd4仍然从buffer这里开始分配地址，因为定位new运算符是在传入的那个地址处分配指定大小的空间而不去管这个空间之前有没有被占用。所以会出现两次地址一样（新写入的数据会复盖之前的）

```
Calling new and placement new:
Memory addresses:
 heap:00AACC10 static:0087D2D8
Memory contents:
1000 at 00AACC10; 1000 at 0087D2D8
1020 at 00AACC18; 1020 at 0087D2E0
1040 at 00AACC20; 1040 at 0087D2E8
1060 at 00AACC28; 1060 at 0087D2F0
1080 at 00AACC30; 1080 at 0087D2F8

Calling new and placement new a second time:
Memory contents:
1000 at 00AA48C0; 1000 at 0087D2D8
1040 at 00AA48C8; 1040 at 0087D2E0
1080 at 00AA48D0; 1080 at 0087D2E8
1120 at 00AA48D8; 1120 at 0087D2F0
1160 at 00AA48E0; 1160 at 0087D2F8

Calling new and placement new a third time:
Memory contents:
1000 at 00AACC10; 1000 at 0087D300
1060 at 00AACC18; 1060 at 0087D308
1120 at 00AACC20; 1120 at 0087D310
1180 at 00AACC28; 1180 at 0087D318
1240 at 00AACC30; 1240 at 0087D320
```

```
    ...
    delete[] pd1;
    pd1 = new double[N];
    pd2 = new (buffer + N * sizeof(double)) double[N];
    ...
}
```

- 第三次的pd1和第一次的地址一样，它在new的时候重新占据了被delete之前所占据的空间
- pd2通过计算，避开了正在被使用的内存空间，这里的定位new运算符被传入了指向buffer后面的地址。所以不会复盖之前存在buffer里的数据

```
Calling new and placement new:
Memory addresses:
 heap:00AACC10 static:0087D2D8
Memory contents:
1000 at 00AACC10; 1000 at 0087D2D8
1020 at 00AACC18; 1020 at 0087D2E0
1040 at 00AACC20; 1040 at 0087D2E8
1060 at 00AACC28; 1060 at 0087D2F0
1080 at 00AACC30; 1080 at 0087D2F8

Calling new and placement new a second time:
Memory contents:
1000 at 00AA48C0; 1000 at 0087D2D8
1040 at 00AA48C8; 1040 at 0087D2E0
1080 at 00AA48D0; 1080 at 0087D2E8
1120 at 00AA48D8; 1120 at 0087D2F0
1160 at 00AA48E0; 1160 at 0087D2F8

Calling new and placement new a third time:
Memory contents:
1000 at 00AACC10; 1000 at 0087D300
1060 at 00AACC18; 1060 at 0087D308
1120 at 00AACC20; 1120 at 0087D310
1180 at 00AACC28; 1180 at 0087D318
1240 at 00AACC30; 1240 at 0087D320
```

3.2.5 定位new运算符

- 定位new运算符创建的指针，不需要delete，这不同于常规的new运算符
- 常规new调用接收一个参数的new()函数，标准定位new调用接收两个参数的new()函数：

```
int *p1 = new int;              //invokes new(sizeof(int))
int *p2 = new(buffer) int;      //invokes new(sizeof(int),buffer)
int *p3 = new(buffer) int[40]; //invokes new(40*sizeof(int),buffer)
```

- 定位new函数不可替换，但可以重载。它至少需要接收两个参数。（了解）

总结

- 动态内存的基本概念
- C中的相关函数
  - void *malloc(unsigned size);
  - void *calloc(unsigned n, unsigned size);
  - void *realloc(void *ptr, unsigned newsize);
  - void free(void *p);
- C++中的相关运算符
  - new delete
- 内存的动态申请与释放（熟练）
- 定位new运算符（了解）