§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

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）

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？

数组

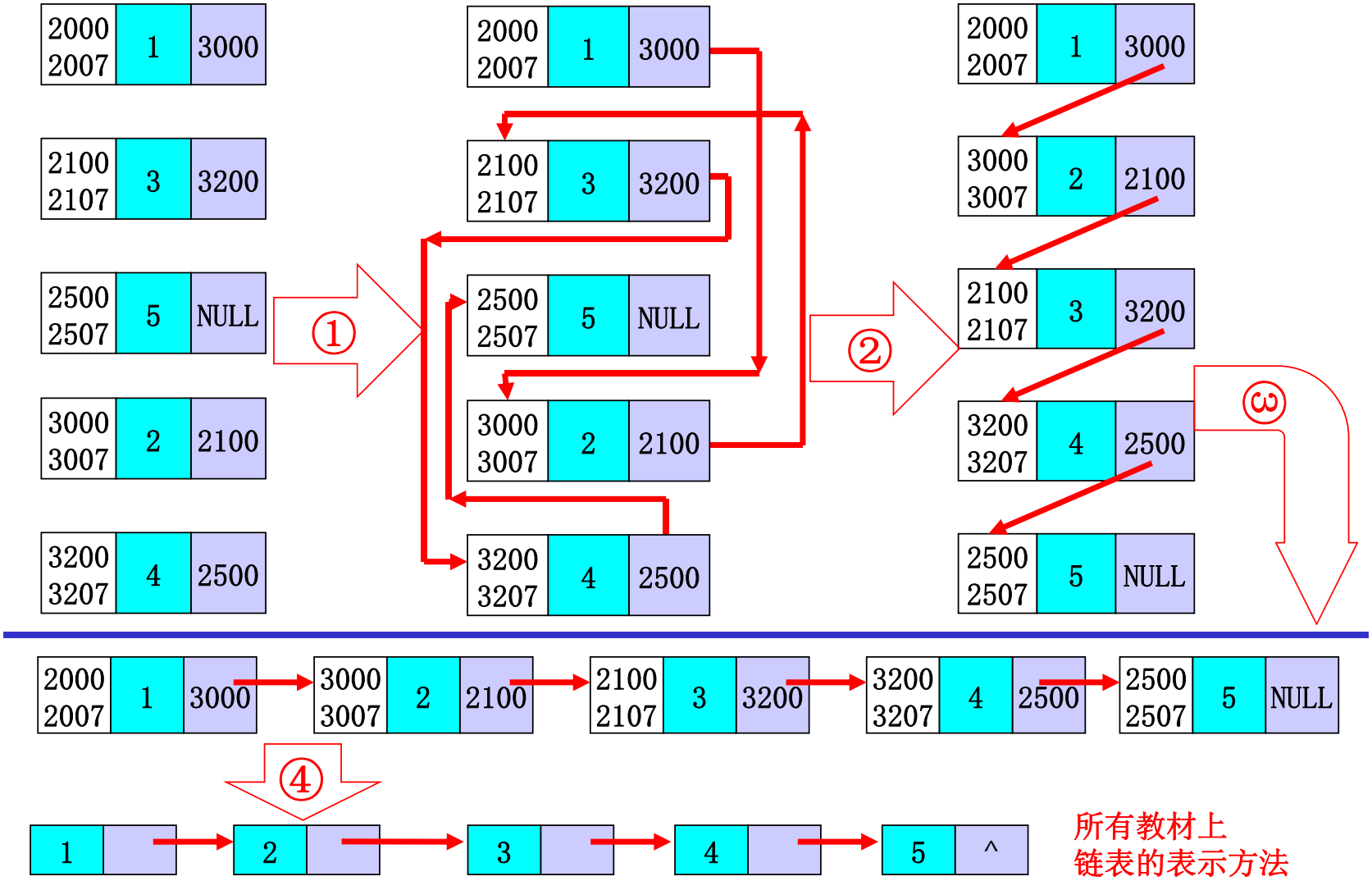
2000 2003	1
2004 2007	2
2008 2011	3
2012 2015	4
2016 2019	5

链表

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

§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

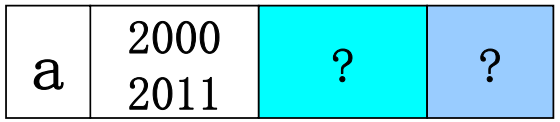
例：一个简单的静态方式链表(非链表的常规用法)

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

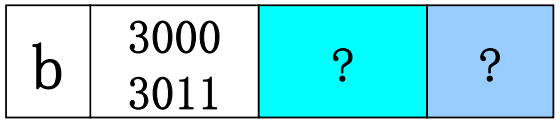
指向结构体自身的指针
成员类型不允许是自身的结构体类型，
但可以是指针(因为指针占用空间已知)

例：一个简单的静态方式链表(非链表的常规用法)

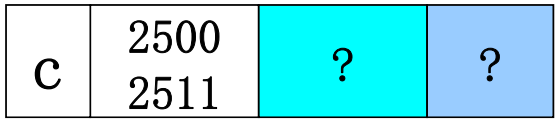
```
struct student {
    long num;
    float score;
    struct student *next;
    };
int main()
{   student a,b,c, *head, *p;
    a.num = 31001; a.score=89.5;
    b.num = 31003; b.score=90;
    c.num = 31007; c.score=85;
    head = &a;    a.next = &b;    b.next = &c;    c.next = NULL;
    p=head;
  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



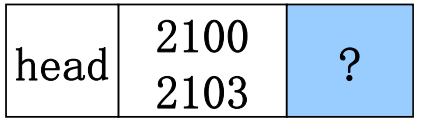
(结点)

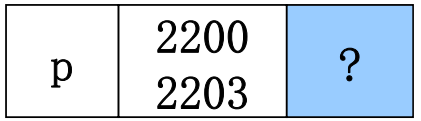


(结点)



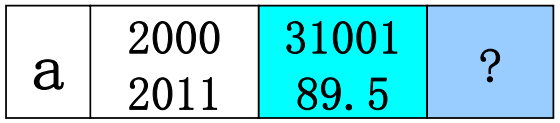
(结点)



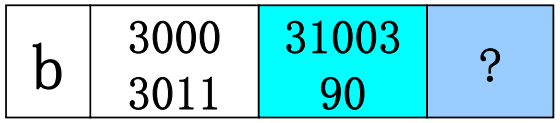


例：一个简单的静态方式链表(非链表的常规用法)

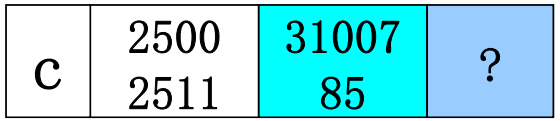
```
struct student {
    long num;
    float score;
    struct student *next;
    };
int main()
{   student a,b,c, *head, *p;
    a.num = 31001; a.score=89.5;
    b.num = 31003; b.score=90;
    c.num = 31007; c.score=85;
    head = &a;    a.next = &b;    b.next = &c;    c.next = NULL;
    p=head;
  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



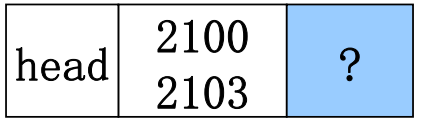
(结点)

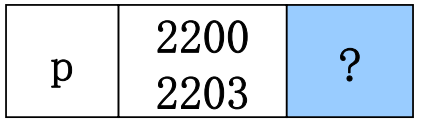


(结点)



(结点)





例：一个简单的静态方式链表(非链表的常规用法)

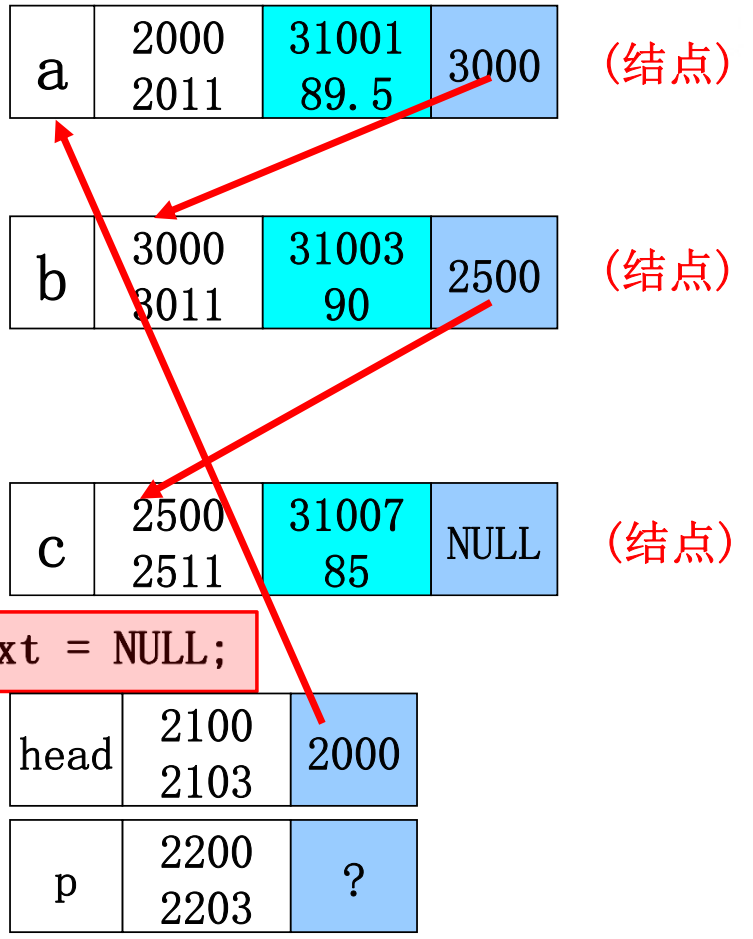
```
struct student {
    long num;
    float score;
    struct student *next;
    };
int main()
{   student a,b,c, *head, *p;
    a.num = 31001; a.score=89.5;
    b.num = 31003; b.score=90;
    c.num = 31007; c.score=85;
    head = &a;   a.next = &b;   b.next = &c;   c.next = NULL;
    p=head;
    do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

a	2000 2011	31001 89.5	3000	(结点)
b	3000 3011	31003 90	2500	(结点)
c	2500 2511	31007 85	NULL	(结点)

head	2100 2103	2000
p	2200 2203	?

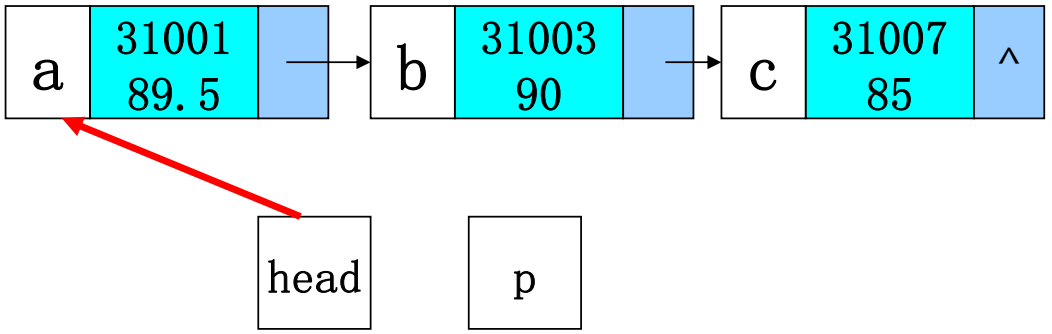
例：一个简单的静态方式链表(非链表的常规用法)

```
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);
}
```



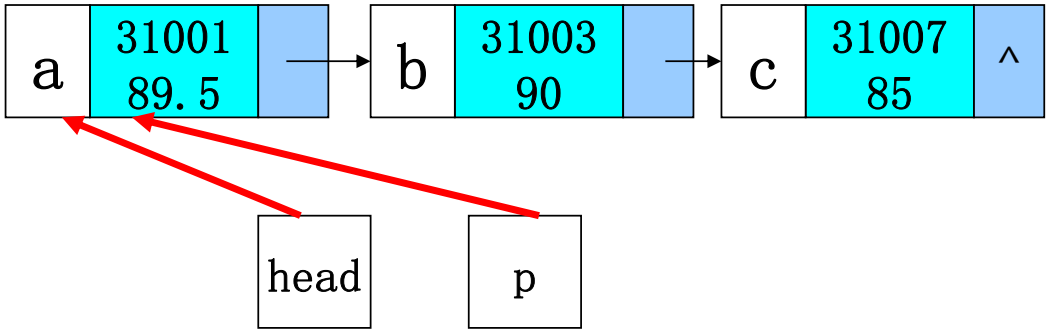
例：一个简单的静态方式链表(非链表的常规用法)

```
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);
}
```



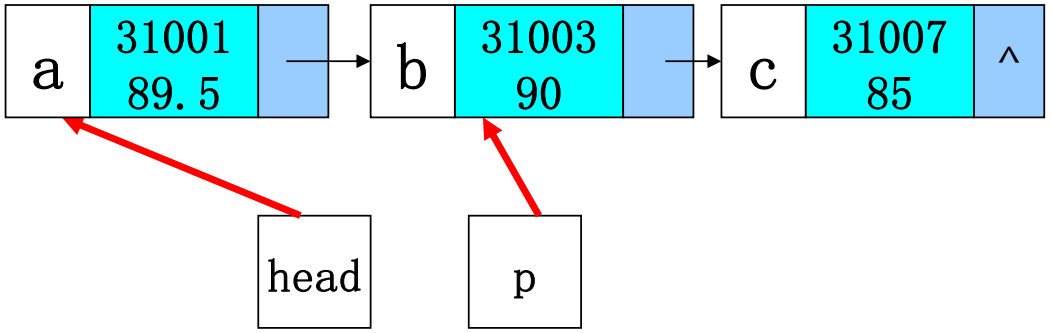
例：一个简单的静态方式链表(非链表的常规用法)

```
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);
}
```



例：一个简单的静态方式链表(非链表的常规用法)

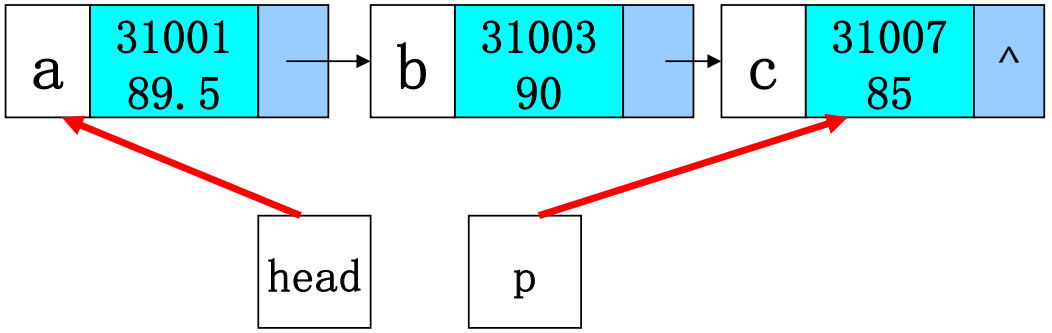
```
struct student {
    long num;
    float score;
    struct student *next;
    };
int main()
{   student a,b,c, *head, *p;
    a.num = 31001; a.score=89.5;
    b.num = 31003; b.score=90;
    c.num = 31007; c.score=85;
    head = &a;   a.next = &b;   b.next = &c;   c.next = NULL;
    p=head;
    do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

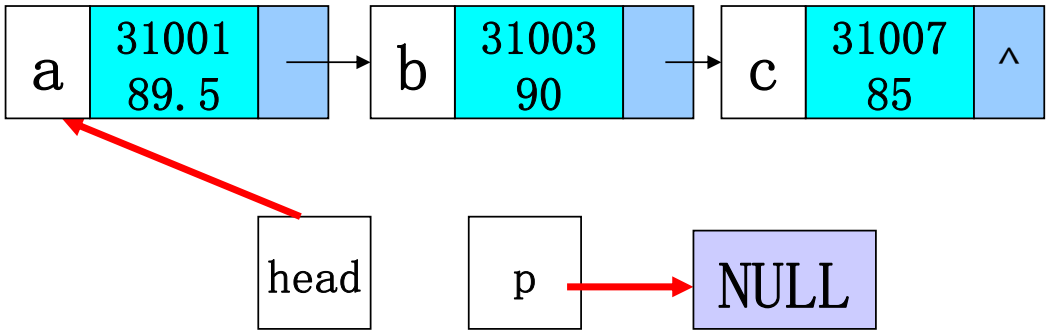
```
struct student {
    long num;
    float score;
    struct student *next;
    };
int main()
{   student a,b,c, *head, *p;
    a.num = 31001; a.score=89.5;
    b.num = 31003; b.score=90;
    c.num = 31007; c.score=85;
    head = &a;   a.next = &b;   b.next = &c;   c.next = NULL;
    p=head;
    do {
        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
	int *p;
	p = (int *)malloc(sizeof(int));
	p = (int *)calloc(1, sizeof(int));
		形式1：先定义指针变量，再申请
		int *p;
		p=new int;

	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));
		形式2：定义指针变量的同时申请
		int *p=new int;

	说明： 虽然初次申请时也可以用 p = (int *)realloc(NULL, sizeof(int)); 但一般不用	形式3：申请空间时赋初值 int *p; 或 int *p=new int(10); p=new int(10);

§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
	int *p;
	p = (int *)malloc(10*sizeof(int));
	p = (int *)calloc(10, sizeof(int));
		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

	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));
		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

	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
		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
		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

§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
	`int (*p)[4];`
	`p = (int (*)[4])malloc(3*4*sizeof(int));`
	`p = (int (*)[4])calloc(3*4, sizeof(int));`
		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!!!

	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));`
		形式2：定义指针变量的同时申请空间
		`float (*f)[4]=new float[3][4];`

	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	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 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];` `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 注：字符型在使用字符串方式初始化时，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	int *p = (int *)malloc(sizeof(int));
	int *p = (int *)calloc(1, sizeof(int));
	free(p);
		int *p = new int;
		delete p;

一维数组	int *p = (int *)malloc(10*sizeof(int));
	int *p = (int *)calloc(10, sizeof(int));
	free(p);
		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[]（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	int (*p)[4] = (int (*)[4])malloc(3*4*sizeof(int));
	int (*p)[4] = (int (*)[4])calloc(3*4, sizeof(int));
	free(p);
		int (*p)[4] = new int[3][4];
		delete []p;
		● 二维以上必须加一个[]，否则编译警告

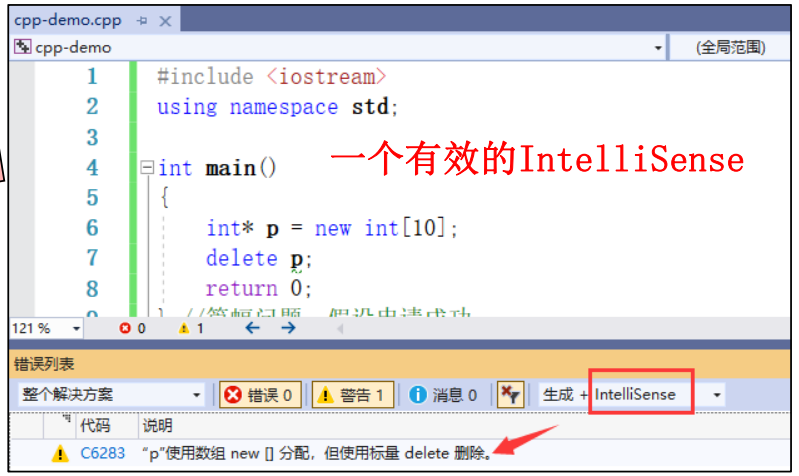
§13. 动态内存申请

3. 内存的动态申请与释放

释放对象	C++的运算符
一维数组	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[]（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
```



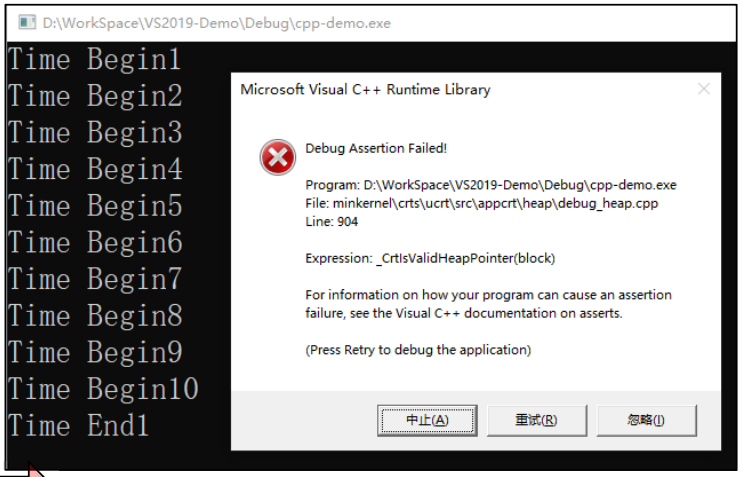
§13. 动态内存申请

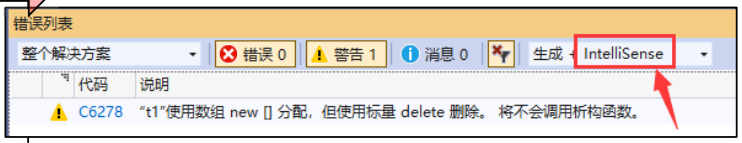
3. 内存的动态申请与释放

释放对象	C++的运算符
一维数组	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[]（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

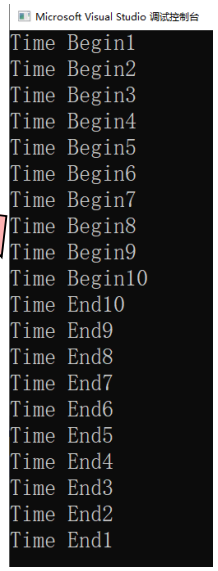
```
#include <iostream>
using namespace std;

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
```



§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

```
#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

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

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

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

```
#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;
}
```

§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

```
#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
```

嵌套申请
先student变量，再name成员

```
#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
```

★ 嵌套申请时，按定义顺序进行申请，反序进行释放

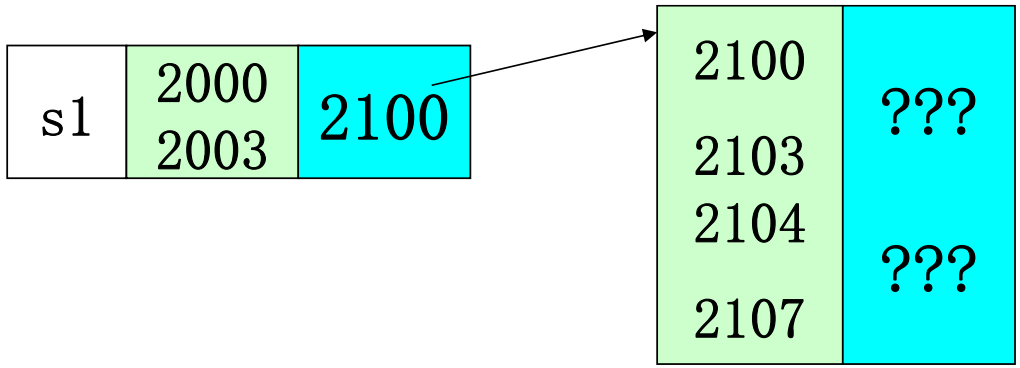
```
struct student {
    int num;
    char *name;
};
int main()
{   student *s1;
    s1 = (student *)malloc(sizeof(student)); //申请8字节
    s1->name = (char *)malloc(6*sizeof(char));//申请6字节
    s1->num = 1001;
    strcpy(s1->name, "zhang");
    cout << s1->num << ":" << s1->name << endl;
    free(s1->name);//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	???

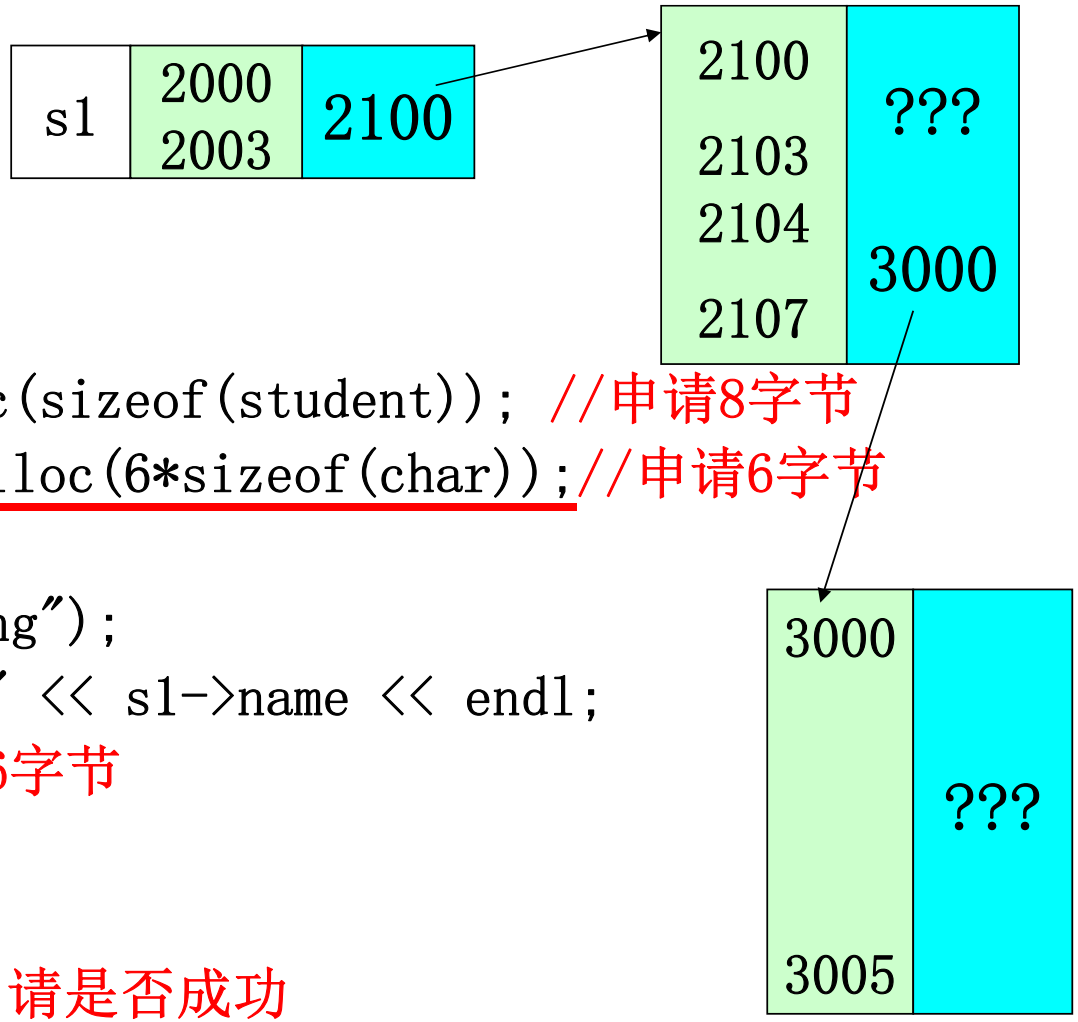
★ 嵌套申请时，按定义顺序进行申请，反序进行释放




```
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
```

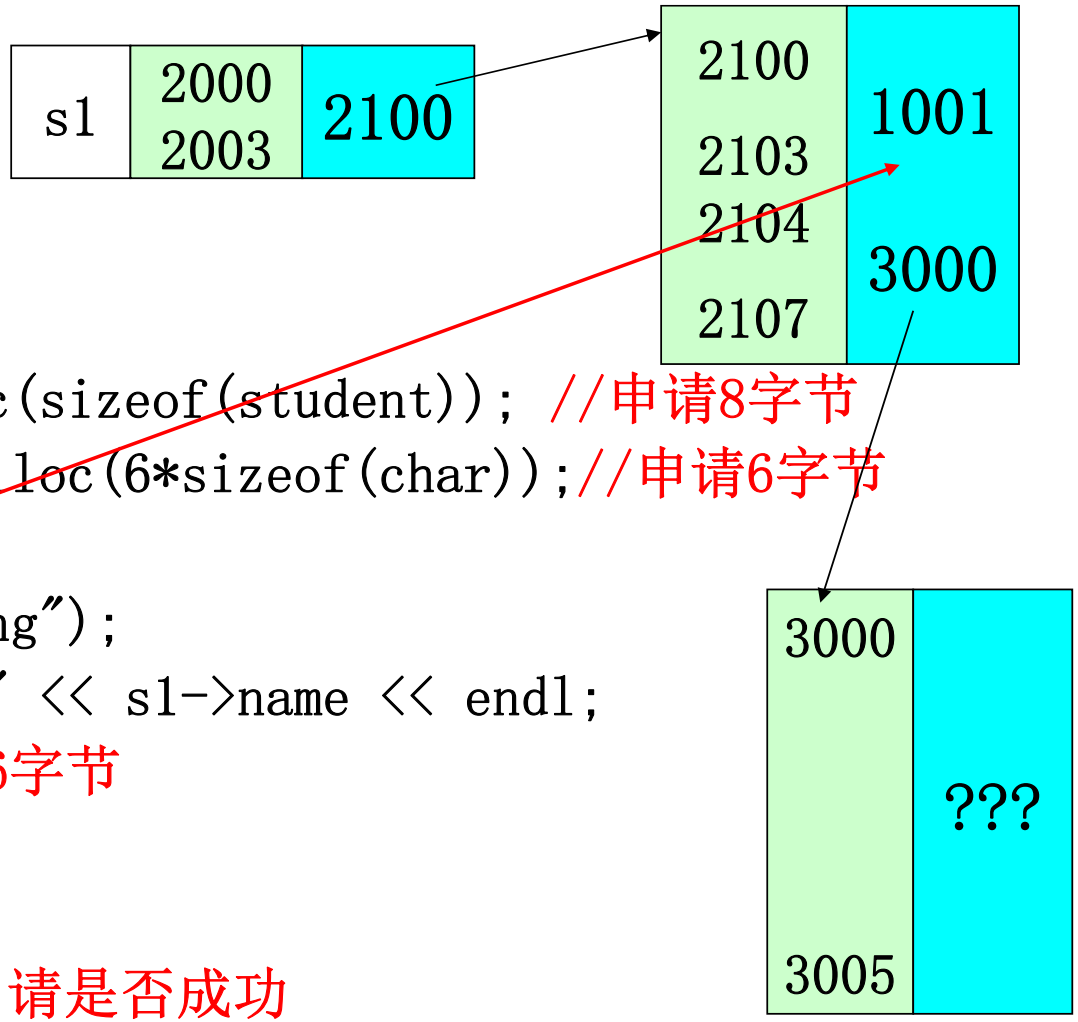
★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
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
```



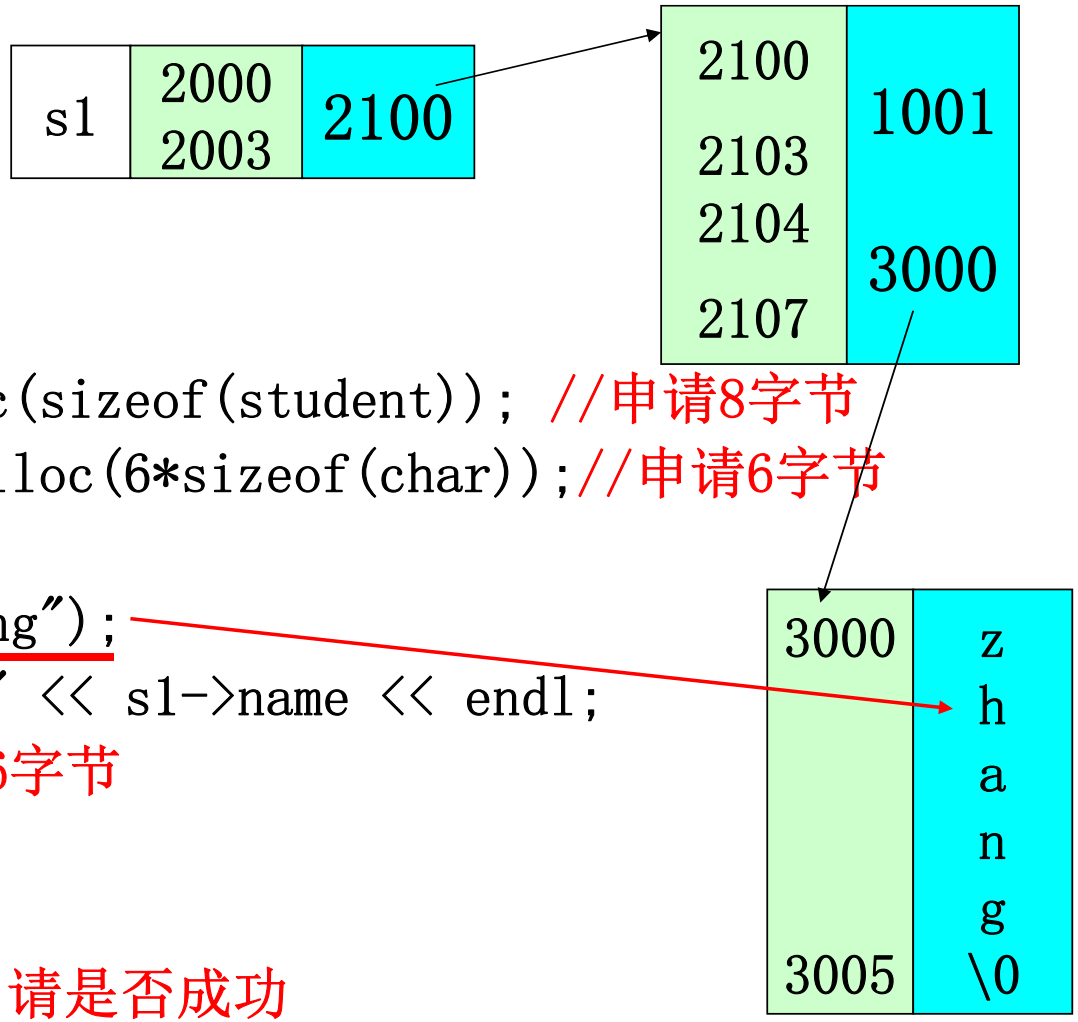
★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
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
```



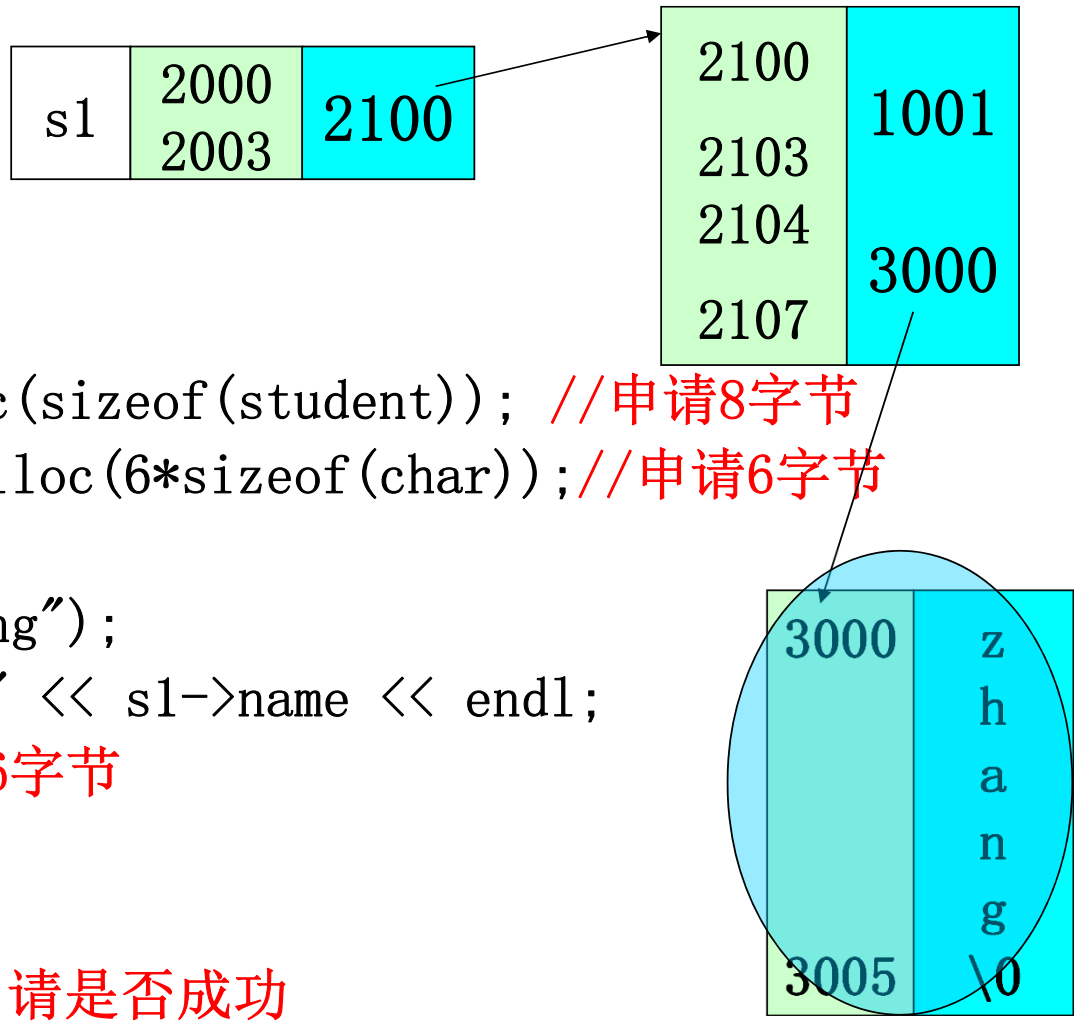
★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
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
```



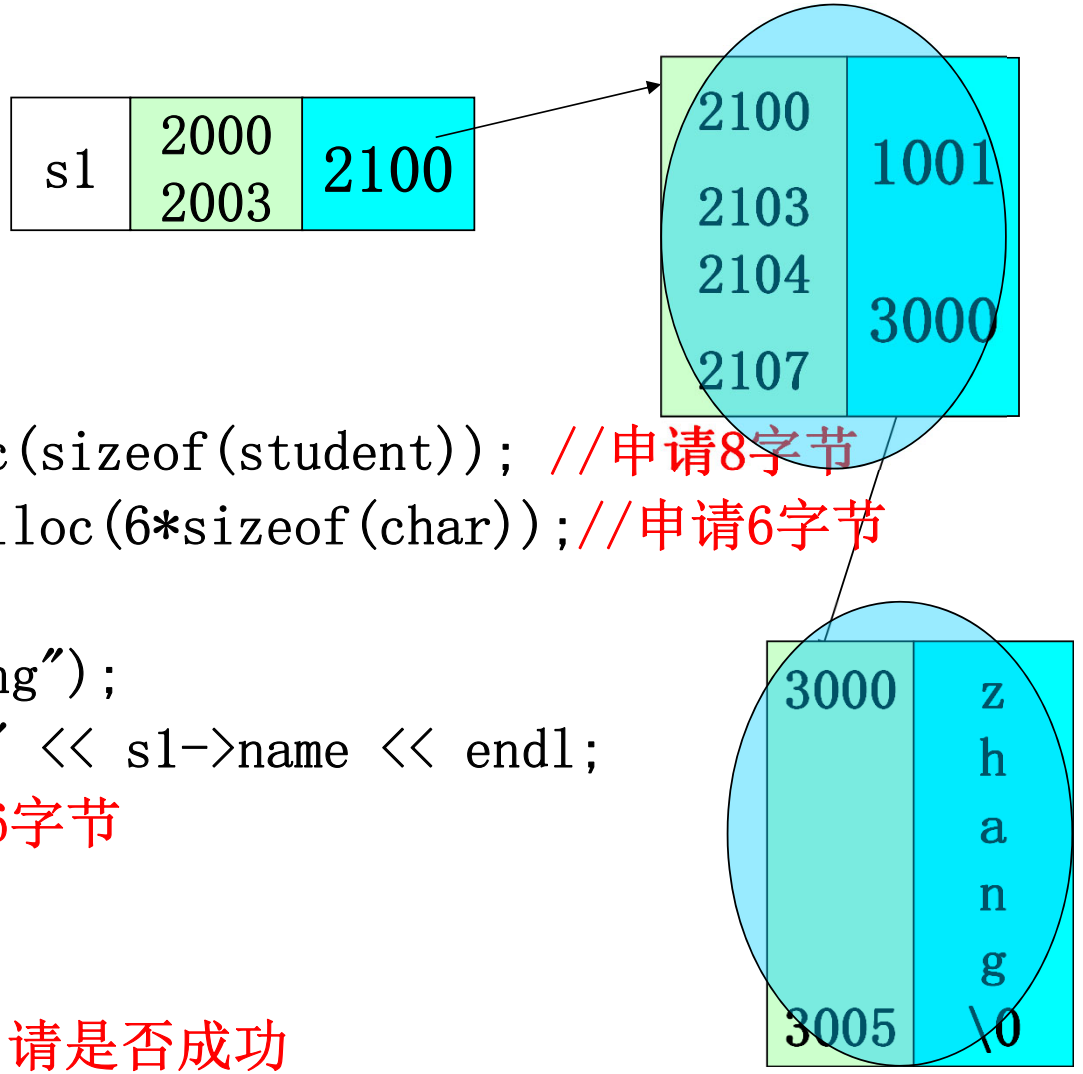
★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
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
```



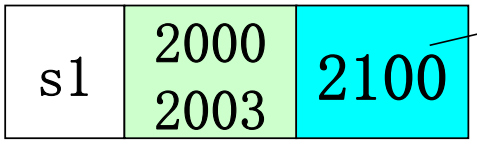
★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
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
```

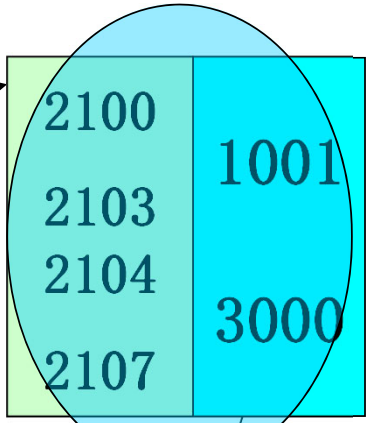


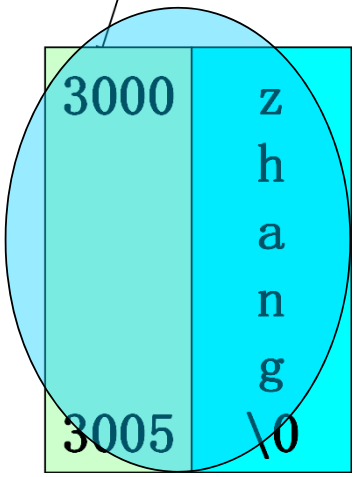
★ 嵌套申请时，按定义顺序进行申请，反序进行释放

```
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自身所占4字节
由操作系统回收





free的顺序不能反

§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;
}
```

1、多编译器观察程序的运行结果
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）

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!!!!!

```
//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

    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;
}
```

问题：为什么加U?