

§ 7. 用户自定义数据类型

7.1. 自定义数据类型的引入

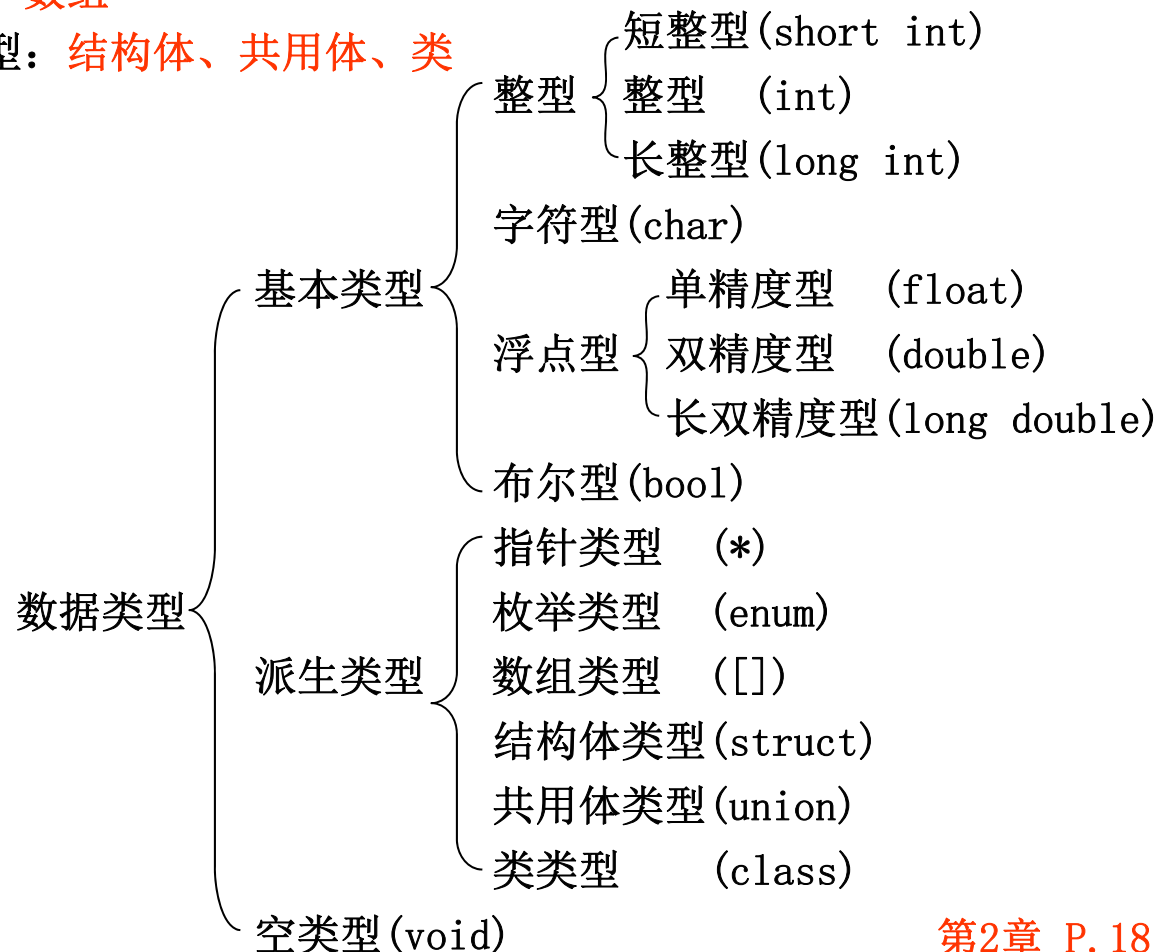
7.1.1. 自定义数据类型的含义

用**基本数据类型**以及**已存在的自定义数据类型**组合而成的新数据类型

7.1.2. 自定义数据类型的分类

元素同类型的自定义数据类型：**数组**

元素不同类型的自定义数据类型：**结构体、共用体、类**



§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.1. 引入

例1: 键盘输入学生的学号、姓名、性别、年龄、成绩和家庭住址, 再依次输出

```
int main()
{
    int num, age;
    char sex, name[20], addr[30];
    float score;
    cin >> num ... ;
    ...
    cout << sex ... ;
    return 0;
}
```

1个学生的6方面信息:
用6个彼此完全独立的
不同类型的变量来表达

缺点: 访问时无整体性

例2: 键盘输入100个学生的学号、姓名、性别、年龄、成绩和家庭住址, 再依次输出

```
const int N=100;
int main()
{
    int num[N], age[N], i;
    char sex[N], name[N][20], addr[N][30];
    float score[N];
    for(i=0; i<N; i++) {
        cin >> num[i] ... ;
        ...
        cout << sex[i] ... ;
    }
}
```

100个学生的6方面信息:
用6个彼此完全独立的不同类型的
数组变量来表达

缺点: 1. 访问时无整体性
2. 访问同一个人时, 不同数组
的下标必须对应

例3: 键盘输入学生的学号、姓名、性别、年龄、成绩和家庭住址, 再依次输出, 要求以指针方式操作

```
int main()
{
    int num, age, *p_num=&num, *p_age=&age;
    char sex, name[20], addr[30];
    char *p_sex=&sex, *p_name=name, *p_addr=addr;
    float score, *p_score=&score;
    cin >> *p_num ... ;
    ...
    cout << *p_sex ... ;
    return 0;
}
```

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.1. 引入

将描述一个事物的各方面特征的数据组合成一个有机的整体，说明数据之间的内在关系

7.2.2. 结构体类型的定义

struct 结构体名 {	struct student {
结构体成员1 (类型名 成员名)	int num;
...	char name[20];
结构体成员n (类型名 成员名)	char sex;
}; (带分号)	int age;
	float score;
	char addr[30];
	};

★ 结构体名, 成员名命名规则同变量

★ 同一结构体的成员名不能同名, 但可与其它名称 (其它结构体的成员名, 其它变量名等) 相同

struct x1 {	struct x2 {	struct x3 {
int num;	int num;	float num;
...
};	};	};
main()	fun()	void num()
{	{	{
long num;	int num[10];	...
}	}	}

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.2. 结构体类型的定义

- ★ 结构体名, 成员名命名规则同变量
- ★ 同一结构体的成员名不能同名, 但可与其它名称 (其它结构体的成员名, 其它变量名等) 相同
- ★ 每个成员的类型可以相同, 也可以不同
- ★ 每个成员的类型既可以是基本数据类型, 也可以是已存在的自定义数据类型

```
struct date {  
    int year;  
    int month;  
    int day;  
};  
  
struct student {  
    int num;  
    char name[20];  
    char sex;  
    struct date birthday;  
    float score;  
    char addr[30];  
};
```

struct date必须在struct student的前面定义, 否则无法知道birthday占多少字节

```
struct student {  
    int num;  
    char name[20];  
    char sex;  
    struct student monitor;  
    float score;  
    char addr[30];  
};
```

★ 每个成员的类型不允许是自身的结构体类型

无法判断 monitor 占多少个字节

- ★ 每个成员的类型不允许是自身的结构体类型
- ★ 结构体类型的定义既可以放在函数外部, 也可以放在函数内部 (具体定义及访问规则见 7.2.8)
- ★ 结构体类型的大小为所有成员的大小的总和, 可用sizeof(struct 结构体名) 计算, 但不占用具体的内存空间 (结构体变量占用一段连续的内存空间)

int i; sizeof(int)得4
但int型不占空间, i占4字节

§ 7. 用户自定义数据类型

7.2. 结构体类型

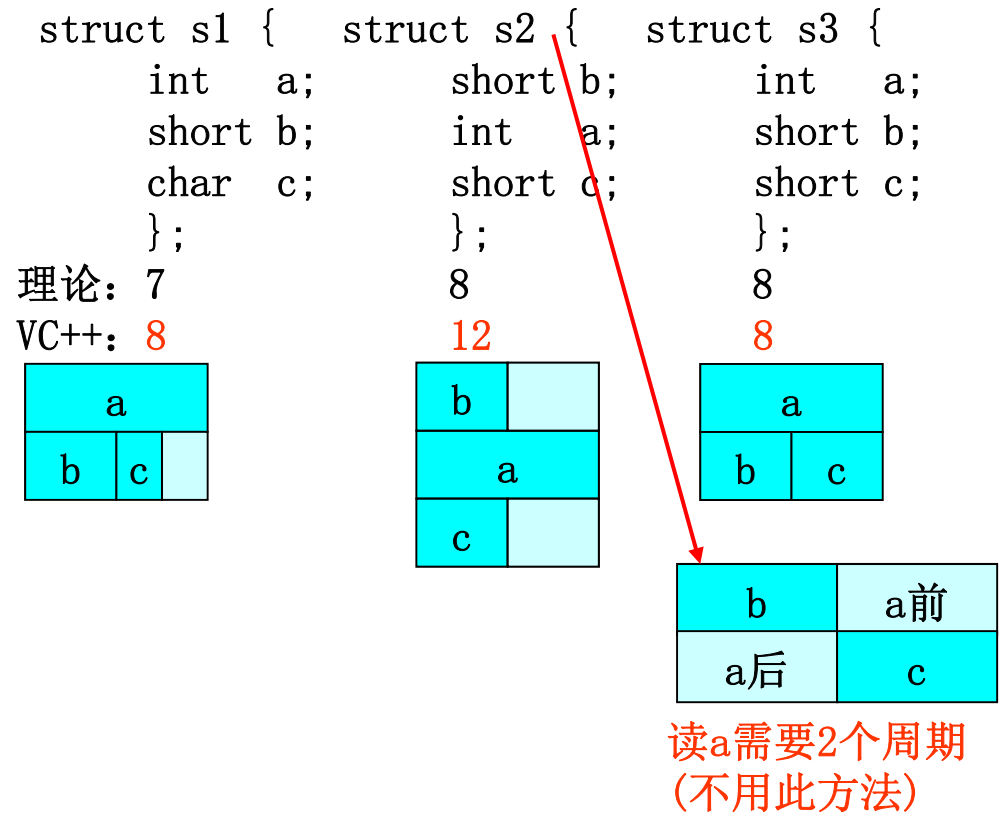
7.2.2. 结构体类型的定义

★ 在不同的编译系统中, 有时为了加快程序运行速度, 采用按**数据总线宽度**对齐的方法来计算结构体类型的大小, 可能出现填充字节

=>此概念需了解, 本书不继续讨论, 仍按无填充计算

P.196 注释① sizeof(student)结果为 64 => 68

```
#include <iostream>
using namespace std;
struct s1 {
    int a;
    short b;
    char c;
};
struct s2 {
    short b;
    int a;
    short c;
};
struct s3 {
    int a;
    short b;
    short c;
};
void main()
{
    cout << sizeof(s1) << endl; 8
    cout << sizeof(s2) << endl; 12
    cout << sizeof(s3) << endl; 8
}
```



§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.2. 结构体类型的定义

- ★ 结构体名, 成员名命名规则同变量
- ★ 同一结构体的成员名不能同名, 但可与其它名称 (其它结构体的成员名, 其它变量名等) 相同
- ★ 每个成员的类型可以相同, 也可以不同
- ★ 每个成员的类型既可以是基本数据类型, 也可以是已存在的自定义数据类型
- ★ 每个成员的类型不允许是自身的结构体类型
- ★ 结构体类型的定义既可以放在函数外部, 也可以放在函数内部 (具体定义及访问规则见 7.2.10)
- ★ 结构体类型的大小为所有成员的大小的总和, 可用sizeof(struct 结构体名) 计算, 但不占用具体的内存空间 (结构体变量占用一段连续的内存空间)
- ★ 在不同的编译系统中, 有时为了加快程序运行速度, 采用按数据总线宽度对齐的方法来计算结构体类型的大小, 可能出现填充字节 (需了解, 本书不讨论)
- ★ C的结构体只能包含数据成员, C++还可以包含函数

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.3. 结构体变量的定义及初始化

7.2.3.1. 先定义结构体类型，再定义变量

```
struct student {  
    ...  
};  
struct student s1;  
struct student s2[10];  
struct student *s3;
```

★ struct在C中不能省，在C++中可省略

★ 结构体变量占用实际的内存空间，根据变量的不同类型(静态/动态/全局/局部)在不同区域进行分配

7.2.3.2. 在定义结构体类型的同时定义变量

```
struct student {  
    ...  
} s1, s2[10], *s3;  
struct student s4;
```

★ 可以再次用7.2.3.1的方法定义新的变量(struct在C++中可省)

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.3. 结构体变量的定义及初始化

7.2.3.3. 直接定义结构体类型的变量 (结构体无名)

```
struct {  
    ...  
} s1, s2[10], *s3;
```

- ★ 因为结构体无名，因此无法再用7.2.3.1的方法进行新的变量定义
(适用于仅需要一次性定义的地方)

7.2.3.4. 结构体变量定义时初始化

```
student s1={1, "张三", 'M', 20, 78.5, "上海"};
```

- ★ 按各成员依次列出

- ★ 若嵌套使用，要列出最低级成员

```
student s1={1, "张三", 'M', {1982, 5, 9}, 78.5};
```

- ★ 可用一个同类型变量初始化另一个变量

```
student s1={1, "张三", 'M', {1982, 5, 9}, 78.5};  
student s2=s1;
```

```
struct student {  
    int    num;  
    char   name[20];  
    char   sex;  
    int    age;  
    float  score;  
    char   addr[30];  
};
```

内{}可省
但不建议

```
struct date {  
    int year;  
    int month;  
    int day;  
};
```

```
struct student {  
    int    num;  
    char   name[20];  
    char   sex;  
    struct date birthday;  
    float  score;  
};
```

内{}可省
但不建议

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.4. 结构体变量的使用

★ P. 198 7.1.3 结构体变量的引用（与“引用”无关）

7.2.4.1. 形式

变量名. 成员名

★ P. 199 说明(5)中：“由于. 运算符的优先级最高” => C中最高，C++中次高

```
struct student {  
    int    num;          s1.num = 1;  
    char   name[20];     strcpy(s1.name, "张三");  
    char   sex;          s1.sex = 'M';  
    int    age;          s1.age = 20;  
    float  score;        s1.score = 76.5;  
    char   addr[30];     strcpy(s1.addr, "上海");  
} s1;
```

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.4. 结构体变量的使用

7.2.4.1. 形式

7.2.4.2. 使用

★ 结构体变量允许进行整体赋值操作

```
student s1={...}, s2;
```

```
s2=s1; //赋值语句
```

用一个同类型变量初始化另一个变量:

```
student s1={...}, s2=s1; //定义时初始化
```

★ 在所有基本类型变量出现的地方，均可以使用该基本类型的结构体变量的成员

<code>int i, *p;</code>	<code>student s1; int *p;</code>	
<code>i++;</code>	<code>s1.num++;</code>	自增/减
<code>... + i*10 +...;</code>	<code>... + s1.num*10 +...;</code>	各种表达式
<code>if (i>=10)</code>	<code>if (s1.num>=10)</code>	
<code>p = &i;</code>	<code>p = &s1.num;</code>	取地址
<code>scanf("%d", &i);</code>	<code>scanf("%d", &s1.num);</code>	输入
<code>cout << i;</code>	<code>cout << s1.num;</code>	输出
<code>fun(i);</code>	<code>fun(s1.num);</code>	函数实参
<code>return i;</code>	<code>return s1.num;</code>	返回值

§ 7. 用户自定义数据类型

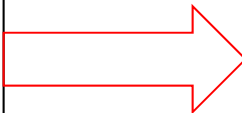
7.2. 结构体类型

7.2.4. 结构体变量的使用

7.2.4.2. 使用

- ★ 结构体变量允许进行整体赋值操作
- ★ 在所有基本类型变量出现的地方，均可以使用该基本类型的结构体变量的成员
- ★ 若嵌套使用，只能对最低级成员操作

```
struct date {  
    int year;  
    int month;  
    int day;  
};  
struct student {  
    int num;  
    char name[9];  
    char sex;  
    struct date birthday ;  
    float score;  
};
```



```
s1.birthday.year=1980;  
cin >> s1.birthday.month;  
cout << s1.birthday.day;
```

- ★ 结构体变量不能进行整体的输入和输出操作

```
student s1={...};  
cin >> s1;    ✖  
cout << s1;   ✖
```

§ 7. 用户自定义数据类型

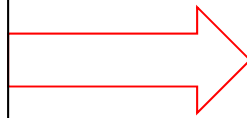
7.2. 结构体类型

7.2.4. 结构体变量的使用

7.2.4.2. 使用

例1：键盘输入学生的学号、姓名、性别、年龄、成绩和家庭住址，再依次输出 (前面例子的对比)

```
int main()
{
    int num;
    int age;
    char sex;
    char name[20];
    char addr[30];
    float score;
    cin >> num ... ;
    ...
    cout << sex ... ;
    return 0;
}
```



```
struct student {
    ...;
};
int main()
{
    struct student s1;

    cin >> s1.num ... ;
    ...
    cout << s1.sex ... ;
    return 0;
}
```

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.5. 结构体变量数组

7.2.5.1. 含义

一个数组，数组中的元素是结构体类型

7.2.5.2. 定义

~~struct~~ 结构体名 数组名[正整常量表达式] 包括整型常量、整型符号常量和整型只读变量

~~struct~~ 结构体名 数组名[正整常量1][正整常量2]

~~struct~~ student s2[10];

~~struct~~ student s4[10][20];

7.2.5.3. 定义时初始化

```
struct student s2[10] = { {1, "张三", 'M', 20, 78.5, "上海"},  
                          {2, "李四", 'F', 19, 82, "北京"},  
                          {..}, {..}, {..}, {..}, {..}, {..}, {..}, {..} };
```

内{}可省
但不建议

★ 其它同基本数据类型数组的初始化

(占用空间、存放、下标范围、初始化时省略大小)

7.2.5.4. 使用

数组名[下标].成员名

s2[0].num=1;

cin >> s2[0].age >> s2[0].name;

cout << s2[1].age << s2[1].name;

s2[2].name[0] = 'A'; //注意两个[]的位置

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.5. 结构体变量数组

7.2.5.1. 含义

7.2.5.2. 定义

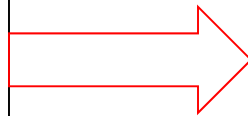
7.2.5.3. 定义时初始化

7.2.5.4. 使用

例2：键盘输入100个学生的学号、姓名、性别、年龄、成绩和家庭住址，再依次输出(前面例子对比)

```
const int N=100;

int main()
{   int num[N], age[N], i;
    char sex[N];
    char name[N][20];
    char addr[N][30];
    float score[N];
    for(i=0; i<N; i++) {
        cin >> num[i] ... ;
        ...
        cout << sex[i] ...;
    }
}
```



```
const int N=100;
struct student {
    ...;
};
int main()
{   int i;
    struct student s2[N];
    for(i=0; i<N; i++) {
        cin >> s2[i].num ... ;
        ...
        cout << s2[i].sex ...;
    }
    return 0;
}
```

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.5. 结构体变量数组

7.2.5.4. 使用

P. 202 例7.2

```
struct Person {
    char name[20];
    int count;
};

int main()
{
    Person leader[3]={...};
    int i, j;
    char leader_name[20];
    for(i=0; i<10; i++) {
        cin >> leader_name;
        for(j=0; j<3; j++)
            if (!strcmp(leader_name, leader[j].name))
                leader[j].count++;
    }
    cout << endl;
    for(i=0; i<3; i++)
        cout << leader[i].name << ":" <<
            leader[i].count << endl;
    return 0;
}
```

if(!strcmp(leader_name, leader[j].name)){
leader[j].count++;
break;
} 运行效率高, 避免不必要的比较

P. 203 例7.2的变化

```
struct Person {
    string name;
    int count;
};

int main()
{
    Person leader[3]={...};
    int i, j;
    string leader_name;
    for(i=0; i<10; i++) {
        cin >> leader_name;
        for(j=0; j<3; j++)
            if (leader_name == leader[j].name)
                leader[j].count++;
    }
    cout << endl;
    for(i=0; i<3; i++)
        cout << leader[i].name << ":" <<
            leader[i].count << endl;
    return 0;
}
```

if (leader_name == leader[j].name) {
leader[j].count++;
break;
} 运行效率高, 避免不必要的比较

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.6. 指向结构体变量的指针

含义：存放结构体变量的地址

7.2.6.1. 结构体变量的地址与结构体变量中成员地址

student s1;

&s1 : 结构体变量的地址

(基类型是结构体变量, +1表示一个结构体)

&s1.age : 结构体变量中某个成员地址

(基类型是该成员的类型, +1表示一个成员)

7.2.6.2. 结构体指针变量的定义

struct 结构体名 *指针变量名

struct student s1, *s3;

int *p;

s3=&s1; 结构体变量的指针

s3的值为2000, ++s3后值为2063

p=&s1.age; 结构体变量成员的指针

p的值为2025, ++p后值为2029

```
struct student s1;  
&s1  
&s1.age
```

s1	2000 2003	num
	2004 ... 2023	name
	2024	sex
	2025 2028	age
	2029 2032	score
	2033 ... 2062	addr

```
#include <iostream>
#include <iomanip>
using namespace std;
struct student {
    int    num;
    char  name[20];
    char  sex;
    int    age;
    float score;
    char  addr[30];
};

void main()
{
    struct student s1;
    cout << &s1 << endl;           地址X
    cout << &s1+1 << endl;         地址X + 68

    cout << &s1.num << endl;       地址X
    cout << &s1.num+1 << endl;     地址X + 4

    cout << hex <<int(&s1.sex) << endl;  地址Y (X + 24)
    cout << hex <<int(&s1.sex+1)<<endl;  地址Y + 1

    cout << &s1.age << endl;       地址Z (Y+1 + 3)
    cout << &s1.age+1 << endl;     地址Z + 4
}
```


§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.6. 指向结构体变量的指针

7.2.6.1. 结构体变量的地址与结构体变量中成员地址

7.2.6.2. 结构体指针变量的定义

7.2.6.3. 使用

(*指针变量名). 成员名

指针变量名->成员名 ⇔ (*指针变量名). 成员名

```
struct student s1, *s3=&s1;  
cout << s1.num    << s1.name    << s1.sex;  
cout << (*s3).num << (*s3).name << (*s3).sex;  
cout << s3->num    << s3->name    << s3->sex;
```

s3->age++; 值后缀++

++s3->age; 值前缀++

§ 7. 用户自定义数据类型

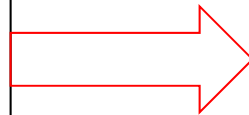
7.2. 结构体类型

7.2.6. 指向结构体变量的指针

7.2.6.3. 使用

例3：键盘输入学生的学号、姓名、性别、年龄、成绩和家庭住址，再依次输出，要求以指针方式操作 (前例)

```
int main()
{  int num, age;
   char sex, name[20], addr[30];
   float score;
   int *p_num=&num;
   int *p_age=&age;
   char *p_sex=&sex;
   char *p_name=name;
   char *p_addr=addr;
   float *p_score=&score;
   cin >> *p_num ... ;
   ...
   cout << *p_sex ...;
   return 0;
}
```



```
struct student {
    ...;
};

int main()
{  struct student s1;
   struct student *s3;
   s3 = &s1;
   cin >> s3->num ... ;
   ...
   cout << s3->sex ...;
   return 0;
}
```

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.6. 指向结构体变量的指针

7.2.6.4. 指向结构体数组的指针

```
struct student s2[10], *p;
```

```
p=s2; ✓
```

```
p=&s2[0]; ✓
```

```
p=&s2[0].num; ✗ 类型不匹配
```

```
p=(struct student *)&s2[0].num; ✓ 强制类型转换
```

与第6章中

```
int a[3][4], *p;
```

```
p=a; //编译报错
```

原因相同

各种表示形式:

`(*p).num` : 取p所指元素中成员num的**值**

`p->num` : ...

`p[0].num` : ...

`p+1` : 取p指元素的下一个元素的**地址**

`(*p+1).num`: 取p指向的元素的下一个元素的num**值**

`(p+1)->num` : ...

`p[1].num` : ...

`(p++)->num` : 先取p所指元素的成员num的值, p再指向下一个元素

`(++p)->num` : p先指向下一个元素, 再取p所指元素的成员num的值

`p->num++` : 取p所指元素中成员num的值, 值++

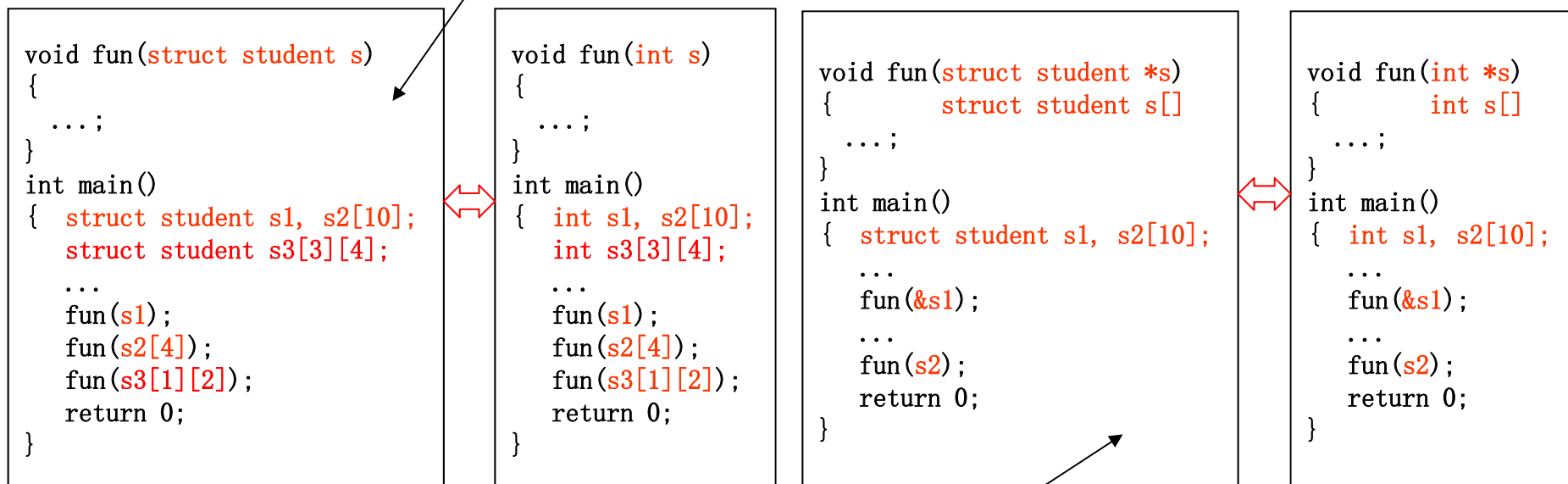
§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.7. 结构体数据类型作为函数参数

7.2.7.1. 形参为结构体简单变量

★ 对应实参为结构体简单变量/数组元素



7.2.7.2. 形参为结构体变量的指针

★ 对应实参为结构体简单变量的地址/一维数组名

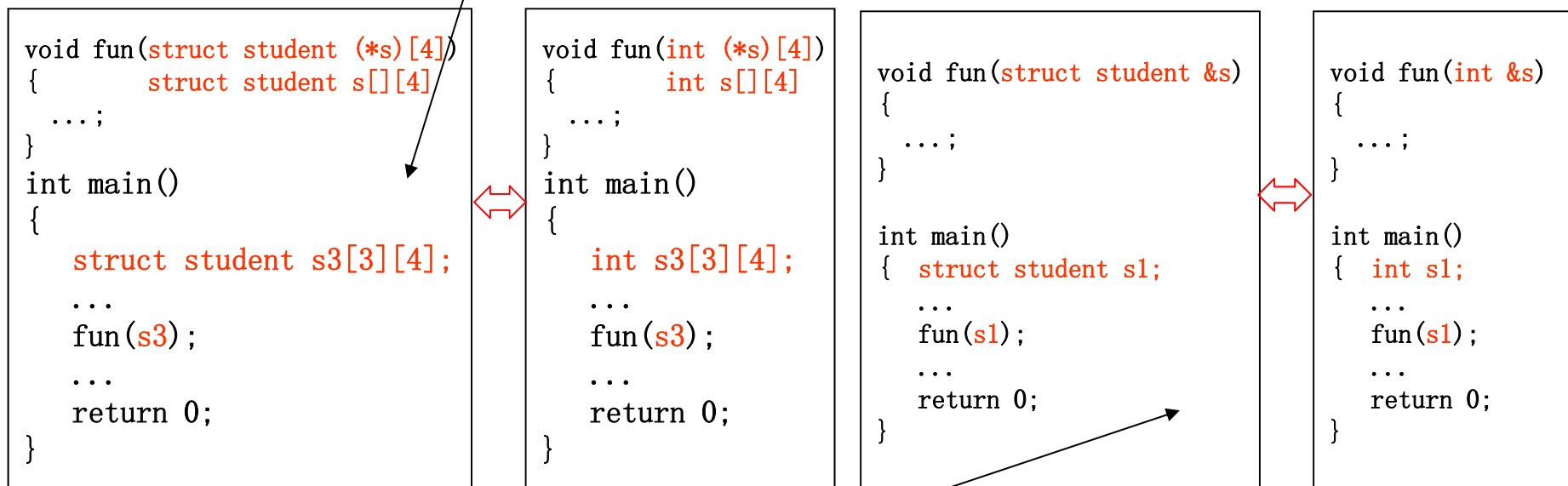
§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.7. 结构体数据类型作为函数参数

7.2.7.3. 形参为结构体数组的指针

★ 对应实参为结构体二维数组名



7.2.7.4. 形参为结构体的引用声明

★ 对应实参为结构体简单变量

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.8. 结构体类型在不同位置定义时的使用 (补7.2.2)

★ 结构体类型的定义既可以放在函数外部, 也可以放在函数内部 (具体定义及访问规则见 7.2.8)

- ★ 函数内部:
 - 可定义结构体类型的各种变量/成员级访问
- ★ 函数外部:
 - 从定义点到本源程序文件的结束前:
 - 可定义结构体类型的各种变量/成员级访问
 - 其它位置 (本源程序定义点前/其它源程序):
 - 有该结构体的提前声明:
 - 仅可定义指针及引用/整体访问
 - 有该结构体的重复定义:
 - 可定义结构体类型的各种变量/成员级访问

类似外部全局变量概念, 但不完全相同

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.8. 结构体类型在不同位置定义时的使用 (补7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部(具体定义及访问规则见 7.2.8)

情况一: 定义在函数内部

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

void fun(void)
{ struct student {
    int num;
    char name[20];
    char sex;
    int age;
    float score;
};

    struct student s1, s2[10], *s3;
    s1.num = 10;
    s2[4].age = 15;
    s3 = &s1;
    s3->score = 75;
    s3 = s2;
    (s3+3)->age = 15;
}
```

正确

```
int main()
{
    struct student s;
    s.age = 15;

    return 0;
}
```

不正确

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.8. 结构体类型在不同位置定义时的使用 (补7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部(具体定义及访问规则见 7.2.8)

情况二: 定义在函数外部,从定义点到本源程序结束前

```
#include <iostream>
using namespace std;
struct student {
    int  num;
    char name[20];
    char sex;
    int age;
    float score;
};
void f1(void)
{
    struct student s1, s2[10], *s3;
    s1.num = 10;
    s2[4].age = 15;
    s3 = &s1;
    s3->score = 75;
    s3 = s2;
    (s3+3)->age = 15;
}
```

都正确

```
void f2(struct student *s)
{
    s->age = 15;
}
struct student f3(void)
{
    struct student s;
    ...
    return s;
}
int main()
{
    struct student s1, s2;
    f1();
    f2(&s1);
    s2 = f3();
    return 0;
}
```


§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.8. 结构体类型在不同位置定义时的使用 (补7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部(具体定义及访问规则见 7.2.8)

情况三: ex1.cpp和ex2.cpp构成一个程序, 无提前声明

<pre>/* ex1.cpp */ #include <iostream> using namespace std; void f1() { 不可定义/使用student型各种变量 ✕ } struct student { ...; }; int fun() { 可定义student型各种变量, 访问成员 ✓ } int main() { 可定义student型各种变量, 访问成员 ✓ }</pre>	<pre>/* ex2.cpp */ #include <iostream> using namespace std; int f2() { 不可定义/使用student型各种变量 ✕ }</pre>
---	--

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.8. 结构体类型在不同位置定义时的使用 (补7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部(具体定义及访问规则见 7.2.8)

情况四: ex1.cpp和ex2.cpp构成一个程序, 有提前声明

```
/* ex1.cpp */
#include <iostream>
using namespace std;
struct student; //结构体声明
void f1(struct student *s1)
{
    s1->age;
}
void f2(struct student &s2)
{
    s2.score;
}
struct student {
    ...;
};
int main()
{
    可定义student型各种变量, 访问成员
}
```

允许

不允许

```
/* ex2.cpp */
#include <iostream>
using namespace std;
struct student; //结构体声明

void f2()
{
    struct student *s1;
    struct student s3, &s2=s3;
    s1.age = 15;
}
```

允许

不允许

虽可定义指针/引用,但不能进行成员级访问,无意义

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.8. 结构体类型在不同位置定义时的使用 (补7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部(具体定义及访问规则见 7.2.8)

情况四(变化1): ex1.cpp和ex2.cpp构成一个程序, 有提前声明

```
/* ex1.cpp */
#include <iostream>
using namespace std;
struct student; //结构体声明
void f1(struct student *s1)
{
    s1->age;
}
void f2(struct student &s2)
{
    s2.score;
}
struct student {
    ...;
};
int main()
{
    可定义student型各种变量, 访问成员
}
```

允许

不允许

```
/* ex2.cpp */
#include <iostream>
using namespace std;

void f2()
{
    struct student *s1;
}

void f3()
{
    struct student; //结构体声明

    struct student *s1;
    s1->age = 15;
}
```

不允许

允许

虽可定义指针/引用,但不能进行成员级访问,无意义

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.8. 结构体类型在不同位置定义时的使用 (补7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部(具体定义及访问规则见 7.2.8)

情况五: ex1.cpp和ex2.cpp构成一个程序, 有重复定义

```
/* ex1.cpp */
#include <iostream>
using namespace std;

struct student { //结构体定义
    ...;
};

int fun()
{
    可定义/使用student型各种变量 ✓
}

int main()
{
    可定义/使用student型各种变量 ✓
}
```

```
/* ex2.cpp */
#include <iostream>
using namespace std;

struct student { //结构体定义
    ...;
};

int f2()
{
    可定义/使用student型各种变量 ✓
}
```

本质上是两个不同的结构体
struct student, 因此即使
不完全相同也能正确, 这样
会带来理解上的偏差

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.8. 结构体类型在不同位置定义时的使用 (补7.2.2)

★ 结构体类型的定义既可以放在函数外部,也可以放在函数内部(具体定义及访问规则见 7.2.8)

问题: 如何在其它位置访问定义和使用结构体?

```
/* ex.h */  
struct student { //结构体定义  
    ...;  
};
```

```
/* ex1.cpp */  
#include <iostream>  
#include "ex.h" ←  
using namespace std;  
  
int fun()  
{  
    可定义/使用student型各种变量 ✓  
}  
  
int main()  
{  
    可定义/使用student型各种变量 ✓  
}
```

```
/* ex2.cpp */  
#include <iostream>  
#include "ex.h" ←  
using namespace std;  
  
int f2()  
{  
    可定义/使用student型各种变量 ✓  
}
```

解决方法: 在头文件中定义

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.9. 指向结构体变量的指针与链表

7.2.9.1. 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)

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

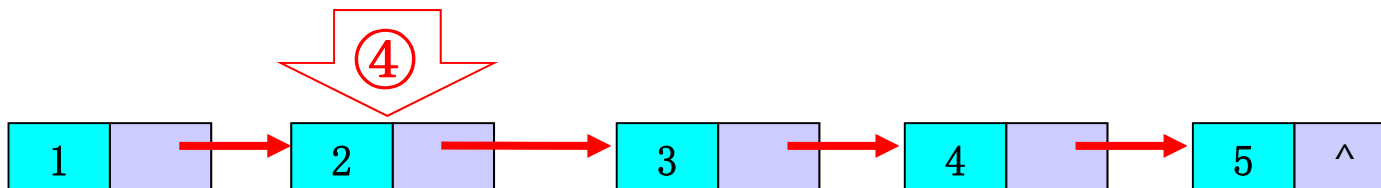
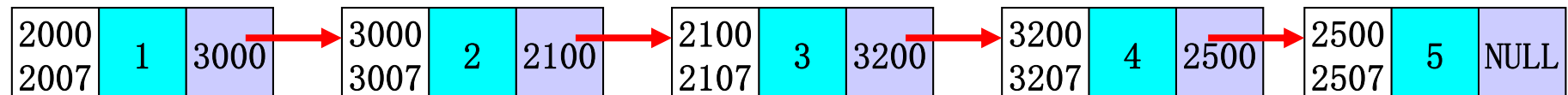
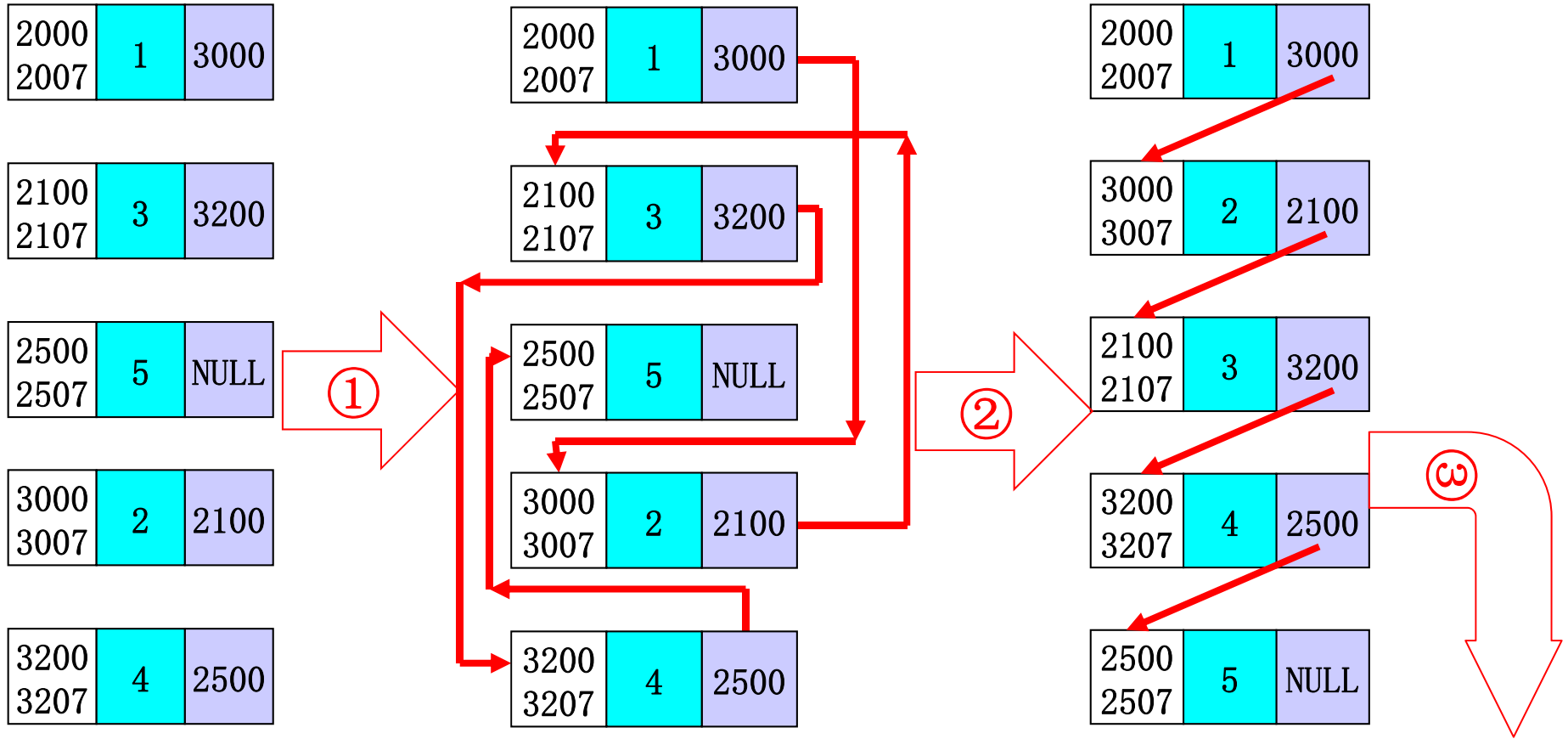
2000	1
2003	
2004	2
2007	
2008	3
2011	
2012	4
2015	
2016	5
2019	

链表

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

§ 7. 用户自定义数据类型

例：数据1，2，3，4，5，分别存放在数组和链表中



所有教材上
链表的表示方法

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.9. 指向结构体变量的指针与链表

7.2.9.1. 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

P. 206 例7.4

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

P. 206 例7.4

```
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);
```

```
return 0;
```

```
}
```

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

(结点)

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

(结点)

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

(结点)

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

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

P. 206 例7.4

```
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);
```

```
    return 0;
```

```
}
```

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	?
---	--------------	---

P. 206 例7.4

```
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);
```

```
return 0;
```

```
}
```

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	?
---	--------------	---

P. 206 例7.4

```
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);
```

```
return 0;
```

```
}
```

a	2000 2011	31001 89.5	3000
---	--------------	---------------	------

(表头结点)

b	3000 3011	31003 90	2500
---	--------------	-------------	------

(中间结点)

c	2500 2511	31007 85	NULL
---	--------------	-------------	------

(表尾结点)

head	2100 2103	2000
------	--------------	------

(头指针)

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

P. 206 例7.4

```
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 {
```

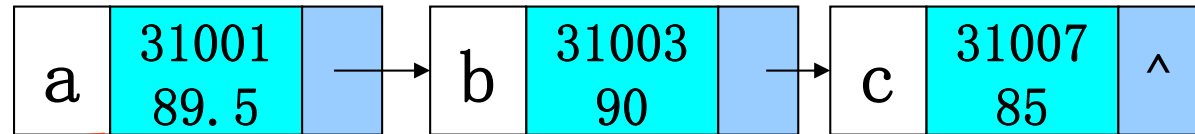
```
    cout << p->num << " " << p->score << endl;
```

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

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

```
return 0;
```

```
}
```



head

p

P. 206 例7.4

```
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 {
```

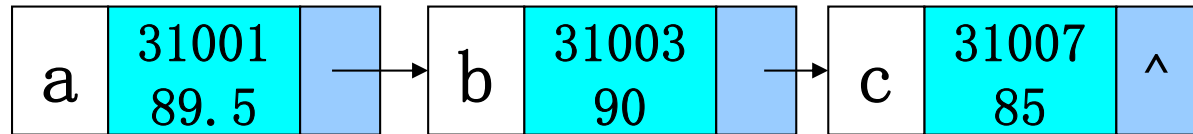
```
        cout << p->num << " " << p->score << endl;
```

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

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

```
    return 0;
```

```
}
```



head

p

P. 206 例7.4

```
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;
```

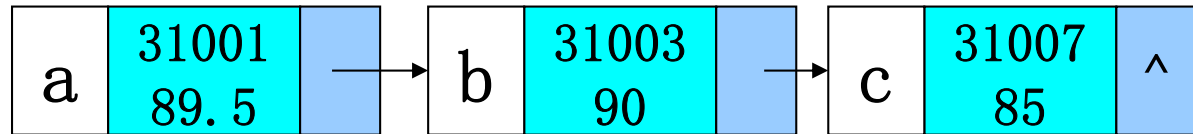
```
    do {
```

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

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

```
    return 0;
```

```
}
```



head

p

31001 89.5

P. 206 例7.4

```
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;
```

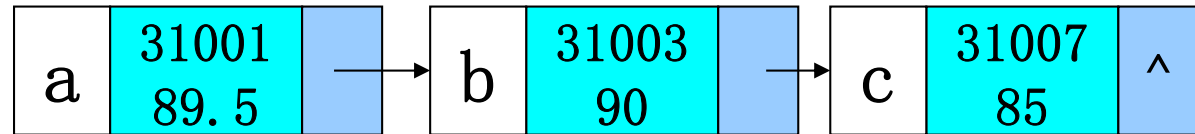
```
    do {
```

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

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

```
    return 0;
```

```
}
```



head

p

```
31001 89.5  
31003 90
```

P. 206 例7.4

```
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;
```

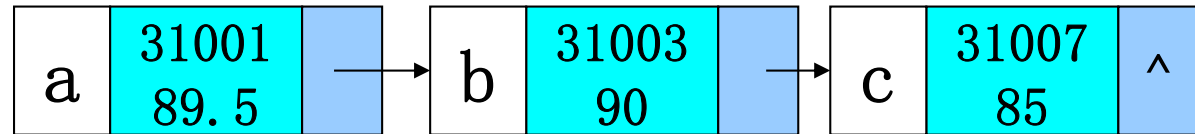
```
    do {
```

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

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

```
    return 0;
```

```
}
```



head

p

NULL

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

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.9. 指向结构体变量的指针与链表

7.2.9.1. 链式结构的基本概念

7.2.9.2. 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

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.10. 内存的动态申请与释放

7.2.10.1. 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 free(void *p);`

释放p所指的内存空间 (p必须是malloc/calloc返回的首地址)

● 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`

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

7.2.10.2. 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

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.10. 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 <code>int *p;</code> <code>p = (int *)malloc(sizeof(int));</code> <code>p = (int *)calloc(1, sizeof(int));</code>	形式1: 先定义指针变量, 再申请 <code>int *p;</code> <code>p=new int;</code>
	形式2: 定义指针变量的同时申请 <code>int *p = (int *)malloc(sizeof(int));</code> <code>int *p = (int *)calloc(1, sizeof(int));</code>	形式2: 定义指针变量的同时申请 <code>int *p=new int;</code>
		形式3: 申请空间时赋初值 <code>int *p;</code> 或 <code>int *p=new int(10);</code> <code>p=new int(10);</code>

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.10. 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>
		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: <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}; //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</pre>

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.10. 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>
		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: <pre>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</pre> 例: <pre>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</pre> 注: 字符型在使用字符串方式初始化时, 一层{ }

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.10. 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; / delete p; (一维数组可理解为元素地址, []加不加均可)</pre>
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; (二维以上数组释放时必须加一个[])</pre>

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.10. 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

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

    ...
    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;
}
```

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

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

强制类型转换

申请10个int型的变量空间
可直接写成calloc(10, 4),
但不建议, 因为适应型差

malloc(10*sizeof(int)) 和
calloc(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

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.10. 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4种情况均是编译不错执行错)

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

//p不是动态申请的空间

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

```
#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;
}
```

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

//p已不指向动态申请的空间

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

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判断, 特此说明

```
#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;
}
```

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.10. 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 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

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.10. 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;
    }
    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

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.10. 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, 则程序退出时操作系统会自动回收
(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;
}
```

×

```
#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;
}
```

×

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.10. 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

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.10. 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

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.10. 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++/--

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.10. 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
```

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.10. 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
```

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.10. 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

```
#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
```

```
#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
先student变量, 再name成员

★ 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

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

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

```
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
```

★ 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

```
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");
```

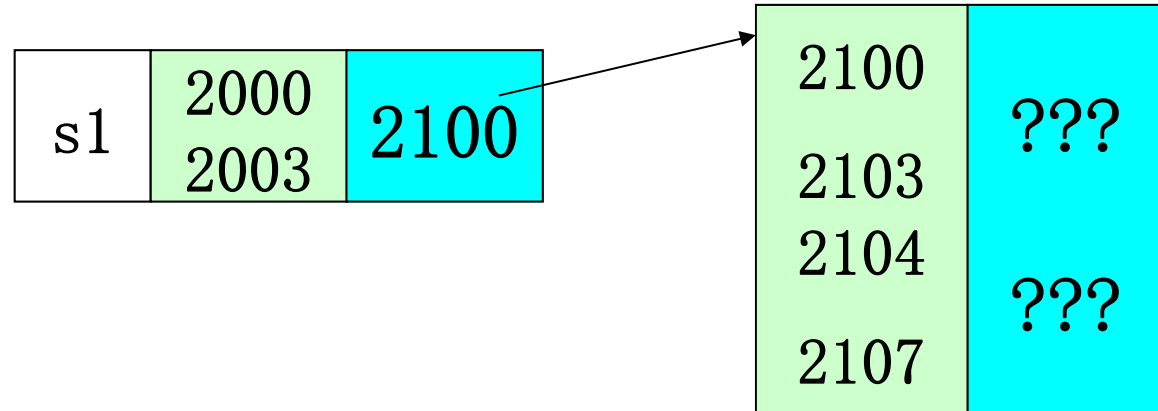
```
    cout << s1->num << ":" << s1->name << endl;
```

```
    free(s1->name); //释放6字节
```

```
    free(s1); //释放8字节
```

```
    return 0;
```

```
}
```



★ 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

```
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");
```

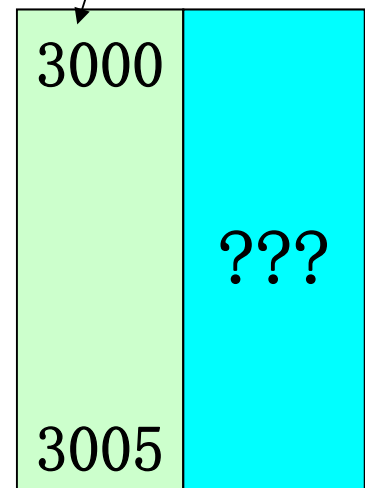
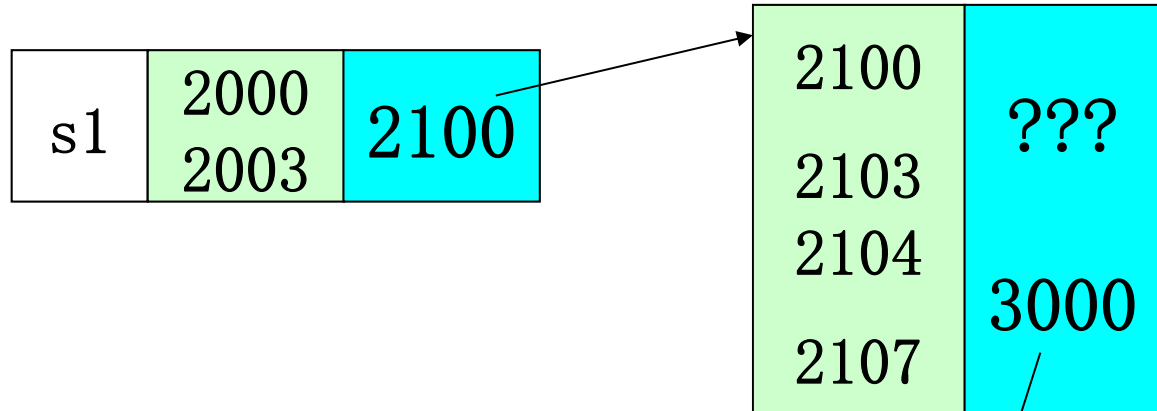
```
  cout << s1->num << ":" << s1->name << endl;
```

```
  free(s1->name); //释放6字节
```

```
  free(s1); //释放8字节
```

```
  return 0;
```

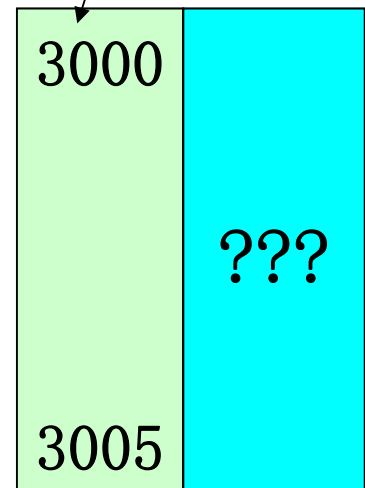
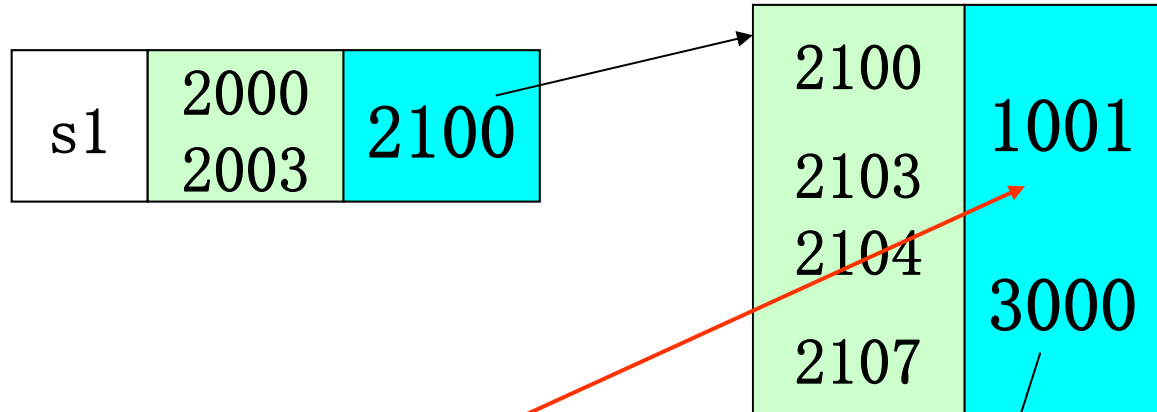
```
}
```



★ 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

```
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;  
}
```



★ 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

```
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");
```

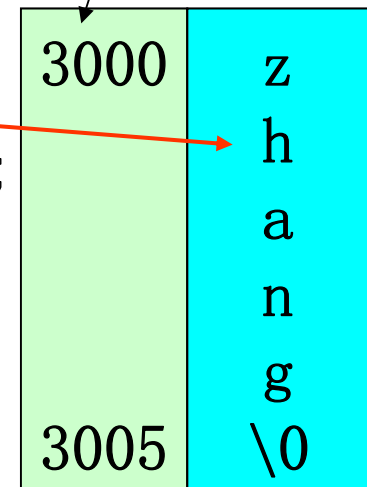
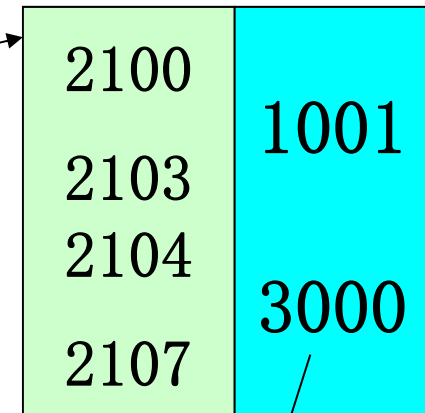
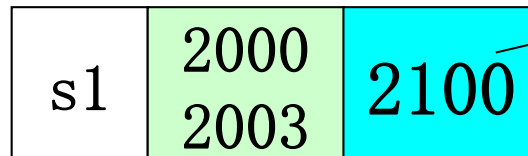
```
cout << s1->num << ":" << s1->name << endl;
```

```
free(s1->name); //释放6字节
```

```
free(s1); //释放8字节
```

```
return 0;
```

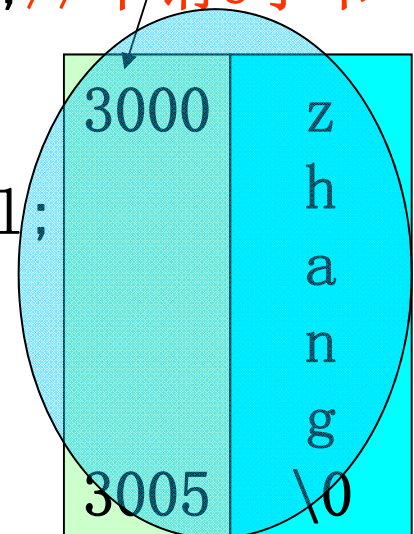
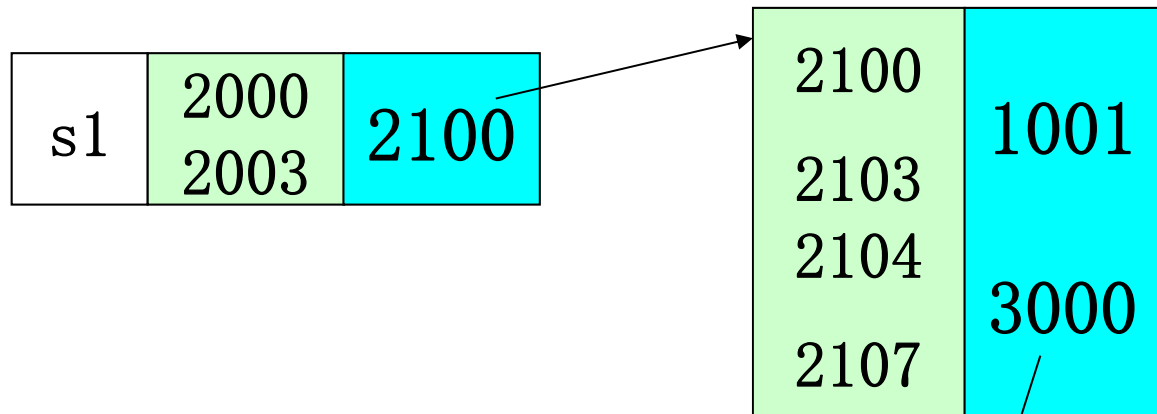
```
}
```



★ 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

```
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;  
}
```



★ 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

```
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");
```

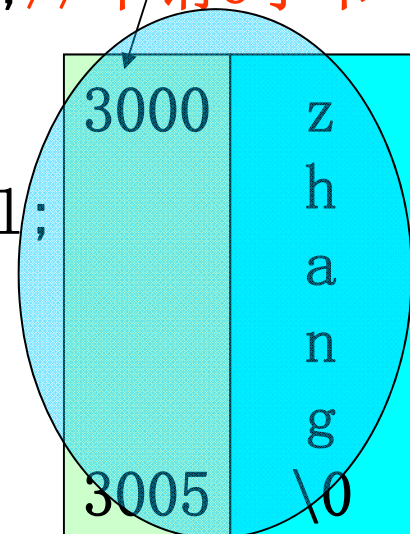
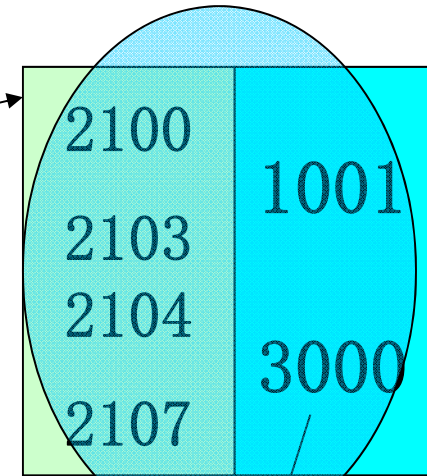
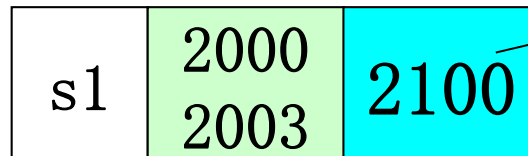
```
  cout << s1->num << ":" << s1->name << endl;
```

```
  free(s1->name); //释放6字节
```

```
  free(s1); //释放8字节
```

```
  return 0;
```

```
}
```



★ 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

```
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");
```

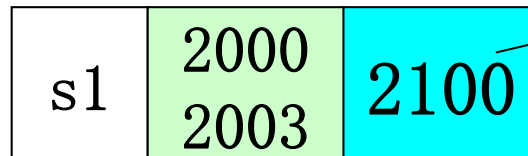
```
  cout << s1->num << ":" << s1->name << endl;
```

```
  free(s1->name); //释放6字节
```

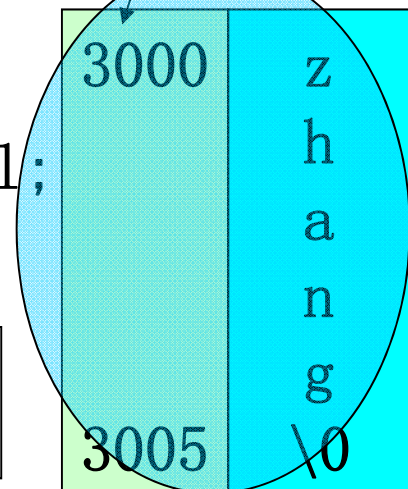
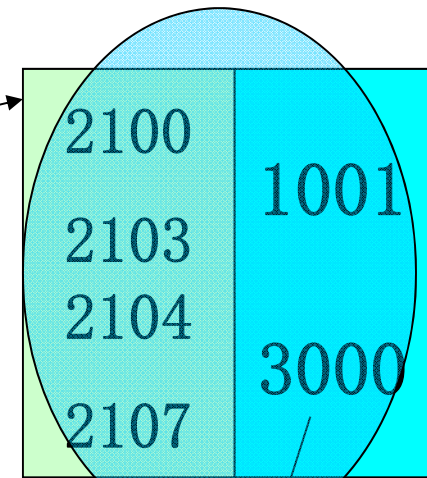
```
  free(s1); //释放8字节
```

```
  return 0;
```

```
}
```



s1自身所占4字节
由操作系统回收



free的顺序不能反

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.10. 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
}
```

```
#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
}
```

重复释放导致错误

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.10. 内存的动态申请与释放

7.2.10.1. C中的相关函数

7.2.10.2. C++中的相关运算符

```
//P.211 例7.6
#include <iostream>
#include <string>    //C++特有的string类需要
using namespace std;

struct student {
    string name; //第5章最后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

```
//P.211 例7.6
#include <iostream>
#include <string>    //C++特有的string类需要
using namespace std;

struct student {
    string name; //第5章最后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;
}
```

本例运行错误
为什么？

§ 7. 用户自定义数据类型

7.2. 结构体类型

7.2.10. 内存的动态申请与释放

7.2.10.1. C中的相关函数

7.2.10.2. C++中的相关运算符

例：在P.211 例7.6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; //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)

在P. 211 例7.6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++) {
        if (i>0)
            q=p;
        p = new(nothrow) student; //思考：为什么不能用malloc
        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;
    }
```

在P. 211 例7.6的基础上建立一个有5个结点的链表

```
int main()
```

初始状态

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

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

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

```
        q=p;
```

```
    p = new(nothrow)
```

```
    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	

在P. 211 例7.6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;
```

i=0的循环

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

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

```
        q=p;
```

```
    p = new(nothrow)
```

```
    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	3000
---	--------------	------

q	2200 2203	???
---	--------------	-----

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

在P. 211 例7.6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;
```

i=0的循环

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

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

```
        q=p;
```

```
    p = new(nothrow)
```

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

```
        return -1;
```

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

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

```
    else
```

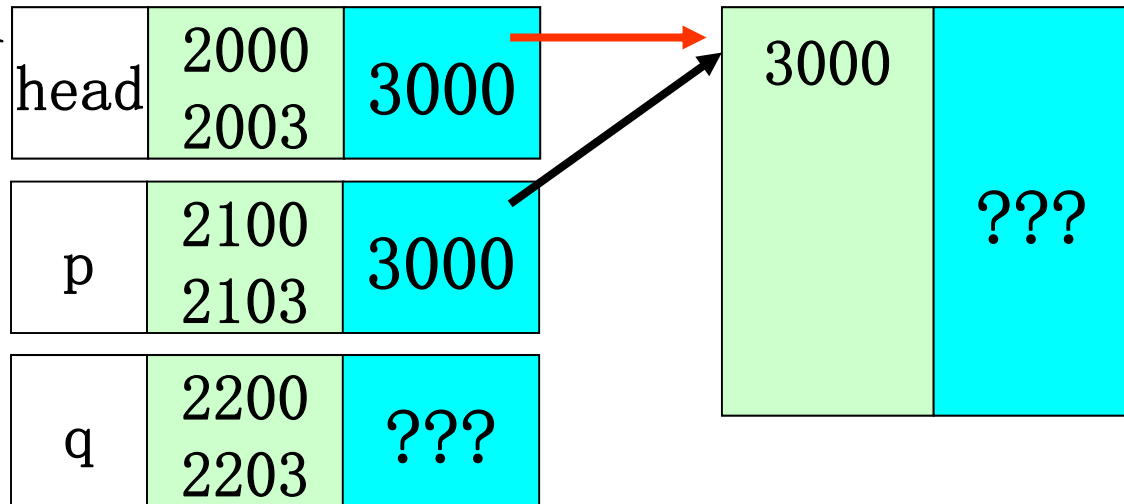
```
        q->next = p;
```

```
    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;
```

```
}
```



在P. 211 例7.6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;
```

i=0的循环

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

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

```
        q=p;
```

```
    p = new(nothrow)
```

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

```
        return -1;
```

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

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

```
    else
```

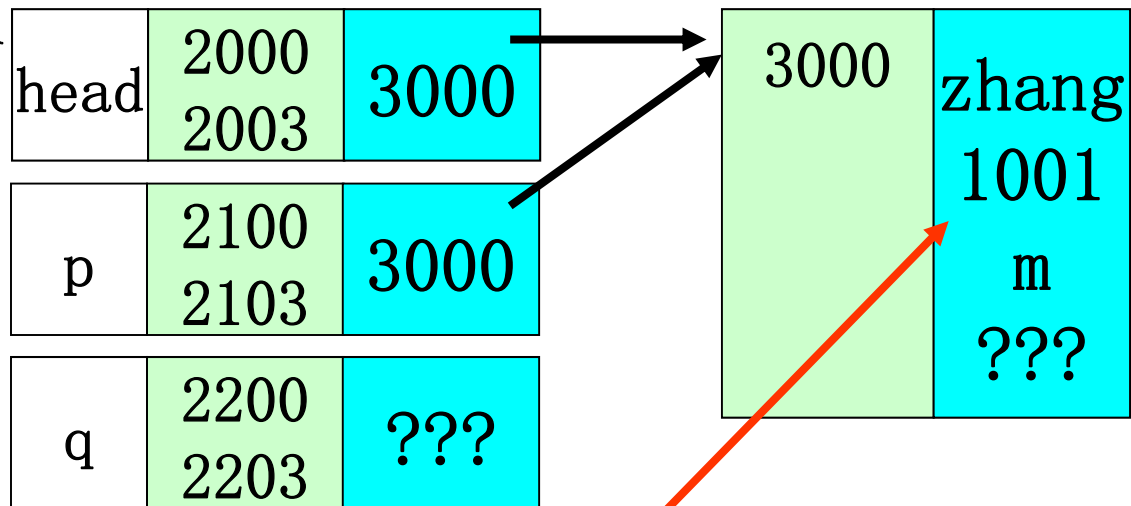
```
        q->next = p;
```

```
    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;
```

```
}
```



在P. 211 例7.6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;
```

i=0的循环结束

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

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

```
        q=p;
```

```
    p = new(nothrow)
```

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

```
        return -1;
```

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

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

```
    else
```

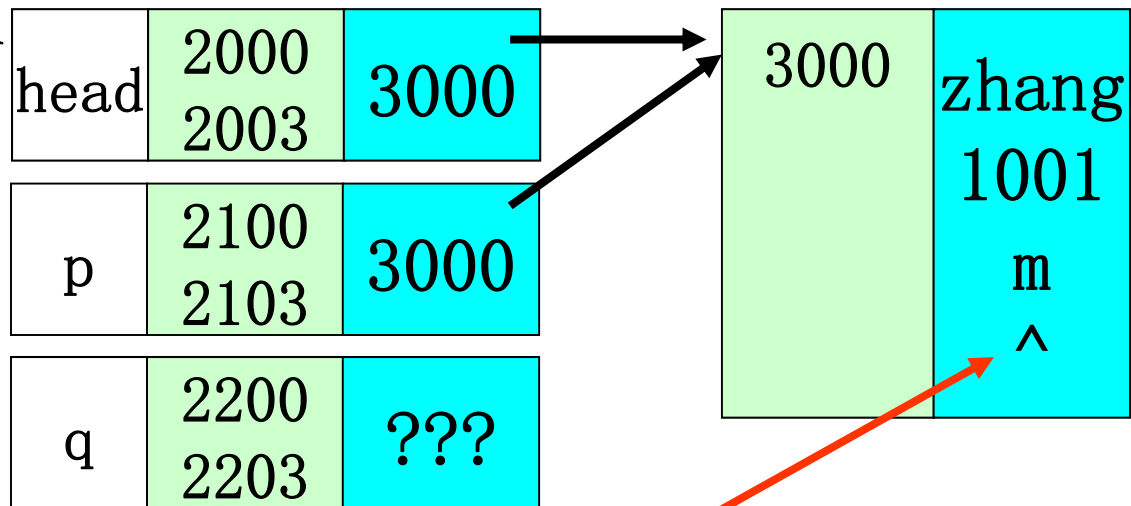
```
        q->next = p;
```

```
    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;
```

```
}
```



在P. 211 例7.6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;
```

i=0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
```

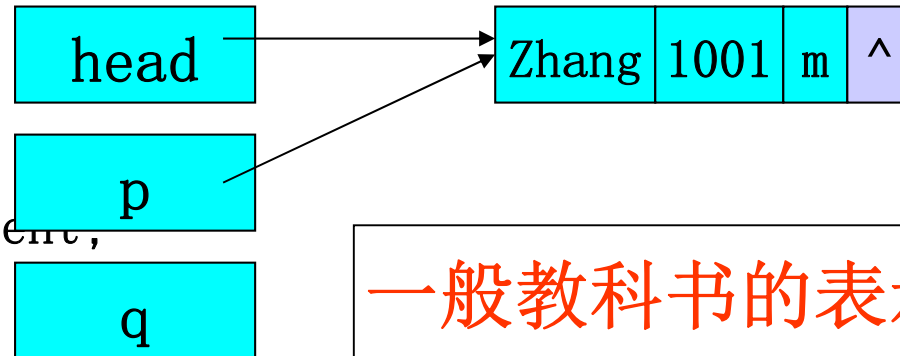
```
        q->next = p;
```

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

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

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

```
}
```



一般教科书的表示

在P. 211 例7.6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;
```

i=1的循环

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

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

```
        q=p;
```

```
    p = new(nothrow)
```

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

```
        return -1;
```

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

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

```
    else
```

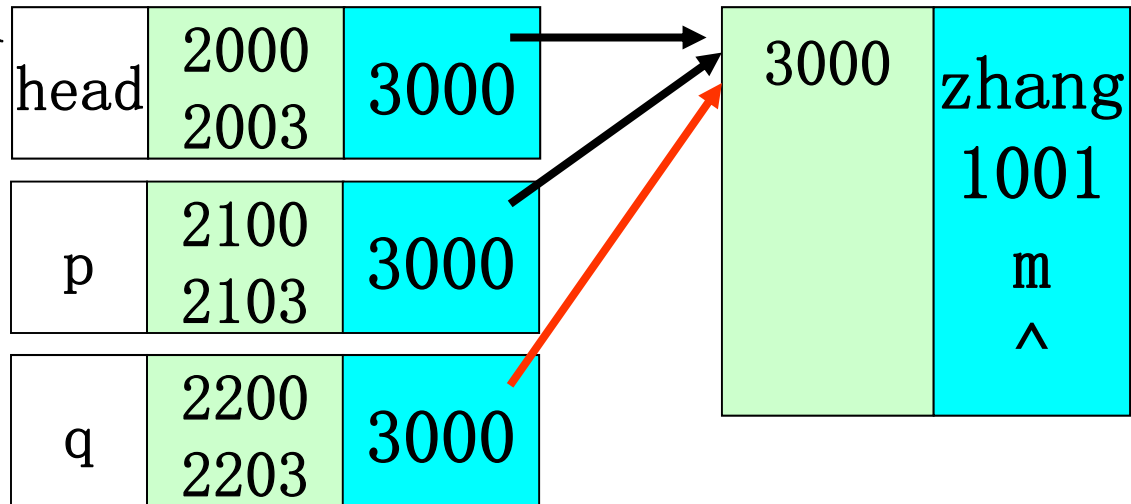
```
        q->next = p;
```

```
    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;
```

```
}
```



在P. 211 例7.6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;
```

i=1的循环

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

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

```
        q=p;
```

```
    p = new(nothrow)
```

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

```
        return -1;
```

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

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

```
    else
```

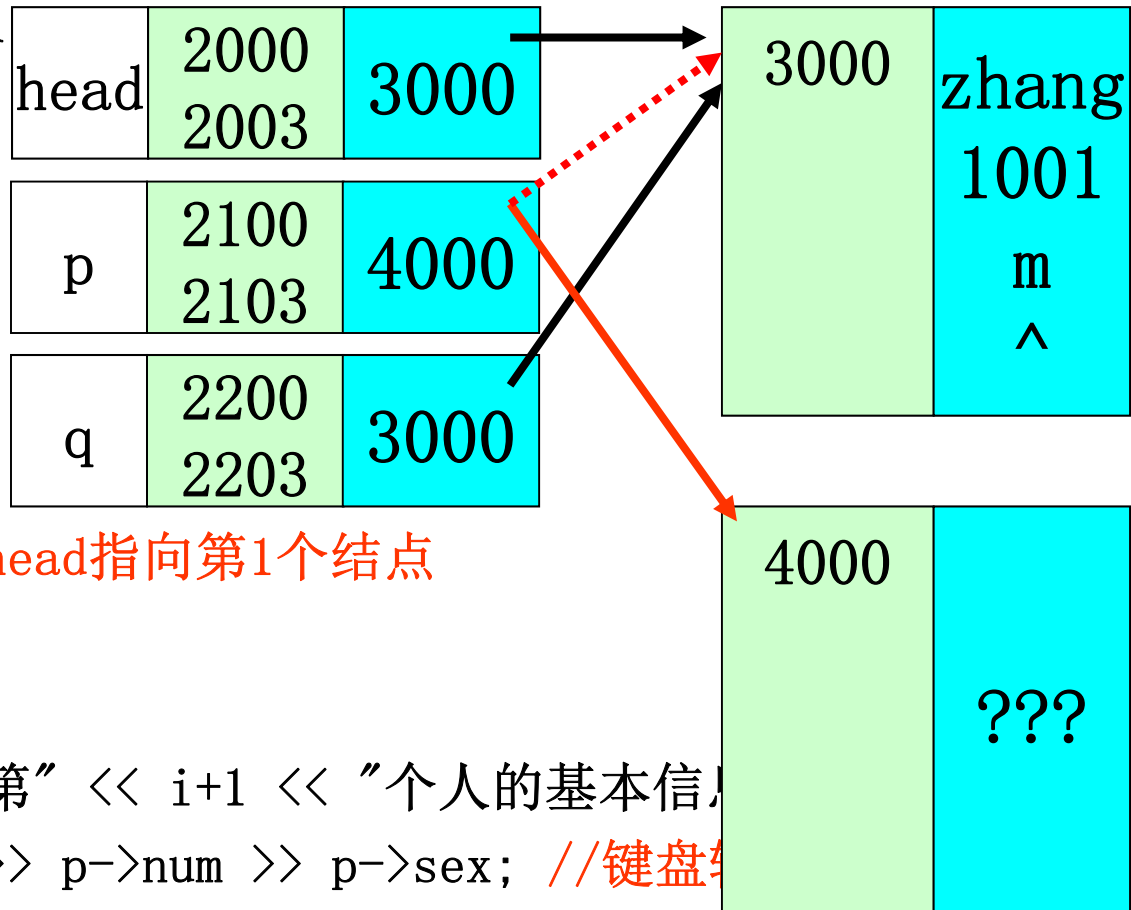
```
        q->next = p;
```

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

```
    cin >> p->name >> p->num >> p->sex; //键盘输入
```

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

```
}
```



在P. 211 例7.6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;
```

i=1的循环

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

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

```
        q=p;
```

```
    p = new(nothrow)
```

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

```
        return -1;
```

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

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

```
    else
```

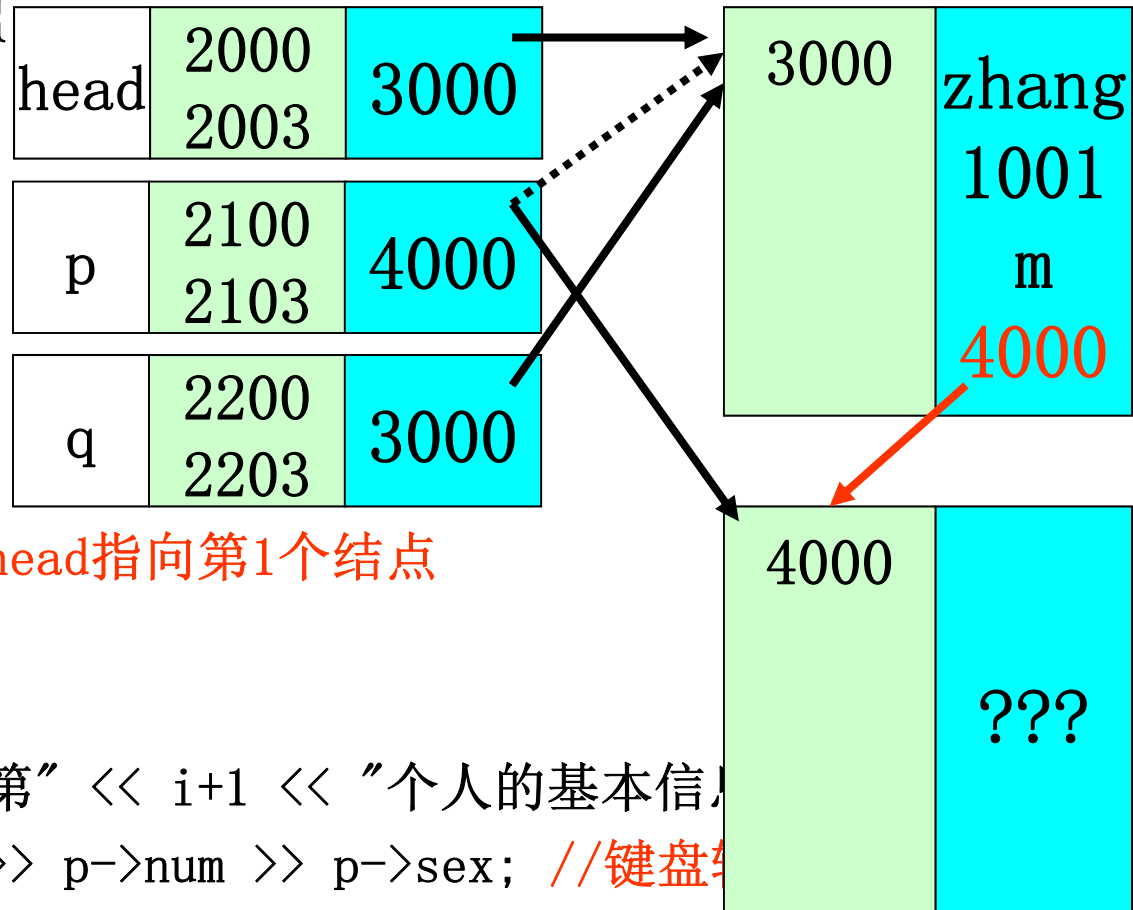
```
        q->next = p;
```

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

```
    cin >> p->name >> p->num >> p->sex; //键盘输入
```

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

```
}
```



在P. 211 例7.6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;
```

i=1的循环

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

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

```
        q=p;
```

```
    p = new(nothrow)
```

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

```
        return -1;
```

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

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

```
    else
```

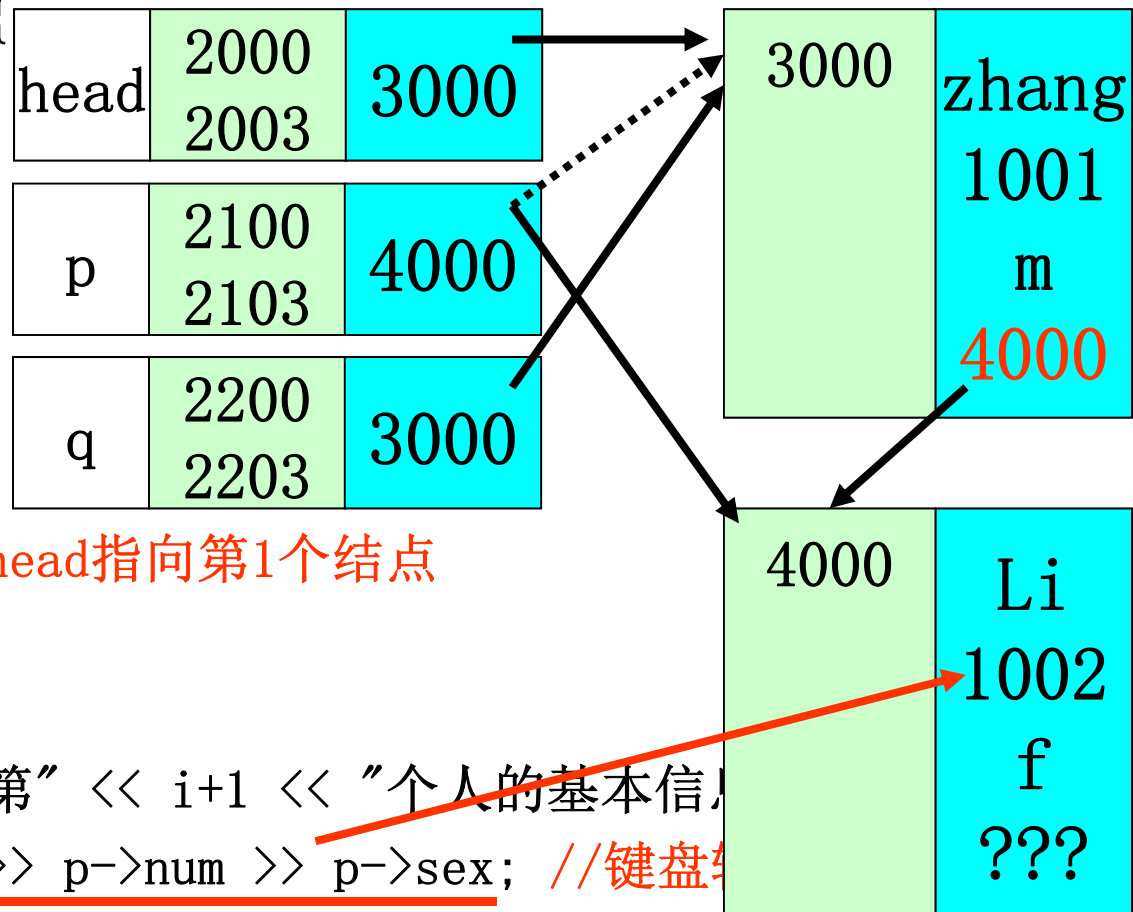
```
        q->next = p;
```

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

```
    cin >> p->name >> p->num >> p->sex; //键盘输入
```

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

```
}
```



在P. 211 例7.6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;
```

i=1的循环结束

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

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

```
        q=p;
```

```
    p = new(nothrow)
```

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

```
        return -1;
```

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

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

```
    else
```

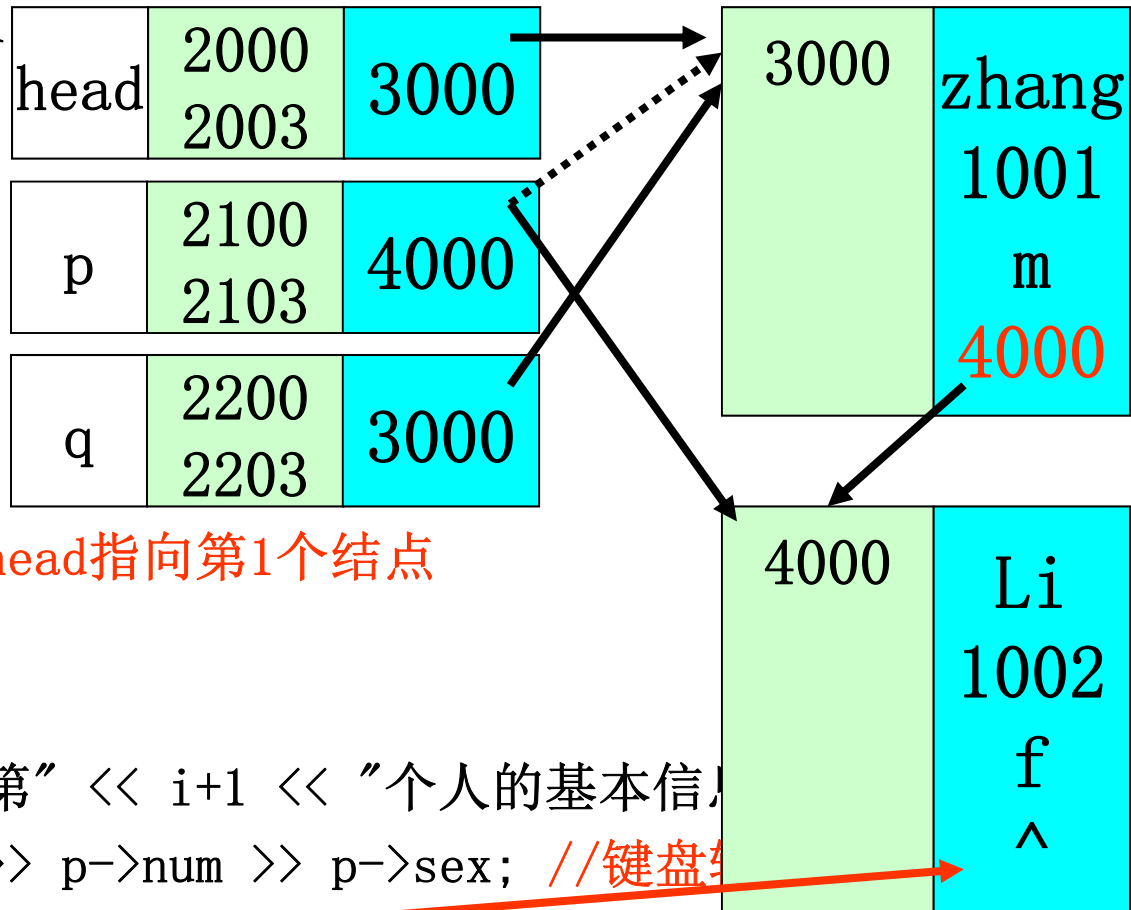
```
        q->next = p;
```

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

```
    cin >> p->name >> p->num >> p->sex; //键盘输入
```

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

```
}
```



在P. 211 例7.6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;
```

i=1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
```

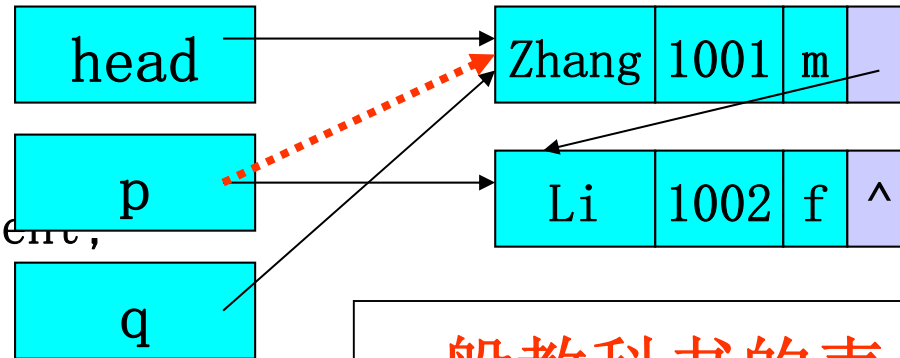
```
        q->next = p;
```

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

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

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

```
}
```



一般教科书的表示

在P. 211 例7.6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;
```

i=2-4自行画图

```
    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;
```

```
        }
```

在P. 211 例7.6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;
```

i=4的循环结束

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

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

```
        q=p;
```

```
    p = new(nothrow)
```

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

```
        return -1;
```

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

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

```
    else
```

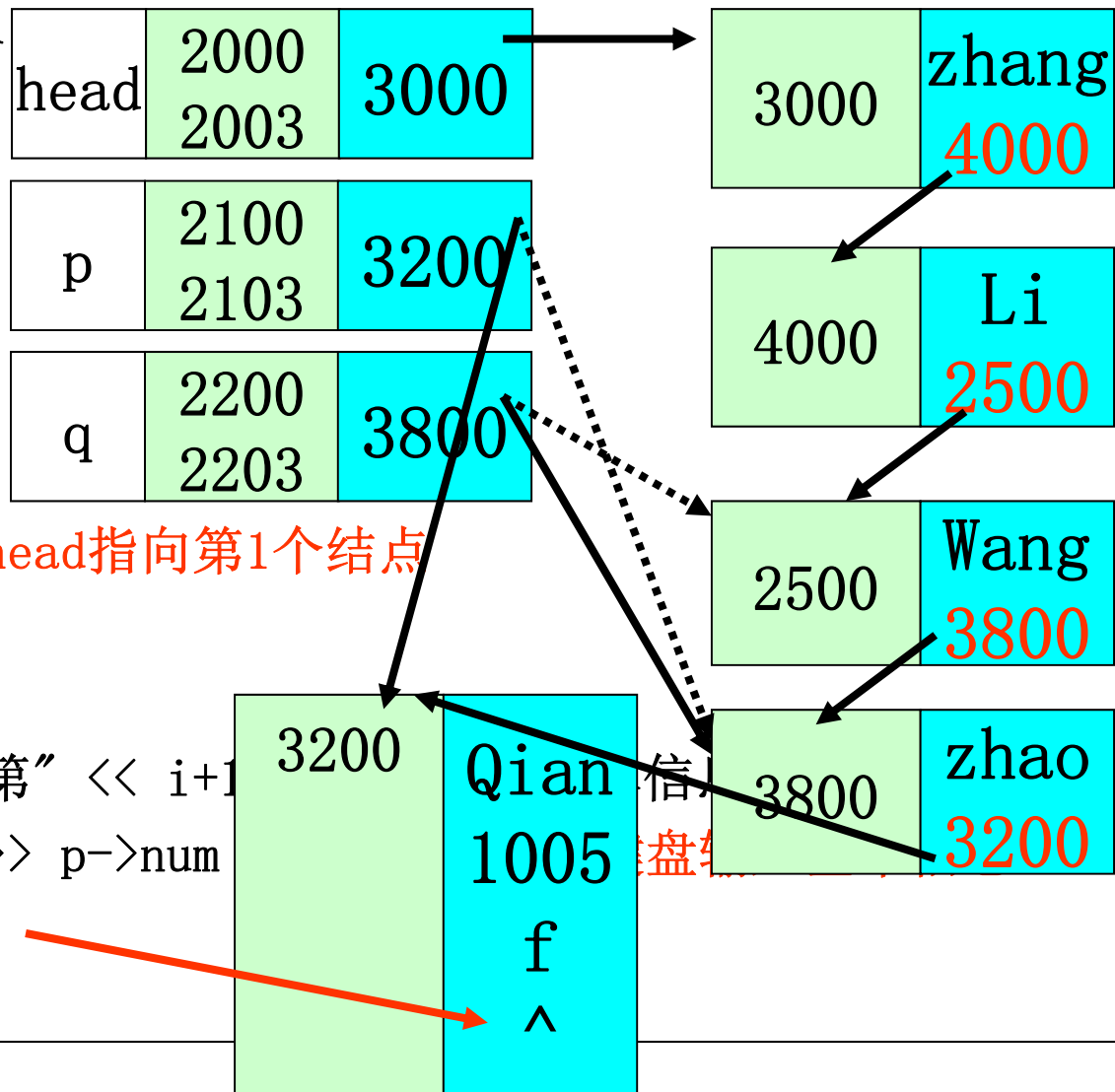
```
        q->next = p;
```

```
    cout << "请输入第" << i+1
```

```
    cin >> p->name >> p->num
```

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

```
}
```



在P. 211 例7.6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;
```

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
```

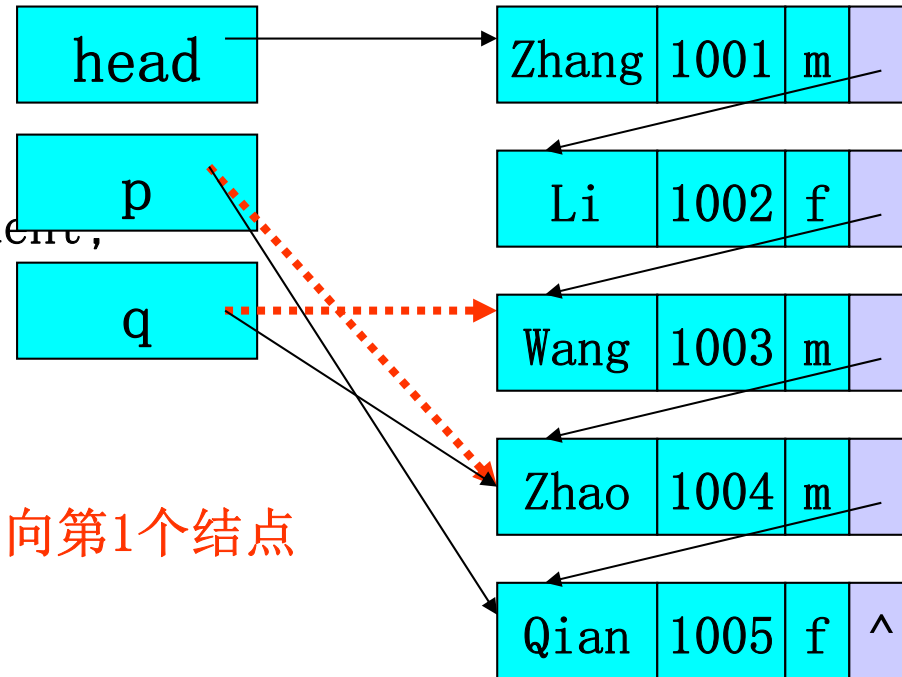
```
        q->next = p;
```

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

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

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

```
}
```



一般教科书的表示

在P. 211 例7.6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;
```

循环完成

```
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
```

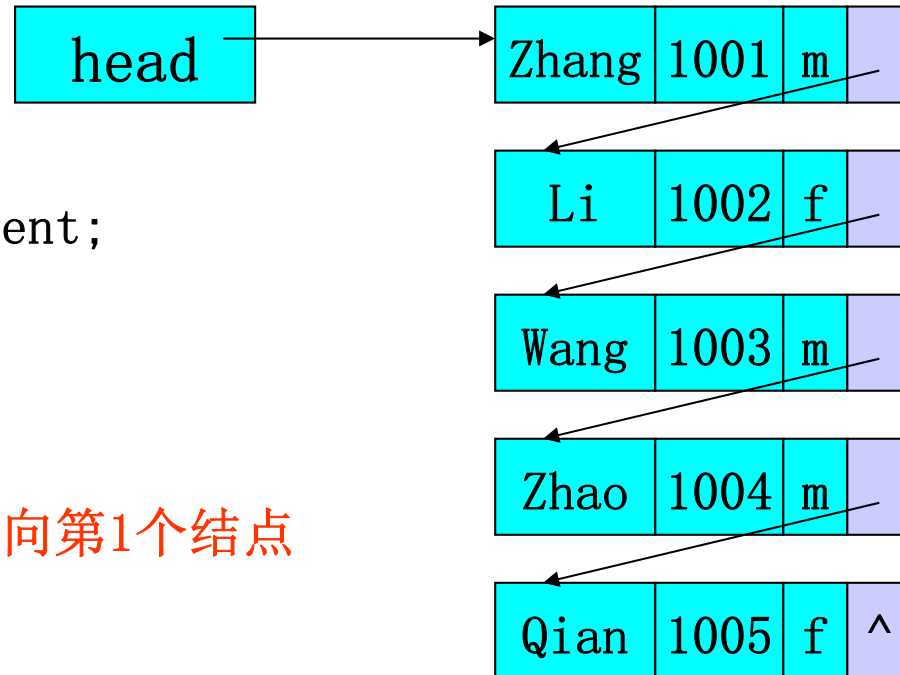
```
        q->next = p;
```

```
    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;
```

```
}
```



在P. 211 例7.6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;
```

```
}
```

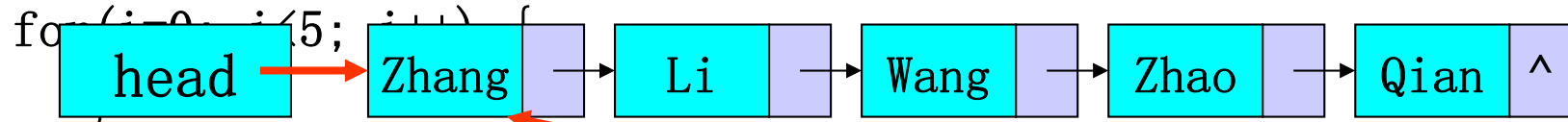
```
return 0;
```

```
}
```


在P. 211 例7.6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;
```



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;
```

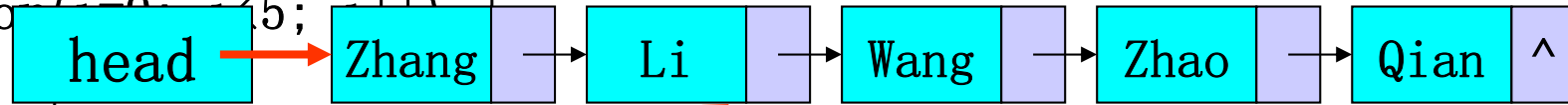
```
}
```

在P. 211 例7.6的基础上建立一个有5个结点的链表

```
int main()
```

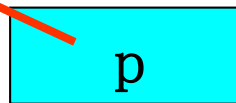
```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

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



```
    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;
```

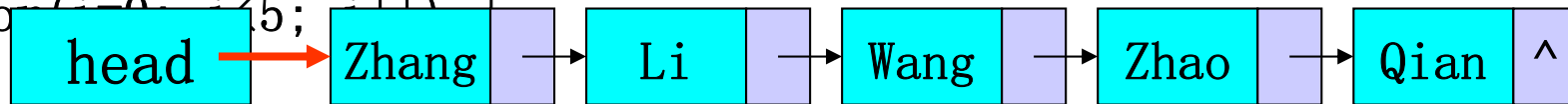
```
}
```

在P. 211 例7.6的基础上建立一个有5个结点的链表

```
int main()
```

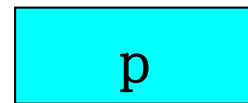
```
{  student *head=NULL, *p=NULL, *q=NULL;  int i;
```

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



```
    p=head; //p复位，指向第1个结点
```

```
    while(p!=NULL) { //循环进行输出
```



后续输出自行画图理解

```
        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;
```

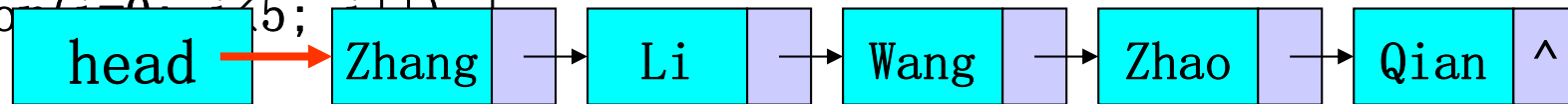
```
}
```

在P. 211 例7.6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++) {
```



```
        p=head; //p复位，指向第1个结点
```

```
        while(p!=NULL) { //循环进行输出
```

p: ^

最后一个结点输出

```
            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;
```

```
}
```

Zhang 1001 m

Li 1002 f

Wang 1003 m

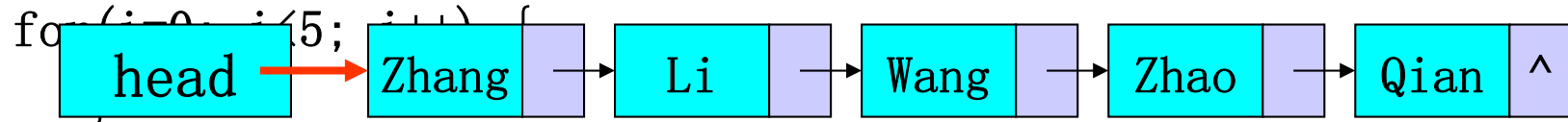
Zhao 1004 m

Qian 1005 f

在P. 211 例7.6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;
```



```
p=head; //p复位，指向第1个结点
```

```
while(p!=NULL) { //循环进行输出
```

p: ^

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

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

```
}
```

循环结束

```
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;
```

```
}
```

Zhang 1001 m

Li 1002 f

Wang 1003 m

Zhao 1004 m

Qian 1005 f

```
return 0;
```

```
}
```

在P. 211 例7.6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;
```

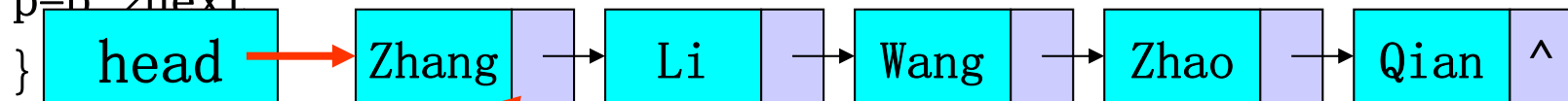
```
  for(i=0; i<5; i++) {  
    }
```

```
  p=head; //p复位，指向第1个结点
```

```
  while(p!=NULL) { //循环进行输出
```

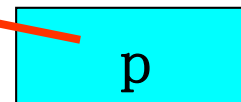
```
    cout << p->name << " " << p->num << " " << p->sex << endl;
```

```
    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;
```

```
}
```

在P. 211 例7.6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;
```

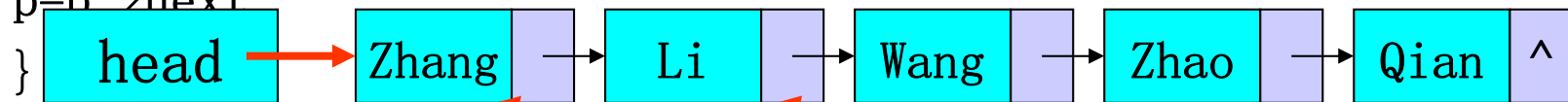
```
  for(i=0; i<5; i++) {  
    }
```

```
  p=head; //p复位，指向第1个结点
```

```
  while(p!=NULL) { //循环进行输出
```

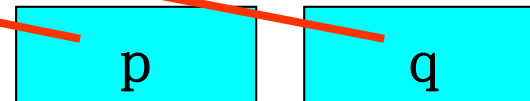
```
    cout << p->name << " " << p->num << " " << p->sex << endl;
```

```
    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;
```

```
}
```

在P. 211 例7.6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;
```

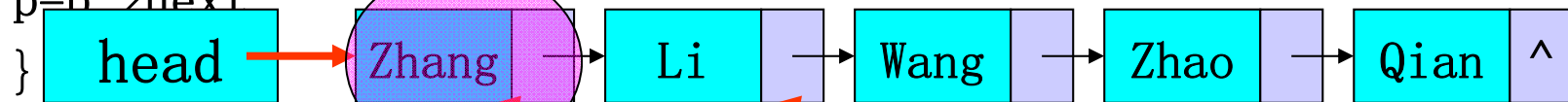
```
  for(i=0; i<5; i++) {  
    }
```

```
  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;
```

```
}
```


在P. 211 例7.6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;
```

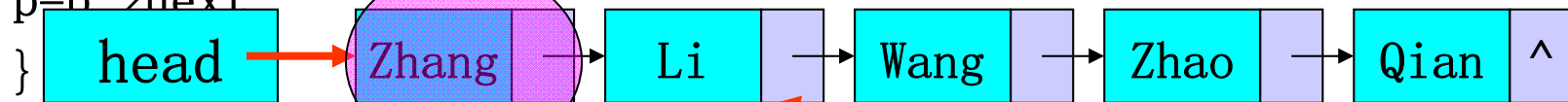
```
  for(i=0; i<5; i++) {  
    }
```

```
  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;
```

```
}
```

在P. 211 例7.6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;
```

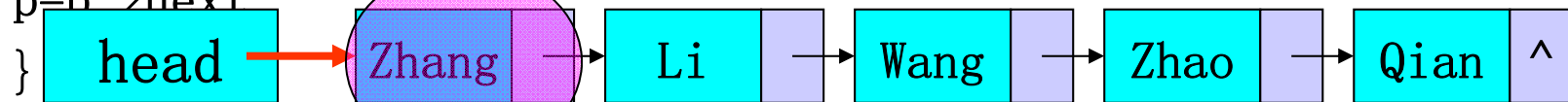
```
  for(i=0; i<5; i++) {  
    }
```

```
  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
```

p

q

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

```
    delete p;
```

```
    p = q;
```

```
  }
```

后续结点释放自行画图理解

```
  return 0;
```

```
}
```

在P. 211 例7.6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;
```

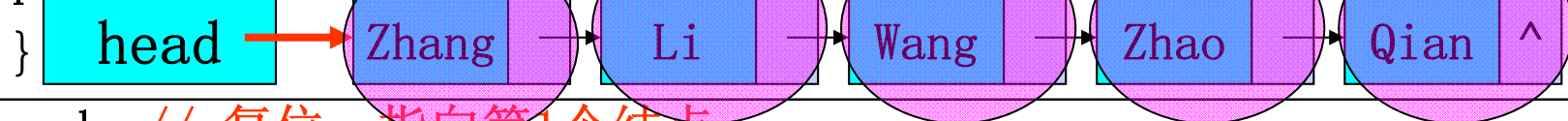
```
  for(i=0; i<5; i++) {  
    }
```

```
  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

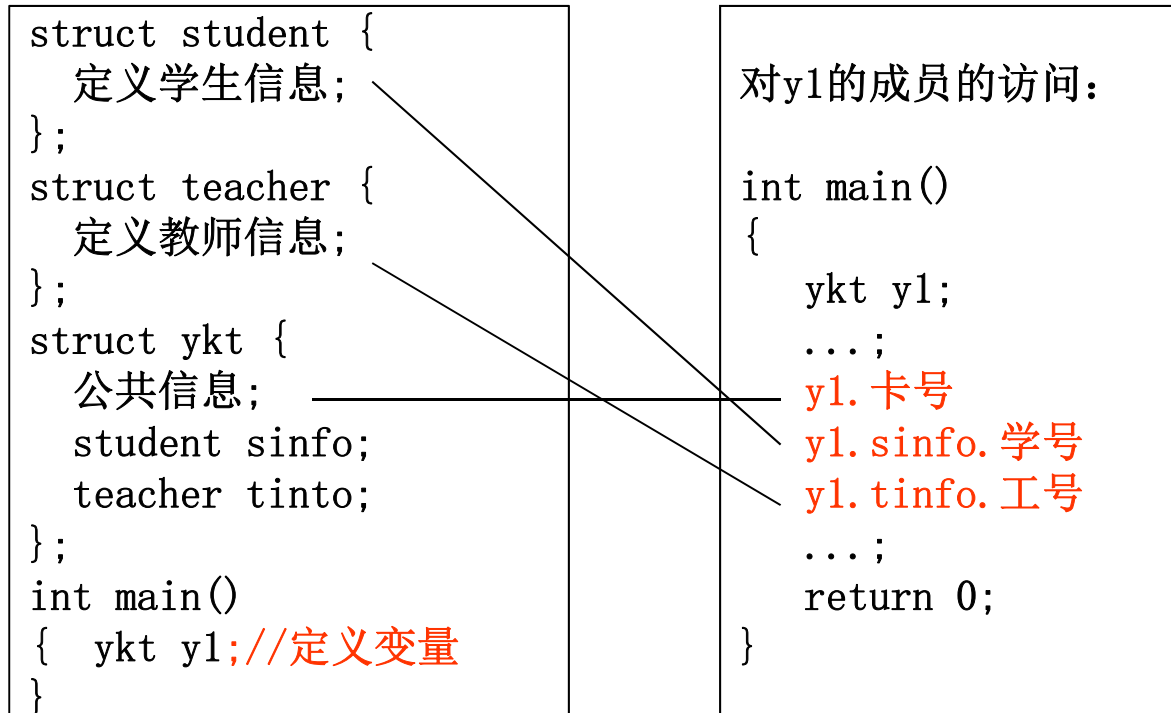
```
  return 0;
```

```
}
```

§ 7. 用户自定义数据类型

7.3. 共用体

例： 定义一个用于一卡通管理系统的结构，要求包含卡号、余额、消费限额、消费密码等**公共信息**，此外，若持卡人是学生，要包含学号、姓名、专业等**学生特有的信息**，若持卡人是教师，则包含工号、姓名、职称等**教师特有的信息**



缺陷：无论持卡人何种身份sinfo和tinfo中必然有一个是不需要填写任何信息的，从而导致存储空间的浪费

解决：能否使sinfo/tinfo共用一段空间，当持卡人是学生时，这段空间按student方式访问，当持卡人是教师时按teacher方式访问 => (共用体)

§ 7. 用户自定义数据类型

7.3. 共用体

```
union 共用体名 {  
    共用体成员1 (类型名 成员名)  
    ...  
    共用体成员n (类型名 成员名)  
};  
  
union data {  
    short a;  
    long b;  
    char c;  
};
```

- ★ 所有成员从同一内存开始，共用体的大小为其中占用空间最大的成员的大小
- ★ 给一个共用体成员赋值后，会覆盖其它成员的值，因此只有最后一次存放的成员是有效的
- ★ 其它所有定义、使用方法同结构体

```
#include <iostream>  
using namespace std;  
  
struct data1 {  
    short a;  
    long b;  
    char c;  
};  
  
union data2 {  
    short a;  
    long b;  
    char c;  
};  
  
void main()  
{ cout << sizeof(data1) << ' ' << sizeof(data2) << endl;  
}
```

12: 所有成员所占空间之和(含填充字节)

4 : 所有成员中最大成员所占空间

12 4

struct data1 d1;

d1	2000	a
	2001	
	2002	b
	2003	
	2004	
	2005	
	2006	c

union data2 d2;

d2	3000	a	b	c
	3001			
	3002			
	3003			

§ 7. 用户自定义数据类型

7.3. 共用体

★ 所有成员从同一内存开始，共用体的大小为其中占用空间最大的成员的大小

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

70000=00000000 00000001 00010001 01110000

```
union data {
    int  a;
    short b;
    char  c;
};
```

```
int main()
{
    union data d;
    d.a=70000;
    cout << d.a << ' ' << d.b << ' ' << d.c << endl;
    d.b=7000;
    cout << d.a << ' ' << d.b << ' ' << d.c << endl;
    d.c='A';
    cout << d.a << ' ' << d.b << ' ' << d.c << endl;
    return 0;
}
```

d: 低位在前存放				
2000	01110000	a	b	c
2001	00010001			
2002	00000001			
2003	00000000			

70000 4464 p

72536 7000 X

72513 6977 A

§ 7. 用户自定义数据类型

7.3. 共用体

★ 所有成员从同一内存开始，共用体的大小为其中占用空间最大的成员的大小

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

```
union data {
    int  a;
    short b;
    char  c;
};
```

```
int main()
{
    union data d;
    d.a=70000;
    cout << d.a << ' ' << d.b << ' ' << d.c << endl;
    d.b=7000;
    cout << d.a << ' ' << d.b << ' ' << d.c << endl;
    d.c='A';
    cout << d.a << ' ' << d.b << ' ' << d.c << endl;
    return 0;
}
```

72536=00000000 00000001 00011011 01011000

d: 低位在前存放

2000	01011000	a	b	c
2001	00011011			
2002	00000001			
2003	00000000			

70000 4464 p

72536 7000 X

72513 6977 A

7000=00011011 01011000

§ 7. 用户自定义数据类型

7.3. 共用体

★ 所有成员从同一内存开始，共用体的大小为其中占用空间最大的成员的大小

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

```
union data {
    int  a;
    short b;
    char  c;
};
```

```
int main()
{
    union data d;
    d.a=70000;
    cout << d.a << ' ' << d.b << ' ' << d.c << endl;
    d.b=7000;
    cout << d.a << ' ' << d.b << ' ' << d.c << endl;
    d.c='A';
    cout << d.a << ' ' << d.b << ' ' << d.c << endl;
    return 0;
}
```

72513=00000000 00000001 00011011 01000001

d:低位在前存放				
2000	01000001	a	b	c
2001	00011011			
2002	00000001			
2003	00000000			

A=0100 0001

§ 7. 用户自定义数据类型

7.3. 共用体

★ 所有成员从同一内存开始，共用体的大小为其中占用空间最大的成员的大小

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

```
union data {
    int  a;
    short b;
    char  c;
};
```

```
int main()
{
    union data d;
    d.c='A';
    cout << d.a << ' ' << d.b << ' ' << d.c << endl;
    d.b=7000;      不确定 不确定 A
    cout << d.a << ' ' << d.b << ' ' << d.c << endl;
    d.a=70000;     不确定 7000 X
    cout << d.a << ' ' << d.b << ' ' << d.c << endl;
    return 0;      70000 4464 p
}
```

d: 低位在前存放

2000	01000001
2001	???
2002	???
2003	???

d: 低位在前存放

2000	01011000
2001	00011011
2002	???
2003	???

d: 低位在前存放

2000	01110000
2001	00010001
2002	00000001
2003	00000000

§ 7. 用户自定义数据类型

7.3. 共用体

```
struct student {  
    定义学生信息;  
};
```

```
struct teacher {  
    定义教师信息;  
};
```

```
struct ykt {  
    公共信息;  
    student sinfo;  
    teacher tinto;  
};
```

空间
浪费

```
int main()  
{ ykt y1;//定义变量  
}
```

```
struct student {  
    定义学生信息;  
};
```

```
struct teacher {  
    定义教师信息;  
};
```

```
union owner {  
    student s;  
    teacher t;  
};
```

此处保证s/t
共用一段空间

```
struct ykt {  
    公共信息;  
    char type; //持卡人类别  
    owner info;  
};
```

```
int main()  
{ ykt y1;//定义变量  
}
```

```
int main()  
{  
    ykt y1;//定义变量  
    ...;  
    y1.卡号;  
    if (y1.type=='s'){  
        y1.info.s.学号;  
    }  
    else {  
        y1.info.t.工号;  
    }  
    ...;  
    return 0;  
}
```

§ 7. 用户自定义数据类型

7.4. 枚举类型

7.4.1. 含义

当变量的取值在有限范围内时，可以一一列出，称为枚举类型

性别、星期、月份、血型

7.4.2. 枚举类型的定义

```
enum 枚举类型名 {枚举元素1, ..., 枚举元素n};
```

```
enum sex {male, female};
```

```
enum week {sun, mon, tue, wed, thu, fri, sat};
```

```
enum month {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
```

```
enum blood_type {A, B, O, AB};
```

★ 枚举类型和元素的命名同变量

★ 枚举元素也称枚举常量，不是字符串，不加“”，作为整型常量处理，值从0开始顺序递增，也可自行指定，在程序执行中值不可变

```
enum week {sun, mon, tue, wed, thu, fri, sat};
```

0 1 2 3 4 5 6

```
enum week {sun=7, mon=1, tue, wed, thu, fri, sat};
```

7 1 2 3 4 5 6

```
enum week {sun=7, mon, tue, wed, thu, fri, sat};
```

7 8 9 10 11 12 13

★ 如果执行的常量值出现重叠，不算错误

```
enum week {sun=3, mon=1, tue, wed, thu, fri, sat};
```

3 1 2 3 4 5 6

§ 7. 用户自定义数据类型

7.4. 枚举类型

7.4.3. 枚举类型变量的定义

~~enum~~ 枚举类型名 变量名 (红色删除线: 在C++下定义时, enum关键字可省略)

~~enum~~ week w1, w2;

7.4.4. 枚举类型变量的使用

赋值: w1=mon w2=fri w2=w1

★ 不能直接赋整型量, 需要进行强制类型转换

w1 = 3; 错误

w1 = (week)3; 正确

比较: w1==mon w2>sat w2>=w1

★ 按枚举常量对应的值进行比较

输出: 直接输出: (按整型输出)

```
w1 = wed;
```

```
cout << w1 << endl; w1=3
```

间接输出: (可以是自定义的任意格式)

```
switch(w1) {  
    case sun:  
        cout << "Sunday" << endl;  
        break;  
    ...  
}
```

输入: 直接输入: 只能用C方式, 且不检查范围

```
week w1;
```

```
scanf("%d", &w1);
```

```
cin >> w1; 不允许
```

间接输入: 可以多种格式

```
char s[80];
```

```
cin >> s;
```

```
if (!strcmp(s, "sun"))
```

```
    w1=sun;
```

```
else if (...)
```

```
int w;
```

```
cin >> w;
```

```
w1=(week)w;
```

§ 7. 用户自定义数据类型

7. 4. 枚举类型

7. 4. 4. 枚举类型变量的使用

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

enum month {Jan=1, Feb, Mar, Apr, May, Jun,
            Jul, Aug, Sep, Oct, Nov, Dec};

int main()
{
    int i, m[13], *p;
    for (i=Jan; i<=Dec; i++)
        if (i==Feb)
            m[i] = 28;
        else if (i==Apr|| i==Jun|| i==Sep|| i==Nov)
            m[i] = 30;
        else
            m[i] = 31;

    for (p=&m[1]; p<m+13; p++)
        cout << *p << " ";
    cout << endl;

    return 0;
}
```

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

int main()
{
    int i, m[13], *p;
    for (i=1; i<=12; i++)
        if (i==2)
            m[i] = 28;
        else if (i==4|| i==6|| i==9|| i==11)
            m[i] = 30;
        else
            m[i] = 31;

    for (p=&m[1]; p<m+13; p++)
        cout << *p << " ";
    cout << endl;

    return 0;
}
```

特别说明:

enum枚举类型不属于一定要使用的概念, 左右的例子分别是使用/未使用enum的情况, 功能相同, 请大家自行体会

§ 7. 用户自定义数据类型

7.5. 用typedef声明类型

7.5.1. 含义

用新的名称来等价代替已有的数据类型

- ★ 不产生新类型，仅使原有类型有新的名称
- ★ 建议声明的新类型为大写，与系统类型区分


7.5.2. 使用

声明名称：

```
typedef 已有类型 新名称;  
typedef int INTEGER;  
typedef struct student STUDENT;  
typedef int ARRAY[10]  
typedef char * STRING
```

定义变量：

INTEGER i, j;		int i, j;
STUDENT s1, s2[10], *s3;		student s1, s2[10], *s3;
ARRAY a, b[5];		int a[10], b[5][10];
STRING p, x[10];		char *p, *x[10];



§ 7. 用户自定义数据类型

7.5. 用typedef声明类型

7.5.3. 声明新类型的一般步骤

- ① 以现有类型定义一个变量
- ② 将变量名替换为新类型名
- ③ 加typedef
- ④ 完成, 可定义新类型的变量



```
① int i;  
② int INTEGER;  
③ typedef int INTEGER  
④ INTEGER i, j;
```

```
① int a[10];  
② int ARRAY[10];  
③ typedef int ARRAY[10]  
④ ARRAY a, b[5];
```

```
① char *s;  
② char *STRING;  
③ typedef char *STRING;  
④ STRING p, x[10];
```

```
① int (*p) ();  
② int (*PFUN) ();  
③ typedef int (*PFUN) ()  
④ PFUN p1, p2;
```

```
① int (*p)[4];  
② int (*PA)[4];  
③ typedef int (*PA)[4]  
④ PA p;
```

★ 使用方法与原来的类型一致, 与原类型可直接混用不需要进行强制类型转换

```
#include <iostream>  
#include <cstring>  
using namespace std;  
typedef char * STRING;  
int main()  
{   char *p1="house";  
    STRING p2="horse";  
    if (strcmp(p1, p2)>0)  
        cout<<"大于"<<endl;  
    else  
        cout<<"不大于"<<endl;  
    return 0;  
}
```

特别说明:
typedef声明新类型
不属于必须使用的方法,
使用后带来的方便/不方便之处请大家自行体会

//运行结果为 “大于”