

# 数据结构与算法



主讲教师：刘峤

# 知识回顾：结构（C语言）

# 结构变量

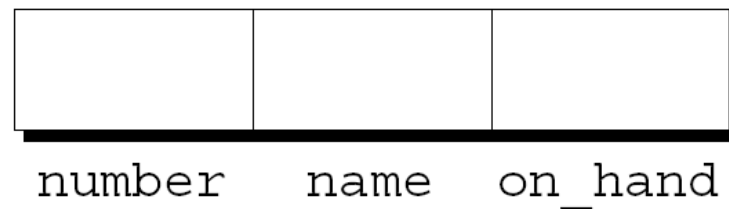
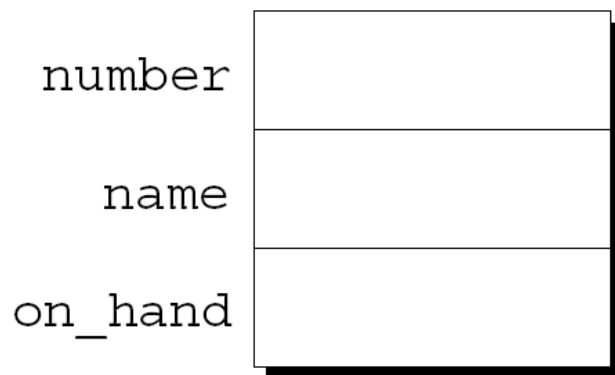
```
struct {  
    int number;  
    char name[LEN+1];  
    int on_hand;  
} partA, partB;
```

- ❧ 结构用于存储一组逻辑上相关的数据
  - 例如：声明两个结构变量，用于表示仓库里的零部件
- ❧ 结构的性质（与数组的区别）
  - 结构的成员可以具有不同的数据类型
  - 结构的成员有名字
  - 可以通过名字而不是位置去选择一个指定的成员



# 声明结构变量

## 结构的抽象表示



## 成员的值将存放在对应的内存空间里

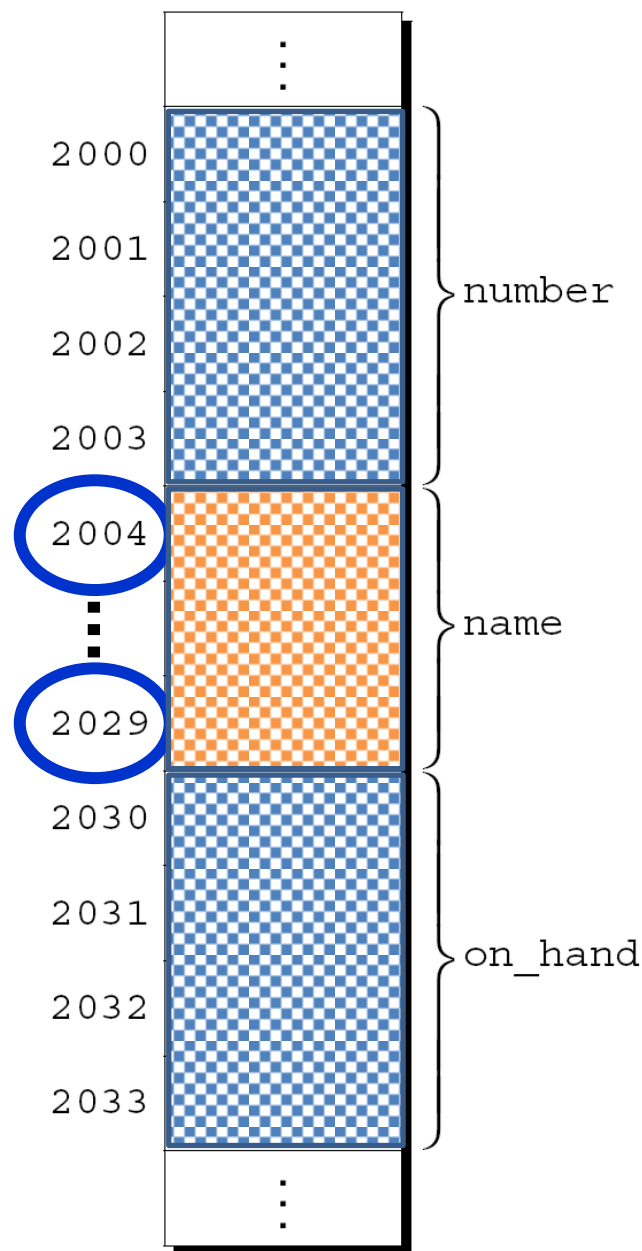
# 声明结构变量

☞ 结构成员按声明顺序存储在内存中

```
struct {  
    int number;  
    char name[LEN+1];  
    int on_hand;  
} partA;
```

☞ 假设 **partA** 的起始地址为2000

- partA 在内存中的存储方式如图
- 整数占4字节
- 此处 **LEN** 的值是25
- 成员之间没有空隙



# 声明结构变量

**struct 结构标记**

**注：结构标识符 ≠ 结构类型名**

{

**类型名1 成员名1;**

**类型名2 成员名2;**

.....

**类型名n 成员名n;**

**} ;**

**注：结构类型定义之后一定要跟一个分号**



# 声明结构变量

例：将学生信息以人为单位进行定义

```
struct student {  
    char name[21];  
  
    int gender;    // 0:male; 1:female  
  
    int birthday;  // 20101111  
  
    double height;  
  
};
```

结构类型名？

**struct student**

结构标识符？

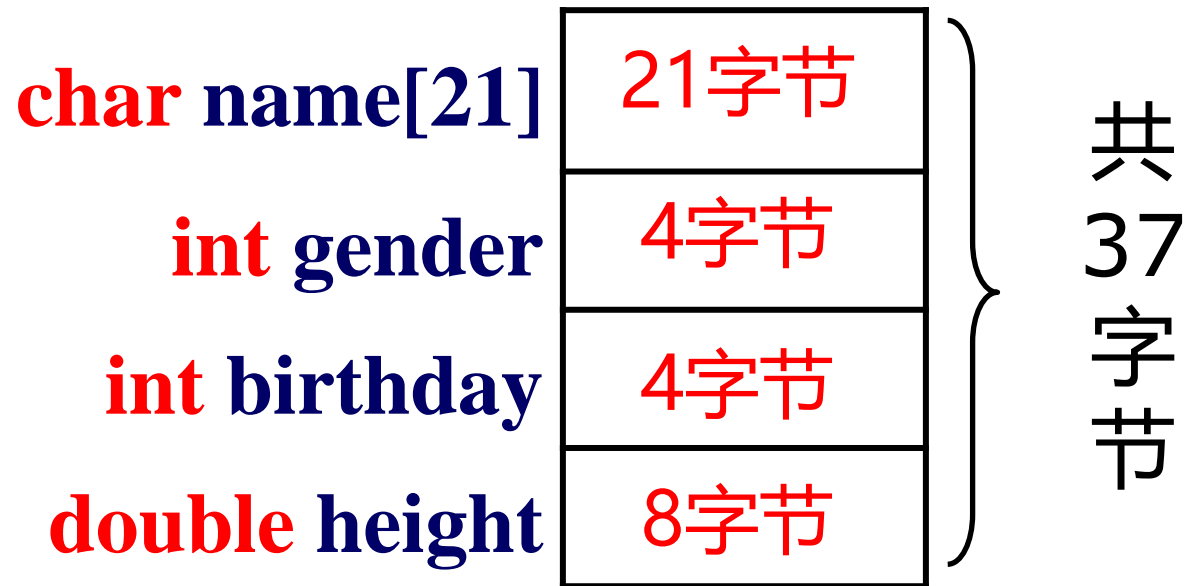
**student**



思考：struct student 的一个变量占用多少内存空间？

```
struct student {  
    char name[21];  
    int gender;  
    int birthday;  
    double height;  
};
```

一个struct student类型的变量  
在内存中占据37字节的连续空间





# 声明结构变量

- 可以在定义结构类型的同时声明变量

```
struct 结构标识符 {  
    类型名1 成员名1;  
    .....  
    类型名n 成员名n;  
} 变量列表;
```

- 类型名相同的多个成员可以一起声明

```
struct student stu1, stu2;
```

- 定义了名字为stu1和stu2的struct student类型的变量
- stu1和stu2变量分别具有结构类型声明的所有成员

# 声明结构变量

```
struct {  
    int number;  
    char name[LEN+1];  
    int gender;  
} stu1, stu2;
```

```
struct {  
    char name[LEN+1];  
    int number;  
    char gender;  
} employee1, employee2;
```

☞ 每个结构代表一个新的范围

- 这个范围里的任何名字不会与程序中的其他名字冲突
- 例如：上面的声明可以出现在同一个程序中
- 每个结构对它的成员构成一个单独的名字空间



# 初始化结构变量

☞ 结构声明可以包含初始化:

```
struct {  
    int number;  
    char name[LEN+1];  
    int on_hand;  
} partA = {528, "Disk drive", 10},  
partB = {914, "Printer cable", 5};
```

☞ 初始化之后的 partsA:

number	528
name	Disk drive
on_hand	10



# 初始化结构变量

## ❧ 结构初始化的规则与数组相似

- 用于初始化的表达式必须是常量
- 初始化的成员数可以比结构的成员数少
- 剩余成员的初始值为 0

## ❧ C99允许对结构变量进行指定初始化

- 例如：对上例中的partA进行指定初始化
- 原初始化式为 {528, "Disk drive", 10}, 改写后为:
- {.number = 528, .name = "Disk drive"}
- 句点和成员名的组合被称为指示符（注意区别于数组）
- 初始化没有说明的成员的默认值为0



# 对结构的操作

∞ 结构变量中各成员的访问方式为：**结构变量名 . 结构成员名**

- 例：stu1.name
- 表示结构变量stu1的name成员

  
**成员访问运算符**

∞ 显示partA的成员值的语句：

```
printf("Part number: %d\n", partA.number);
```

```
printf("Part name: %s\n", partA.name);
```

```
printf("Quantity on hand: %d\n", partA.on_hand);
```

# 对结构的操作

成员访问运算符的优先级高于所有的其他运算符

- 例如: `scanf("%d", &partA.number);`
  - `.` 运算符的优先级高于 `&`
  - 因此 `&` 计算的是 `partA.number` 的地址
- 结构成员可以出现在赋值表达式的左边
  - `part1.number = 258; // 修改part1的number`
- 结构成员可以作为增减表达式的操作数
  - `part1.on_hand++;` // 对成员 `on_hand` 进行自增
- 另一个主要的结构操作是赋值: `part2 = part1;`
  - 执行效果: 拷贝 `part1.number` 到 `part2.number`
  - 拷贝 `part1.name` 到 `part2.name`, 等等

# 对结构的操作

☞ 注意：数组不能使用 = 拷贝

- 但是当结构拷贝时，嵌入到结构中的数组可以拷贝！
- 一些程序员利用该特性创建假结构去封装将要拷贝的数组
  - `struct { int a[10]; } a1, a2;`
  - `a1 = a2;`

☞ “=” 运算符只能用于类型兼容的结构

- 同时声明的两个结构（如 partA 和 partB ）是兼容的
- 使用相同的结构类型名声明的结构也是兼容的

☞ 除了赋值，C 不提供对整个结构的操作！

- **特别注意：** == 和 != 运算符不能用于结构



# 结构类型

❧ 结构标识符用于标识某种特定类型的结构

- 声明一个名为part的结构标记:

```
struct part {  
    int number;  
    char name[LEN+1];  
}; // 注意花括号的后面必须有分号!
```

- 完成对part的声明之后就可以使用它来声明变量
  - **struct part** part1, part2;
- 注意: 不可省略struct关键字
  - 因为part不是类型名, 去掉struct则part没有任何意义
  - 因此结构标识符与程序中的其他变量名不会发生冲突



# 定义一个结构类型

- ❧ 定义结构类型名的另一种方式是使用 typedef
- ❧ 例如：定义一个名为Part的数据类型

```
typedef struct {  
    int number;  
    char name[LEN+1];  
} TPart;
```

- ❧ **TPart** 可以像内置类型一样使用
  - **TPart** **part1**, **part2**;
- ❧ 小结：命名一个结构的两种方法
  - 可以声明结构标记，或者使用 typedef



# 结构作为自变量和返回值

❧ 函数可以将结构作为自变量和返回值

❧ 包含结构自变量的函数示例：

```
void print_part( struct part p ) {  
    printf("Part number: %d\n", p.number);  
    printf("Part name: %s\n", p.name);  
    printf("Quantity on hand: %d\n", p.on_hand);  
}
```

❧ 函数 print\_part的调用方式： print\_part(**part1**);



# 结构作为自变量和返回值

采用part结构作为返回值类型的函数示例：

```
struct part build_part( int num, const char *nam, int oh )  
{  
    struct part p;  
    p.number = num;  
    strcpy(p.name, nam);  
    p.on_hand = oh;  
    return p;  
}
```

函数 build\_part的调用方式：

```
part1 = build_part( 528, "Disk drive", 10 );
```



# 结构作为自变量和返回值

❧ 传递一个结构给函数 & 从函数返回一个结构

- 都需要拷贝结构中的所有成员
- 为避免这种开销, 可采用传递和返回结构指针的方式

❧ 避免结构拷贝还有其他原因

- 例如: `<stdio.h>` 头文件定义了一个名为FILE的结构类型
- 每一个打开文件的函数都返回一个指向FILE结构的指针
- 每一个FILE结构存储一个打开文件的状态信息
  - 因而在程序中必须是唯一的
- 每个对文件进行操作的函数都需要一个FILE指针作为自变量

# 数组和结构的嵌套使用

- ❧ 结构和数组可以无约束地结合
  - 数组可以使用结构作为元素数据类型
  - 结构也可以包含数组和结构成员
- ❧ 把一个结构嵌套进另一个结构常常是有用的
  - 假设声明了如下的 `person_name` 结构

```
struct person_name {  
    char first[LEN+1];  
    char middle_initial;  
    char last[LEN+1];  
};
```

# 嵌套结构

∞ 可以用 person\_name 作为一个更大的结构的一部分

```
struct student {  
    struct person_name name;  
    int id, age;  
    char gender;  
} student1, student2;
```

∞ 访问student1的name成员的子成员

- 需要使用 . 操作符两次!

```
strcpy(student1.name.first, "Fred");
```

# 嵌套结构

---

## ☞ 允许结构嵌套的好处

- 将name定义为结构可以将姓名作为**数据单元**来处理
- 这样做可以减少数据复制操作次数
  - 例如: `display_name(student1.name);`

# 结构数组

## ❧ 结构数组：元素为结构的数组

- 是数组与结构相结合的最常用的形式之一
  - 这类数组可以作为简单的数据库
- 例如：结构数组inventory可以存储100个零件的信息
  - **struct part** inventory[**100**];
- 对结构数组元素的访问仍然是通过下标进行
  - print\_part( inventory[**0**] );
- 访问 part 结构的成员需要联合使用下标和成员选择
  - inventory[i].number = 883;
  - inventory[i].name[0] = '\0';



# 初始化结构数组

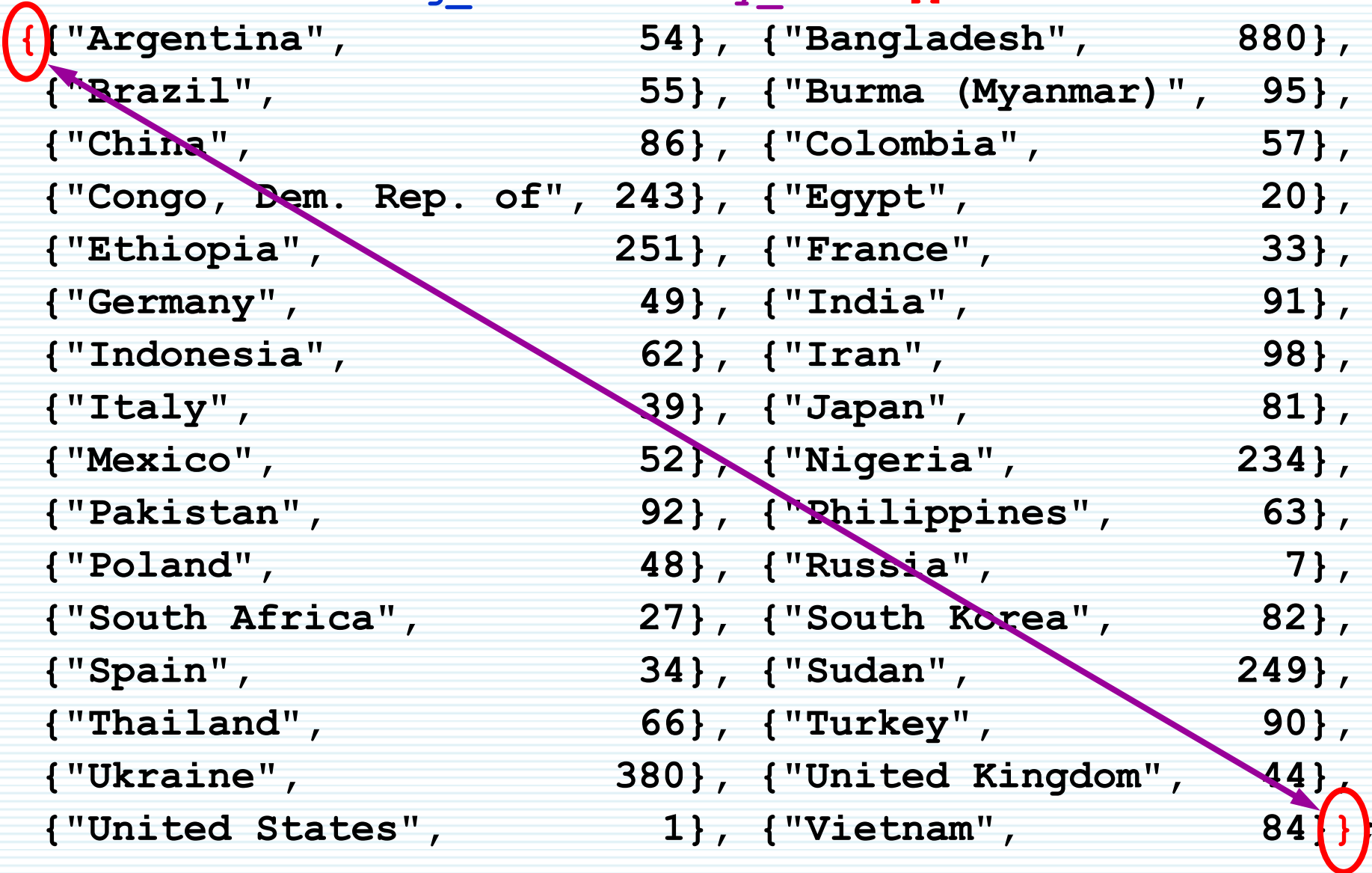
## ☞ 初始化**结构数组**与初始化**多维数组**的方法类似

- 每个结构都拥有自己的由大括号括起来初始化式
- 数组的初始化简单地在结构初始化式外围增加大括号
- 例如：初始化一个包含国家名称和区号的结构数组
  - 数组的元素为包含国家名称和区号的结构

```
struct dialing_code {  
    char *country;  
    int code;  
};
```

# 初始化结构数组

```
const struct dialing_code country_codes[] =  
{ {"Argentina",          54}, {"Bangladesh",          880},  
  {"Brazil",             55}, {"Burma (Myanmar)",      95},  
  {"China",              86}, {"Colombia",            57},  
  {"Congo, Dem. Rep. of", 243}, {"Egypt",          20},  
  {"Ethiopia",           251}, {"France",           33},  
  {"Germany",            49}, {"India",             91},  
  {"Indonesia",          62}, {"Iran",             98},  
  {"Italy",              39}, {"Japan",             81},  
  {"Mexico",             52}, {"Nigeria",          234},  
  {"Pakistan",           92}, {"Philippines",      63},  
  {"Poland",             48}, {"Russia",           7},  
  {"South Africa",       27}, {"South Korea",     82},  
  {"Spain",             34}, {"Sudan",          249},  
  {"Thailand",           66}, {"Turkey",         90},  
  {"Ukraine",           380}, {"United Kingdom", 44},  
  {"United States",      1}, {"Vietnam",         84} }
```



# 知识回顾：动态内存分配（C语言）

# C语言知识回顾：动态内存分配

☞ 声明一个指针变量不会自动分配内存

- 在对指针执行间接访问前必须对其初始化
  - 或者使它指向预先分配好的内存
  - 或者为指针所指向的目标**动态分配内存**

☞ 动态内存分配函数：**malloc()**

- 函数原型：**void \*malloc (unsigned int size);**
  - 在操作系统管理的内存动态存储区开辟一块空间
  - 分配一段长度为**size个字节**的连续空间
  - 若分配成功则返回一个指向该区域起始地址的指针
  - 否则（例如内存空间不够）返回空指针（NULL）
  - 返回的**指针类型**需由程序员指定！



# void的含义

∞ void关键字表达的意思是“无类型”

- void只起限制函数和指针行为的作用
- 不能声明或定义void类型的变量
  - 因为当对象类型不确定时，则它的大小也是未确定的
- 例如：**void a;** *// illegal use of type 'void'*

∞ void \* 表示“无类型指针”

- void\*型指针被用于如下两种情况
  - 对象的确切类型未知
  - 在特定环境下对象的类型会发生变化
- 任何非const类型的指针都可以被赋值给void\*型的指针

# 指针的强制类型转换

- ☞ 若指针p1和p2的类型相同，则可以直接相互赋值
- ☞ 若p1和p2指向不同的数据类型？
  - 则必须使用强制类型转换运算符
  - 把赋值运算符右边的指针类型转换为左边指针的类型
- ☞ 例如：**float \*p1; int \*p2; p1 = p2;**  
  
Error: '=' : cannot convert from 'int \*' to 'float \*'
  - 须改为：**p1 = (float \*) p2;**

# 指针的强制类型转换

☞ 任何类型的指针都可以赋值给void \* (无需类型转换)

- 例如: **void \*p1; int \*p2; p1 = p2;**

☞ 反之则不正确:

- 例如: **void \*p1; int \*p2; p2 = p1;**

Error: '=' : cannot convert from 'void \*' to 'int \*'

# void关键字的使用

☞ 若函数参数可以是任意类型指针，则应声明为void \*

- 内存操作函数memcpy和memset的函数原型分别为：

```
void * memcpy (void *dest, const void *src, size_t len);
```

```
void * memset ( void * buffer, int c, size_t num );
```

- 因此任何类型的指针都可以传入memcpy和memset中



# void关键字的使用

☞ 若函数参数可以是任意类型指针，则应声明为void \*

- 因此任何类型的指针都可以传入memcpy和memset中
  - `int A[100], B[100];`
  - `memset ( A, 0, 100*sizeof(int) );`
  - 运行结果：将数组A清0
  - `memcpy ( A, B, 100*sizeof(int) );`
  - 将B的内容拷贝到A指向的空间

# 常用动态内存分配函数： calloc函数

## ∞ 函数原型

**void \*calloc (unsigned int n, unsigned int size);**

## ∞ 函数功能

- 在内存**动态存储区**分配n个长度为size 个字节的连续空间
- 如果分配成功，返回指向该区域起始地址的指针
- 如果分配失败，返回空指针（NULL）

# 常用动态内存分配函数： realloc函数

## ∞ 函数原型

**void \*realloc (void \*p, unsigned int size);**

## ∞ 函数功能

- 对指针p所指向的存储空间进行重新分配
  - 将p指向的空间大小调整为size个字节
  - 并将原存储空间存放的数据拷贝到新分配的存储空间
- 如果分配成功，返回一个指向新存储空间起始地址的指针
- 如果分配失败，则返回空指针



# 常用动态内存分配函数： free函数

- ❧ 函数原型： void **free**(**void** \*p);
- ❧ 功能： 释放指针p指向的存储空间， free函数无返回值
- ❧ 使用动态内存分配函数需： #include "**stdlib.h**"
- ❧ 良好的编程习惯
  - 最好在**同一个函数内**动态分配和释放存储空间
  - 最好在定义指针时将指针初始化为**NULL**
  - 最好在释放指针后也将指针赋值为**NULL**
  - 这样便于使用p==NULL语句判断指针是否有效



