



MCS-51单片机

-- C51基础

陈茜茹

Email:vickychenqian@163.com



主要内容

- C51与标准C
- C51的数据类型和运算量
- C51的运算符
- 程序结构
- 函数
- 其它数据类型



C51与标准C

- C语言是一种编译型程序设计语言，它兼顾了多种高级语言的特点，并具备汇编语言的功能。针对8051的C语言日趋成熟，成为了专业化的实用高级语言。很多硬件开发都用C语言编程，如：各种单片机、DSP、ARM等。



C51与标准C的区别

(1) C51中定义的库函数和标准C语言定义的**库函数不同**。标准的C语言定义的库函数是按通用微型计算机来定义的，而C51中的库函数是按MCS-51单片机相应情况来定义的；

(2) C51中的数据类型与标准C的数据类型也有一定的区别，在C51中还增加了几种针对MCS-51单片机**特有的数据类型**；

(3) C51变量的存储模式与标准C中变量的**存储模式不一样**，C51中变量的存储模式是与MCS-51单片机的存储器紧密相关；

(4) C51与标准C的输入输出处理不一样，C51中的输入输出是通过MCS-51串行口来完成的，输入输出指令执行前必须要对串行口进行初始化；

(5) C51与标准C在函数使用方面也有一定的区别，C51中有**专门的中断函数**。



C51的数据类型

基本数据类型

类型	符号	关键字	所占位数	数的表示范围
整型	有	(signed) int	16	-32768~32767
		(signed) short	16	-32768~32767
		(signed) long	32	-2147483648~2147483647
	无	unsigned int	16	0~65535
		unsigned short int	16	0~65535
		unsigned long int	32	0~4294967295
实型	有	float	32	3.4e-38~3.4e38
	有	double	64	1.7e-308~1.7e308
字符型	有	char	8	-128~127
	无	unsigned char	8	0~255



基本类型: *

- 长度根据其所指向的变量有所变化, 1 ~ 3 字节
- 存放的是另外一个数据的地址
- 定义方法: `unsigned int *da;`



C51的数据类型扩充定义

sfr: 特殊功能寄存器定义

sfr16: sfr的16位数据定义

sbit: 可位寻址字节或sfr中位的定义

bit: 位变量定义

例: `sfr SCON = 0X98;`

`sfr16 T2 = 0xCC;`

`sbit OV = PSW^2;`





定义变量例:

数据类型 变量名

unsigned int var1;

bit flags;

unsigned char vector[10];





用 typedef 或 #define 定义数据类型别名

例1:

```
typedef unsigned int uint;
```

```
uint var;
```

例2:

```
#define uchar unsigned char;
```

```
uchar temp;
```



C51数据的存储类型

说明变量所处单片机的存储区域

名	存储空间位置	长度	数据范围
data	直接寻址片内RAM低128B	8位	0 ~ 255
bdata	可位寻址片内RAM (20-2FH)	1位	0/1
idata	间接寻址片全部内RAM	8位	0 ~ 255
pdata	片外RAM低256B	8位	0 ~ 255
xdata	片外RAM全部64K	16位	0 ~ 65535
code	程序ROM64K	16位	0 ~ 65535





定义变量例:

数据类型 [存储类型] 变量名

```
unsigned int data var1;
```

```
bit bdata flags;
```

```
unsigned char code vector[10];
```



C51数据的存储模式

不同的存储模式对变量默认的存储器类型不一样

1. SMALL模式：小编译模式。编译时，函数参数和变量被默认在片内RAM中，**存储器类型为data**。
2. COMPACT模式：紧凑编译模式。编译时，函数参数和变量被默认在片外RAM的低256字节空间，存储器类型为pdata。
3. LARGE模式：大编译模式。编译时，函数参数和变量被默认在片外RAM的64K字节空间，存储器类型为xdata。

存储模式的指定通过#pragma预处理命令来实现。**如果没有指定，则系统都隐含为SMALL模式**。例： #pragma large



C-51数据的存储种类

变量在程序执行过程中的作用范围

- 1. auto:** 作用范围在定义它的函数体或复合语句内部。**所有变量默认为自动(auto)变量。**
- 2. extern:** 在一个函数体内，要使用一个已在该函数体外或别的程序中定义过的外部变量时，该变量在该函数体内要用**extern说明**。
- 3. static:** 分为内部/局部静态变量和外部/全局静态变量。在函数体内部定义的静态变量为内部静态变量，它**在对应的函数体内有效**，一直存在，但在函数体外不可见，当离开函数时值不被改变。外部静态变量作用于**仅限于变量被定义的文件中**。其它文件中即使用extern声明也没法使用它。
- 4. register:** 它定义的变量存放在CPU内部的寄存器中，处理速度快，但数目少。C51编译器编译时能自动识别程序中使用频率最高的变量，并自动将其作为寄存器变量，用户可以无需专门声明。





定义变量例:

[存储种类] 数据类型 [存储类型] 变量名

```
extern unsigned int data var1;
```

```
static bit data flags;
```

```
auto unsigned char code vector[10];
```



C51的包含的头文件

通常有: `reg51.h` `reg52.h` `math.h`
`ctype.h` `stdio.h` `stdlib.h`
`absacc.h`

常用有: `reg51.h` `reg52.h`

（定义特殊功能寄存器和位寄存器）；

`math.h` （定义常用数学运算）；



C-51的运算符

与C语言基本相同:

{		+	-	*	/	% (加 减 乘 除 取余)
{		>	>=	<	<=	(大于 大于等于 小于 小于等于)
{		==	!=			(测试等于 测试不等于)
		&&		!		(逻辑与 逻辑或 逻辑非)
{		>>	<<			(位右移 位左移)
{		&				(按位与 按位或)
{		^	~			(按位异或 按位取反)
		,				逗号运算符
		?:				条件运算符
		* &				(指针运算符 取址运算符)



算术表达式: ++/--

- $++i$: i 自增1后再参与其他运算;
- $--i$: i 自减1后再参与其他运算;
- $i++$: i 参与运算后, i 的值再自增1;
- $i--$: i 参与运算后, i 的值再自减1。



位操作运算符：移位

- 例如：若 $a = ABH = 10101011B$ ，则：
 $a = a \ll 2$ ，将 a 值左移2位，其结果为
 $10101100B = ACH$;
- $a = a \gg 2$ ，将 a 值右移2位，其结果为
 $00101010B = 2AH$ 。



程序结构

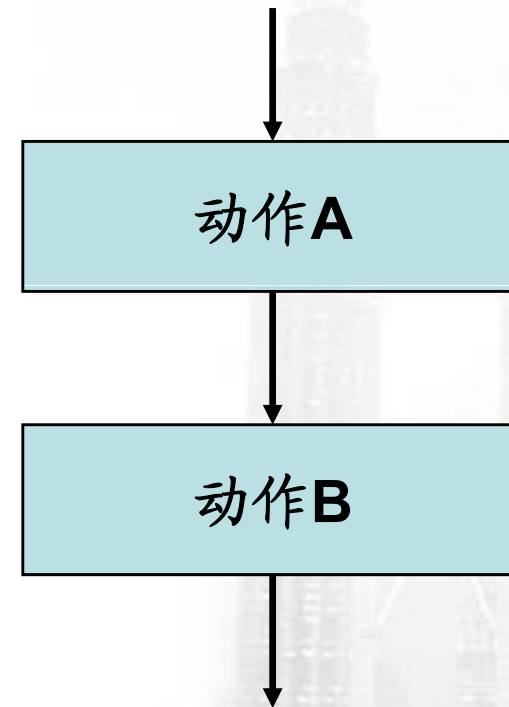


- 顺序结构
- 选择结构
- 循环结构

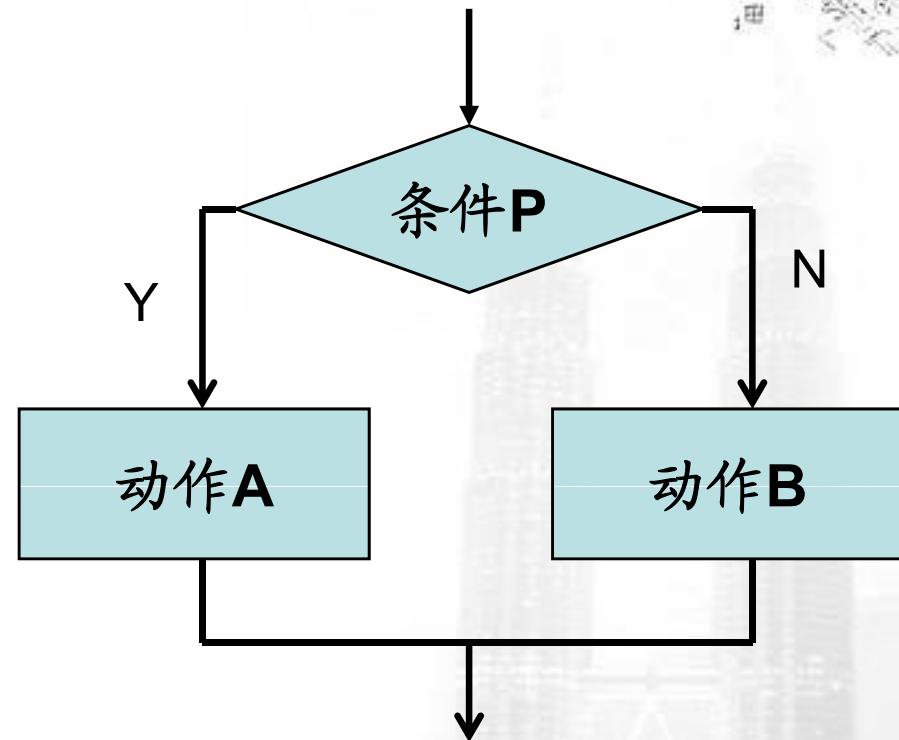
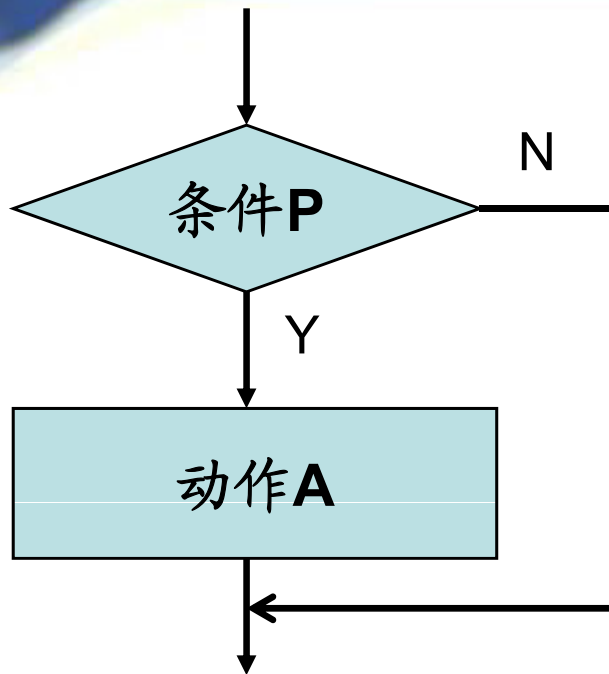


顺序结构

- 适当运用表达式语句就能设计出具有某特定功能的顺序结构C51程序
- 程序只由低地址向高地址顺序执行指令代码
- 设计方法简单
- 无法构成复杂程序



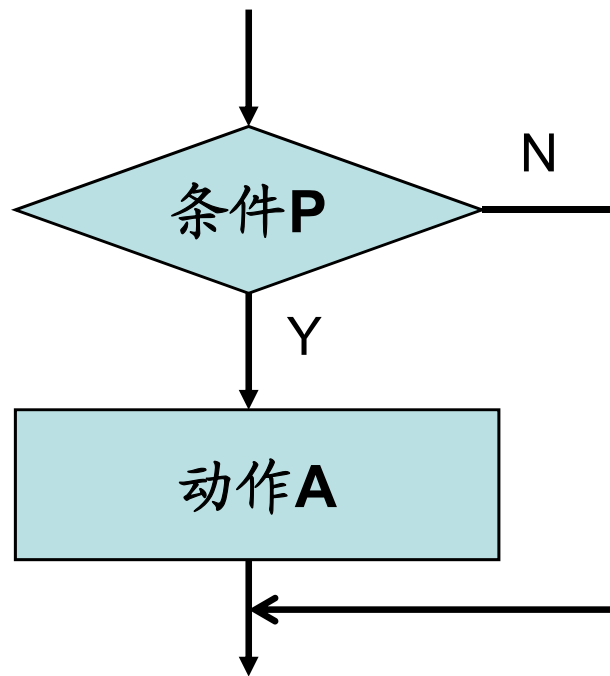
选择结构



- 决策能力
- 也称为分支结构
- 根据判断条件**P**的成立与否，选择执行其中的一路分支



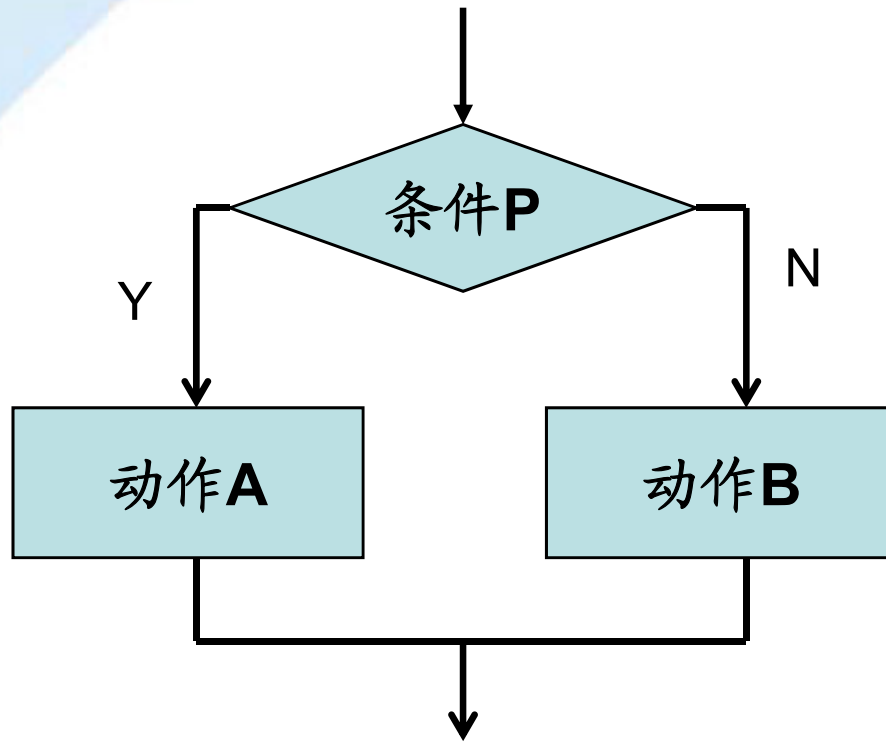
选择结构：基本if结构



```
....  
If(条件P为真)  
{  
    动作A;  
}  
...
```



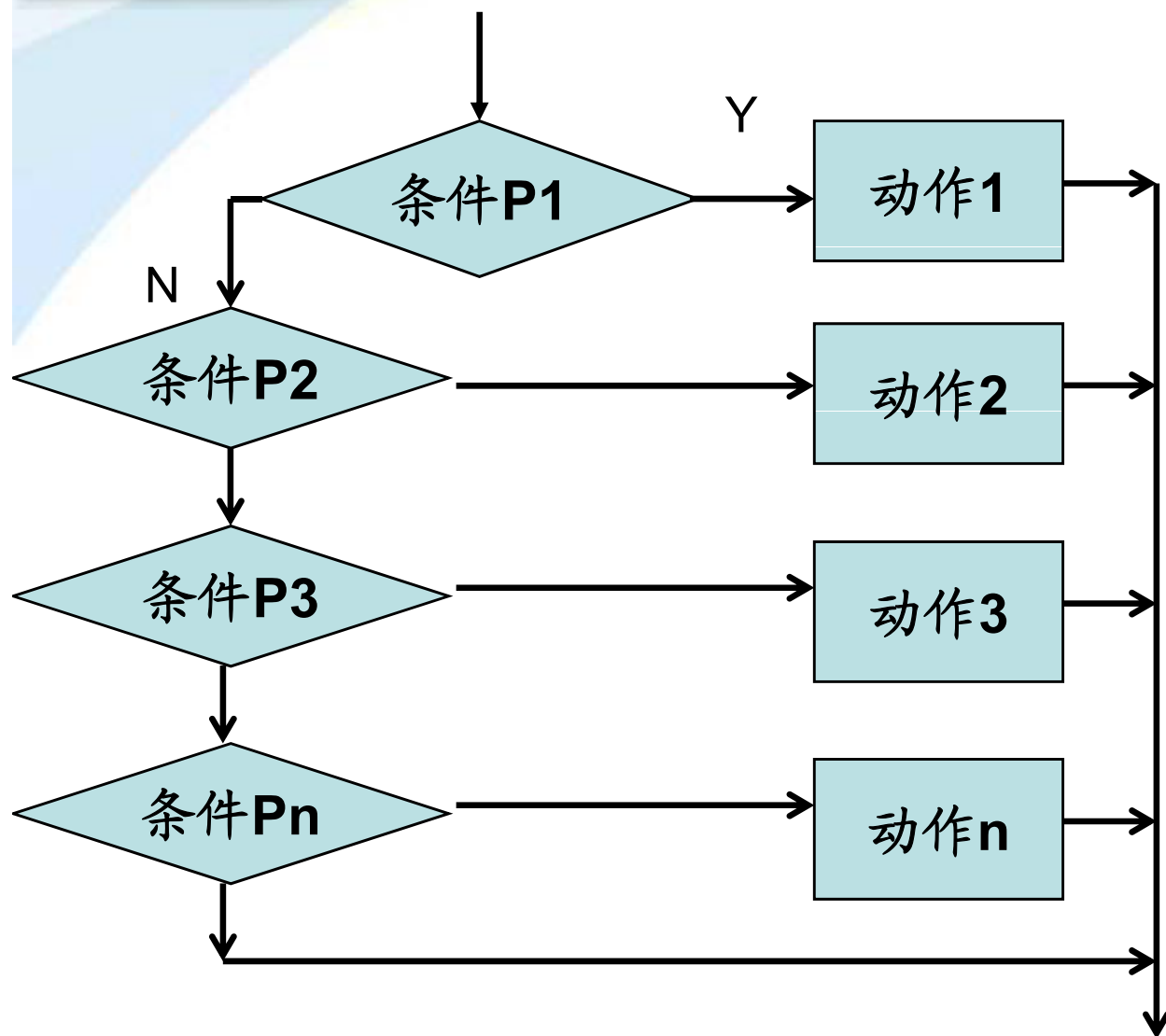
选择结构：if-else结构



```
....  
If(条件P为真)  
{  
    动作A;  
}  
else  
{  
    动作B;  
}  
...
```



选择结构：if-else if结构



```
....  
If(条件P1)  
{  
    动作1;  
}  
else if(条件P2)  
{  
    动作2;  
}  
else if(条件P3)  
...  
...
```



else到底和哪个if配对呢？

```
if(x==0)
if(y==0)error();
else{
    //program code
}
```

else始终与同一括号内最近的未匹配的if语句结合。



关于程序中的分界符 ‘{’ 和 ‘}’

提倡的风格	不提倡的风格
<pre>void Function(int x) { //program code }</pre>	<pre>void Function(int x){ //program code }</pre>
<pre>if(condition) { //program code } else { //program code }</pre>	<pre>if(condition){ //program code }else{ //program code } 或： if(condition) //program code else //program code 或： If(width<height)dosomething();</pre>



提倡的风格

```
for(initialization;condition;update)
{
    //program code
}
```

```
while(condition)
{
    //program code
}
```

```
do
{
    //program code
}
while(condition);
```

不提倡的风格

```
for(initialization;condition;update){
//program code
}
```

```
while(condition){
//program code
}
```

```
do{
//program code
}
while(condition);
```



if语句后面的分号

```
if(p!=0);  
    Fun();
```

Fun函数什么时候调用？

仅由 ‘; ’ 形成空语句。

例： unsigned int i=1000;
while(i--);



使用if语句的其他注意事项

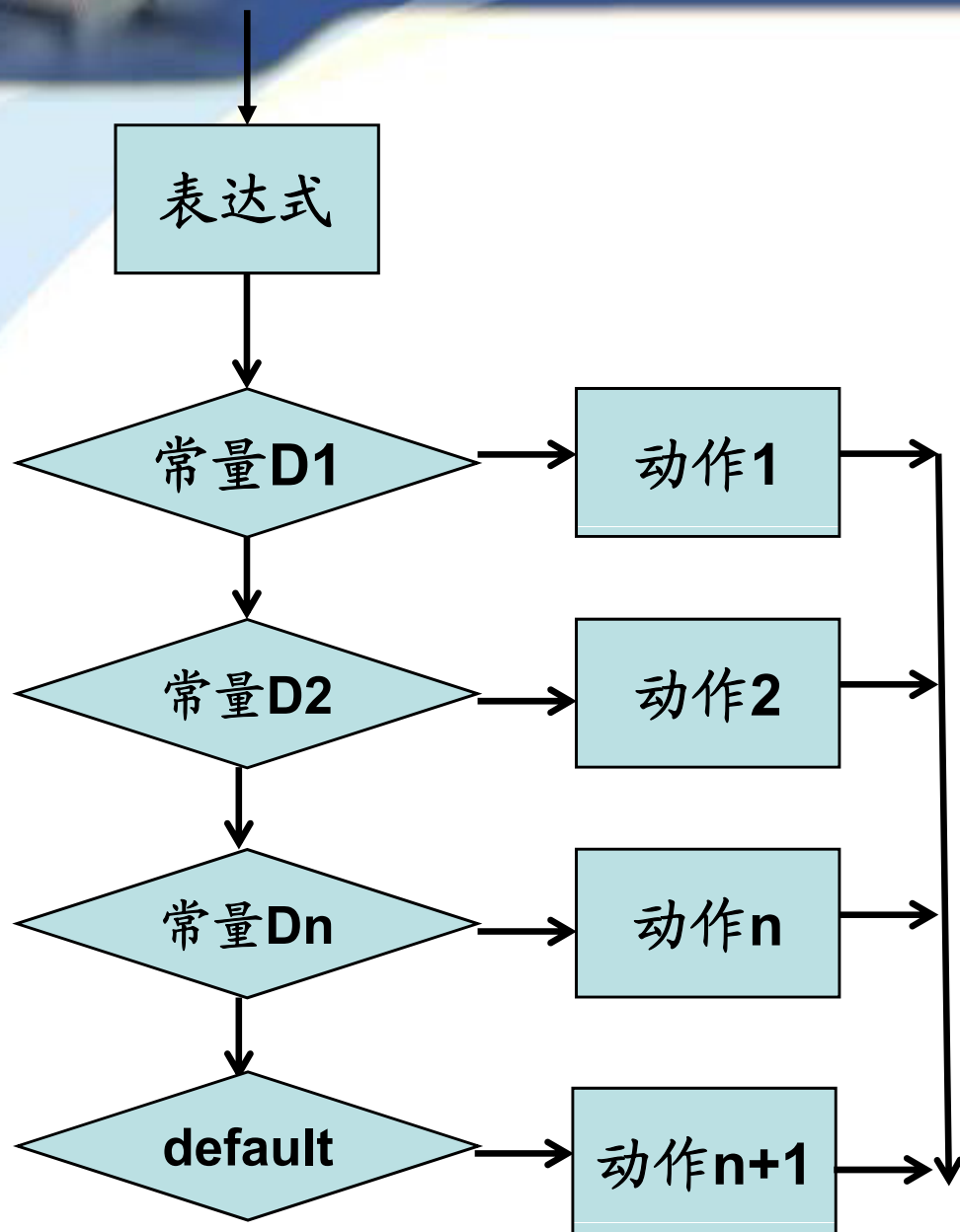
- 先处理正常情况，再处理异常情况

把正常情况的处理放到**if**后面，把异常情况的处理放到**else**后面。因为**if**语句总是需要做判断，而正常情况一般比异常情况发生的概率更大，如果把执行概率更大的语句放到后面，意味着**if**语句将进行多次无谓的比较。

- 确保**if**和**else**子句没有弄反



选择结构: switch



Switch(表达式)

{

case 常量1:
 动作1;
 break;

case 常量2:
 动作2;
 break;

case 常量n:
 动作n;
 break;

default:
 动作n+1;
 break;

}



使用规则

- 每个**case**语句的结尾不要忘了加**break**，否则将导致多个分支重叠。（除非有意使多个分支重叠）
- 最后必须使用**default**语句。即使程序不需要，也应该保留语句：

default:

break;

这样做并非画蛇添足，可以避免让人误以为你忘了**default**处理



case关键字后面的值有什么要求?

- 0.1? -0.1? -1? 0.1+0.9? 1+2? 3/2?
‘A’ ? “A” ? 变量i?

case后面只能是整型或字符型的常量或常量表达式。



case语句的排列顺序

- 按字母或数字
- 正常情况放前面，异常情况放后面
- 按执行的频率



选择结构：试一试

题目：对正常学生成绩数据进行分类计数。
分别计算一群数据中**0 ~ 9**，**10 ~ 19**，**20 ~ 29...**，**90 ~ 99**之间的元素个数。



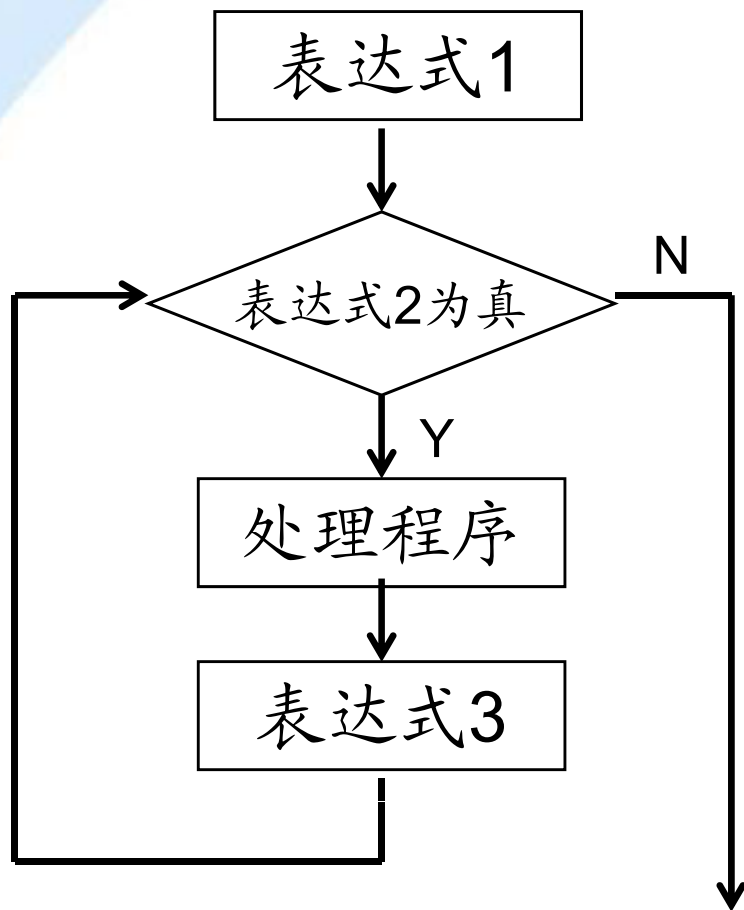
循环结构

- 特点：在给定条件成立时，反复执行某程序段，直到条件不成立时为止
- for语句、while语句、do-while语句



循环结构：for

```
for(表达式1; 表达式2; 表达式3)
{
    处理程序;
}
```

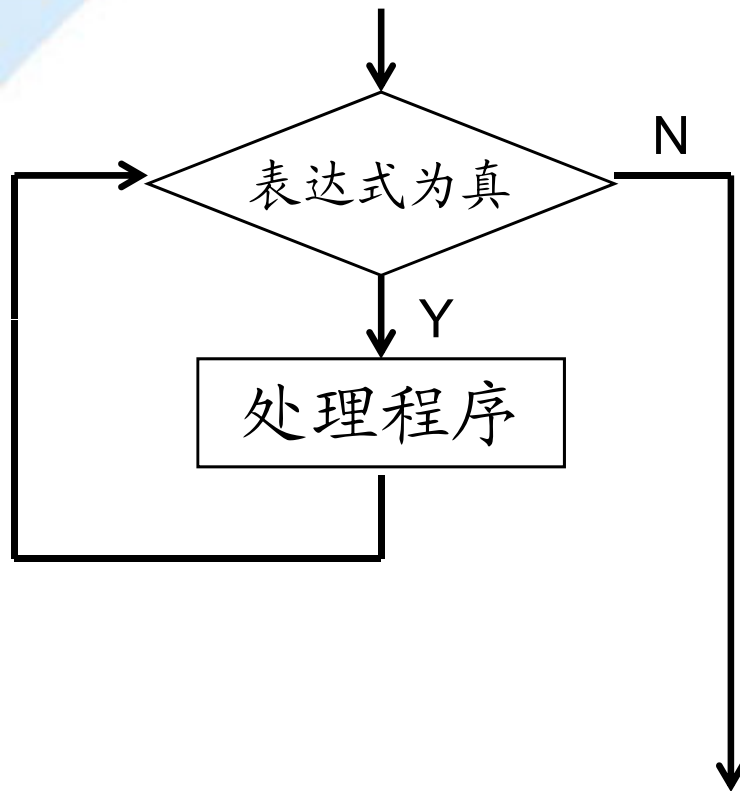


注意：

- ① 处理程序可以为空操作;
- ② for语句的各表达式都可以省，而分号不能省，在省略各表达式时要特别小心分析，防止造成无限死循环。



循环结构：while

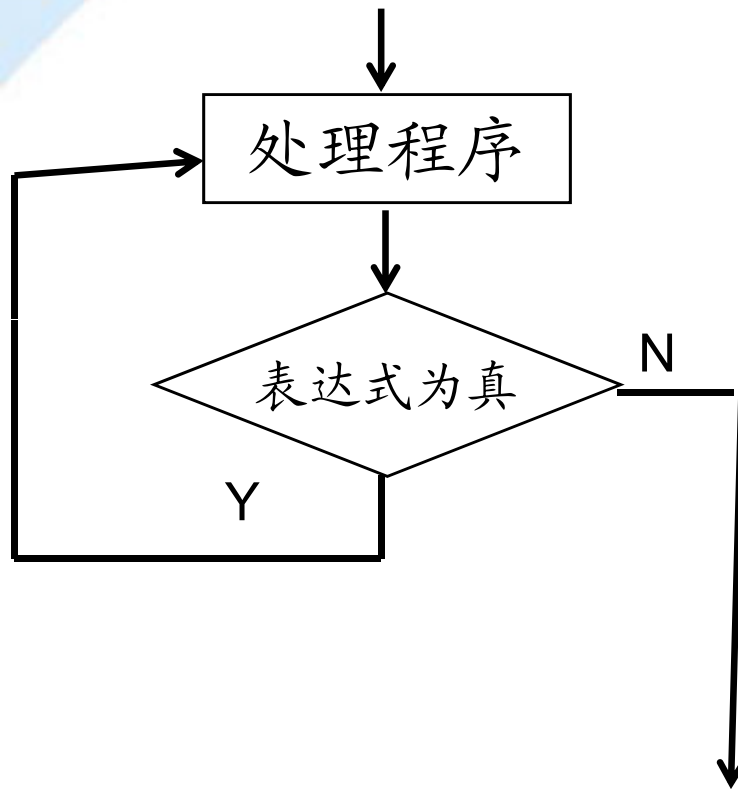


```
while(表达式)
{
    处理程序;
}
```

处理程序;



循环结构：do-while



```
do  
{  
    处理程序;  
}  
while(表达式);
```



循环结构：想一想

单片机是不能处于停机状态的，如何编写这个程序？

```
While(1);
```



循环语句的注意点

- 在多层循环中，如果有可能，应当将最长的循环放到内层，最短的循环放到最外层，以减少CPU跨切循环层的次数。

长循环在最内层，效率高

```
for(col=0;col<5;col++)
{
    for(row=0;row<100;row++)
    {
        sum=sum+a[row][col];
    }
}
```

长循环在最外层，效率低

```
for(row=0;row<100;row++)
{
    for(col=0;col<5;col++)
    {
        sum=sum+a[row][col];
    }
}
```



- **for**语句的循环控制变量的取值采用“半开半闭区间”。“半开半闭区间”写法和“闭区间”写法功能相同，但前者更加直观。

半开半闭区间	闭区间
<pre>for(n=0;n<10;n++) { ... }</pre>	<pre>for(n=0;n<=9;n++) { ... }</pre>



- 不能在**for**循环体内修改循环变量，防止循环失控。

```
for(n=0;n<10;n++)
```

```
{
```

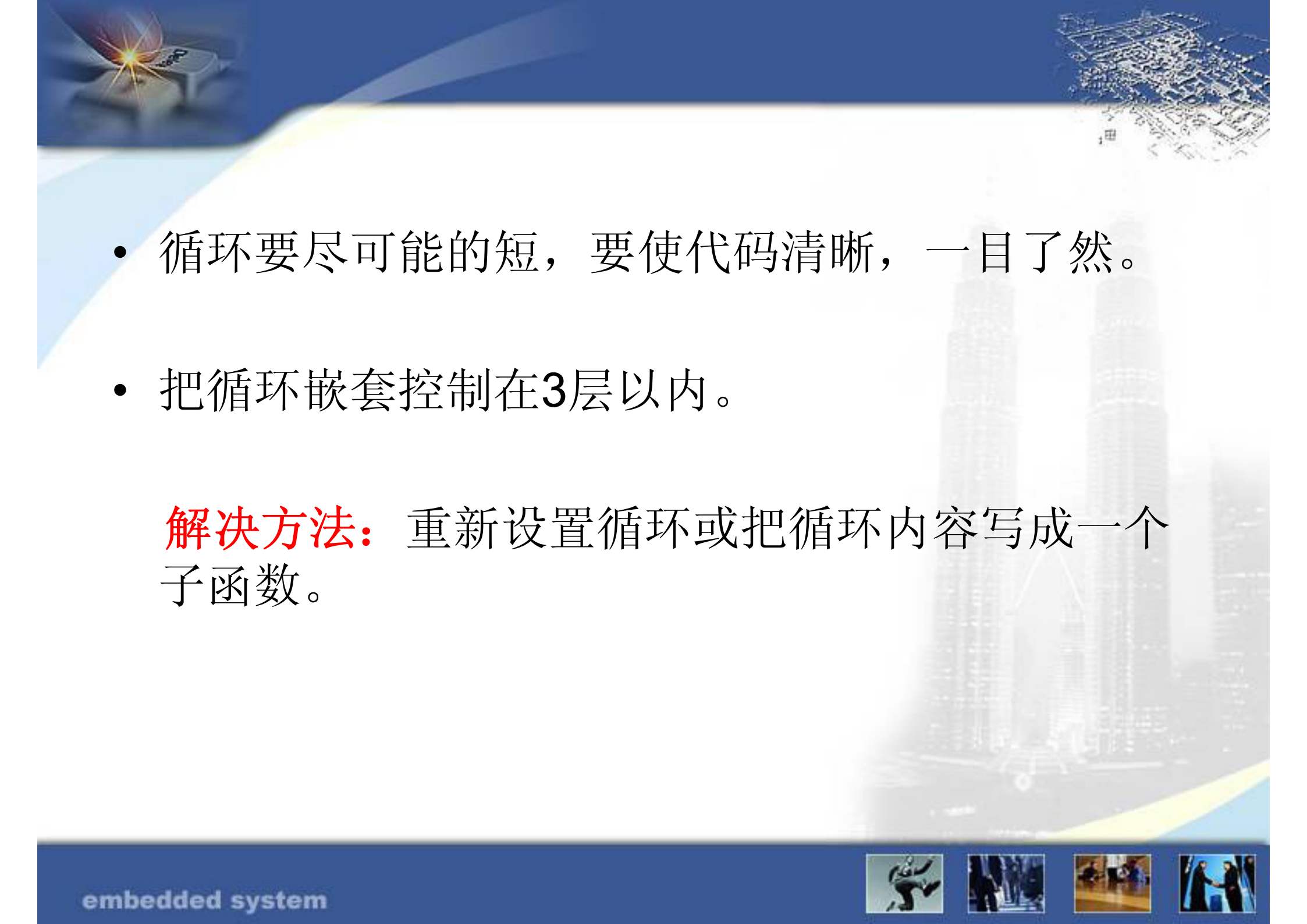
```
...
```

```
n=8; //不可，很可能违背了你的原意
```

```
...
```

```
}
```



- 
- 循环要尽可能的短，要使代码清晰，一目了然。
 - 把循环嵌套控制在3层以内。

解决方法： 重新设置循环或把循环内容写成一个子函数。



break与continue的区别

- break用于终止**本层循环**
- continue用于终止**本次（本轮）循环**。当代码执行到continue时，本轮循环终止，进入下一轮循环



Main 函数

格式: `void main()`

特点: 无返回值, 无参。

任何一个C程序有且仅有一个main函数, 它是整个程序开始执行的入口。

例: `void main()`

```
{  
    总程序从这里开始执行;  
    其他语句;  
}
```



函数定义

函数类型 函数名（形式参数表）

{

局部变量定义

函数体

}

其实就是将功能相对能成为一个整体的代码打包，
通过一些数据接口和外部通信。



中断服务程序

```
函数名 ( ) interrupt m using n  
{  
    函数内部实现 ....  
}
```



修饰符interrupt m

m的取值为0~31，对应的中断情况如下：

0——外部中断0

1——定时/计数器T0

2——外部中断1

3——定时/计数器T1

4——串行口中断

5——定时/计数器T2

其它值预留。



修饰符using n

用于指定本函数内部使用的工作寄存器组，其中n的取值为0~3，表示寄存器组号。

using n修饰符**不能用于有返回值的函数**，因为C51函数的返回值是放在寄存器中的。如寄存器组改变了，返回值就会出错。



MCS-51 中断函数注意事项

- (1) 中断函数**不能进行参数传递**，如果中断函数中包含任何参数声明都将导致编译出错。
- (2) 中断函数**没有返回值**，如果企图定义一个返回值将得不到正确的结果，建议在定义中断函数时将其**定义为void类型**，以明确说明没有返回值。
- (3) 在任何情况下都**不能直接调用中断函数**，否则会产生编译错误。
- (4) 如果在中断函数中调用了其它函数，则被调用函数所使用的寄存器必须与中断函数相同。否则会产生不正确的结果。
- (5) 中断函数**最好写在文件的尾部**，并且禁止使用extern存储类型说明。防止其它程序调用。



函数的调用

函数调用的一般形式如下：

函数名（实参列表）；

对于有参数的函数调用，若实参列表包含多个实参，则各个实参之间用逗号隔开。



函数的声明

在C51中，函数声明一般形式如下：

`[extern]` 函数类型 函数名（形式参数表）；



定义和声明的区别

声明有两重含义：

1、告诉编译器，这个名字已经匹配到一块内存上了。

下面的代码用到的变量或对象是在别处定义的。

例： `extern int i;`

2、告诉编译器，这个名字我先预定了，别的地方再也不能用它作为变量名或对象名。

例： `void Fun(int i,char a);`



数组

一维数组只有一个下标，定义的形式如下：

数据类型说明符 数组名[常量表达式];

例：int a[10];



基本类型: *

- 长度根据其所指向的变量有所变化, 1 ~ 3字节
- 存放的是另外一个数据的地址
- 定义方法: `unsigned int *da;`



例1:

```
#include<stdio.h>
```

```
int main(void)
```

```
{
```

```
    int a=1,b=2,t;
```

```
    int *p1,*p2;
```

```
    p1=&a;p2=&b;
```

```
    printf("a=%d,b=%d,*p1=%d,*p2=%d\n",a,b,*p1,*p2);
```

```
    t=*p1;*p1=*p2;*p2=t;
```

```
    printf("a=%d,b=%d,*p1=%d,*p2=%d\n",a,b,*p1,*p2);
```

```
    return 0;
```

```
}
```



例2:

```
#include<stdio.h>
```

```
int main(void)
```

```
{
```

```
    int a=1,b=2;
```

```
    int *p1=&a,*p2=&b,*pt;
```

```
    printf("a=%d,b=%d,*p1=%d,*p2=%d\n",a,b,*p1,*p2);
```

```
    pt=p1;p1=p2;p2=pt;
```

```
    printf("a=%d,b=%d,*p1=%d,*p2=%d\n",a,b,*p1,*p2);
```

```
    return 0;
```

```
}
```



结构

结构是一种组合数据类型，它是将若干个不同类型的变量结合在一起而形成的一种数据的集合体。组成该集合体的各个变量称为结构元素或成员。整个集合体使用一个单独的结构变量名。

```
struct date
{
int year;
char month,day;
}
struct date d1,d2;
```

```
struct date
{
int year;
char month,day;
}d1,d2;
```



结构变量的引用

结构元素的引用一般格式如下:

结构变量名.结构元素名

或

结构变量名->结构元素名

例:

d1.year;

d2->day;



联合

结构中定义的各个变量在内存中占用不同的内存单元，在位置上是分开的，而联合中定义的各个变量在内存中都是从同一个地址开始存放，即采用了所谓的“覆盖技术”。这种技术可使不同的变量分时使用同一内存空间，提高内存的利用效率。

```
union data
```

```
{
```

```
float i;
```

```
int j;
```

```
char k;
```

```
}
```

```
union data a,b,c;
```

```
union data
```

```
{
```

```
float i;
```

```
int j;
```

```
char k;
```

```
}data a,b,c;
```



联合变量的引用

联合元素的引用一般格式如下：

联合变量名.联合元素名

或

联合变量名->联合元素名

例：

a.i;

b->j;



结构与联合的区别

定义时，结构与联合的区别只是将关键字由struct换成union。

注意：在内存的分配上两者完全不同。结构变量占用的内存长度是其中各个元素所占用的内存长度的总和；而联合变量所占用的内存长度是其中各元素的长度的最大值。结构变量中的各个元素可以同时进行访问，联合变量中的各个元素在一个时刻只能对一个进行访问。



枚举

枚举数据类型是一个有名字的某些整型常量的集合。这些整型常量是该类型变量可取的所有合法值。即该类型变量只能是花括号中的任何一个值。枚举定义时应当列出该类型变量的所有可取值。

```
enum week {Sun,Mon,Tue,Wed,Thu,Fri,Sat};
```

```
enum week d1;
```

```
enum week {Sun,Mon,Tue,Wed,Thu,Fri,Sat} d1;
```

