控制

为了实现有条件的代码执行，例如选择分支、循环语句等，

条件码

```
CF:进位标志，最近的操作使得最高位产生进位则为1
ZF:零标志，最近操作得出结果为0则该位为1
SF:符号标志，最近操作得到结果为负数则为1
OF:溢出标志，最近操作导致补码溢出则为1
```

最近的操作，即该条指令产生的结果影响条件寄存器的内容，执行下一条指令后会覆盖上一次的结果。

例如 `t=a+b`

条件码	结果	意义	值
CF	(unsigned)t < (unsigned)a	无符号溢出	1
ZF	(t==0)	0	1
SF	(t<0)	负数	1
OF	(a<0==b<0) && (t<0 != a<0)	有符号溢出	1

example：

```
unsigned a = 255;
unsigned b = 1;
unsigned t = a+b;
此时CF=1
```

一些指令及其对条件码的影响

指令	Effect
LEA	不改变条件码
XOR	CF=0、OF=0
SAL SHL SAR SHR	CF=最后被移除的位、OF=0
INC DEC	OF=1、ZF=1、CF不变

只设置条件码而不改变寄存器的两个命令：

指令		基于	描述
CMP	S_1，S_2	S_2-S_1	比较
cmpb			比较字节
cmpw			比较字
cmpl			比较双字
cmpq			比较四字
TEST	S_1，S_2	$S_1 \& S_2$	测试
testb			测试字节
testw			测试字
testl			测试双字
testq			测试四字

图 3-13　比较和测试指令。这些指令不修改任何寄存器的值，只设置条件码

这里CMP和SUB的行为一致，即比较的操作是通过两数之差来判断的

当两个操作数相等，OF=1

同理 TEST和AND的行为一致

使用条件码

```
1.根据条件码组合，将一个字节设置为0或1
2.根据条件跳转到程序的某个其他部分
3.有条件地传送数据
```

指令	同义名	效果	设置条件
sete D	setz	D ← ZF	相等/零
setne D	setnz	D ← ~ZF	不等/非零
sets D		D ← SF	负数
setns D		D ← ~SF	非负数
setg D	setnle	D ← ~(SF ^ OF) & ~ZF	大于（有符号>）
setge D	setnl	D ← ~(SF ^ OF)	大于等于（有符号>=）
setl D	setnge	D ← SF ^ OF	小于（有符号<）
setle D	setng	D ← (SF ^ OF) | ZF	小于等于（有符号<=）
seta D	setnbe	D ← ~CF & ~ZF	超过（无符号>）
setae D	setnb	D ← ~CF	超过或相等（无符号>=）
setb D	setnae	D ← CF	低于（无符号<）
setbe D	setna	D ← CF | ZF	低于或相等（无符号<=）

图 3-14 SET 指令。每条指令根据条件码的某种组合，将一个字节设置为 0 或者 1。有些指令有“同义名”，也就是同一条机器指令有别的名字

比较相等的情况：

```
int cmp(long a, long b)
{
    return (a==b);
}

# a in %rdi, b in %rsi
cmp %rsi, %rdi //a-b 相等则返回0
sete %ai //equal 若a和b相等 ZF=0 %al=0
movzbl %al, %eax
ret
```

比较小于的情况：

```
int cmp(long a, long b)
{
    return (a < b);
}

# a in %rdi, b in %rsi
cmp %rsi, %rdi //a-b
setl %ai //less
movzbl %al, %eax
ret
```

以`t=a-b`为例

1. `a<b` t<0 SF=1

2. `a>b` t>0 SF=0

3. `a<b` a=-2 b=127 t=127>0 SF=0 OF=1

4. `a>b` a=1 b=-128 t=127<0 SF=1 OF=1

跳转指令

配合使用的跳转指令

指令	同义名	跳转条件	描述
jmp *Label*		1	直接跳转
jmp **Operand*		1	间接跳转
je *Label*	jz	ZF	相等/零
jne *Label*	jnz	~ZF	不相等/非零
js *Label*		SF	负数
jns *Label*		~SF	非负数
jg *Label*	jnle	~(SF ^ OF) & ~ZF	大于（有符号>）
jge *Label*	jnl	~(SF ^ OF)	大于或等于（有符号>=）
jl *Label*	jnge	SF ^ OF	小于（有符号<）
jle *Label*	jng	(SF ^ OF) | ZF	小于或等于（有符号<=）
ja *Label*	jnbe	~CF & ~ZF	超过（无符号>）
jae *Label*	jnb	~CF	超过或相等（无符号>=）
jb *Label*	jnae	CF	低于（无符号<）
jbe *Label*	jna	CF | ZF	低于或相等（无符号<=）

图 3-15 jump 指令。当跳转条件满足时，这些指令会跳转到一条带标号的目的地。有些指令有“同义名”，也就是同一条机器指令的别名

栗子如

```
movq $0,%rax
jmp .L1
movq (%rax),%rdx

.L1:
    popq %rdx
```

这里执行jmp指令就会跳过下面的movq指令，直接到达.L1位置执行代码

选择结构

if-else

这是通常使用的格式`test-expr`是一个值为0或1的表达式

```
if(test-expr)
    then-statement
else
    else-statement
```

而汇编的实现通常使用这样的形式

```
t = test-expr
if(!t)
    goto false;
then-statement
goto done;
false:
    else-statement
done:
```

而这中间，这是条件跳转指令

```
if(!t)
    goto false;
```

这是无条件跳转指令

```
goto done;
```

循环结构

1.do-while

重复执行**body-statement**，对**test-expr**求值，若求值结果非0则继续循环，所以**body-statement**至少会被执行一次

```
do
    body-statement
while(test-expr)


loop:
    body_statement
    t=test_expr;
    if (t) goto loop;
```

2.while

在第一次执行**body-statement**之前就会对**test-expr**求值，循环有可能就此中止，所以有可能不会进入循环。

```
while(test-expr)
    body-statement


//第一种翻译方法
    goto test;
loop:
    loop_body_statement
    t=test_expr;
    if (t) goto loop;
done:

//第二种翻译方法
t=test-expr;
if(!t)
```

```
    goto done;
loop:
    body-statement
    t=test-expr;
    if(t)
        goto loop;
done
```

3.for

先对初始表达式**init-expr**求值，然后进入循环，在循环中再对测试条件**test-expr**求值，若为假则退出，否则执行**body-statement**，最后对更新表达式**update-expr**求值，所以在效果上和while循环的行为一样

```
for(init-expr; test-expr; update-expr)
    body-statement


    init_expr;
    t=test_expr;
    if (!t) goto done;
loop:
    body_statement
    update_expr;
    t=test_expr;
    if (t) goto loop;
done:
```

数组

C语言中数组的声明格式：

```
T A[N]
```

这表示内存中一块连续的区域，首地址就是A，可以用0~N-1的索引来访问数组的元素

声明	元素大小	总大小	起始地址	元素i
char A[8]	1	8	x	x+i
char *B[8]	8	64	x	x+8i
int C[8]	4	32	x	x+4i
double *D[8]	8	64	x	x+8i

也可以通过内存引用

```
E[i]
# addres of E in %rdx & i in %rdx
movl (%rdx,%rcx,4), %eax
```

指针运算

对表示某对象的Expr，&Expr给出该对象的地址的指针

对表示某地址的AExpr，*AExpr给出该地址的值

表达式	类型	值	汇编代码
E	int*	x_E	movq %rdx,%rax
E[0]	int	M[x_E]	movl (%rdx),%rax
E[i]	int	M[x_E+4i]	movl (%rdx,%rcx,4),%eax
&E[2]	int*	x_E+8	leaq 8(%rdx),%rax
E+i-1	int*	x_E+4i-4	leaq-4(%rdx,%rcx,4),%rax
*(E+i-3)	int	M[$x_E+4i-12$]	movl-12(%rdx,%rcx,4),%eax
&E[i]-E	long	i	movq %rcx,%rax

静态区数组的初始化和访问

```c
//buf.c
int buf[2] = {10, 20};//静态区分配的数组
int main()
{
    int i, sum=0;
    for(i=0; i<2; i++)
    {
        sum += buf[i];
    }
    return sum
}
```

`gcc buf.c -o buf`

`objdump -D buf`

```
00004018 <buf>:
    4018:       0a 00                   or     (%eax),%al
    401a:       00 00                   add    %al,(%eax)
    401c:       14 00                   adc    $0x0,%al
```

链接后，buf在可执行目标文件的数据段中分配到空间

此时`buf=&buf[0]`

```
00001189 <main>:
    1189:       55                      push   %ebp
    118a:       89 e5                   mov    %esp,%ebp
    118c:       83 ec 10                sub    $0x10,%esp
    118f:       e8 31 00 00 00          call   11c5 <__x86.get_pc_thunk.ax>
    1194:       05 6c 2e 00 00          add    $0x2e6c,%eax
    1199:       c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%ebp)
    11a0:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%ebp)
    11a7:       eb 11                   jmp    11ba <main+0x31>
    11a9:       8b 55 fc                mov    -0x4(%ebp),%edx
    11ac:       8b 94 90 18 00 00 00    mov    0x18(%eax,%edx,4),%edx
    11b3:       01 55 f8                add    %edx,-0x8(%ebp)
```

```
    11b6:       83 45 fc 01             addl   $0x1,-0x4(%ebp)
    11ba:       83 7d fc 01             cmpl   $0x1,-0x4(%ebp)
    11be:       7e e9                   jle    11a9 <main+0x20>
    11c0:       8b 45 f8                mov    -0x8(%ebp),%eax
    11c3:       c9                      leave
    11c4:       c3                      ret
```

假设i被分配在ECX中，sum被分配在EAX中：

```
sum += buf[i];    ->  addl buf( , %ecx, 4), %eax 或 addl 0(%edx , %ecx, 4), %eax
i++               ->  addl &1, %ecx
```

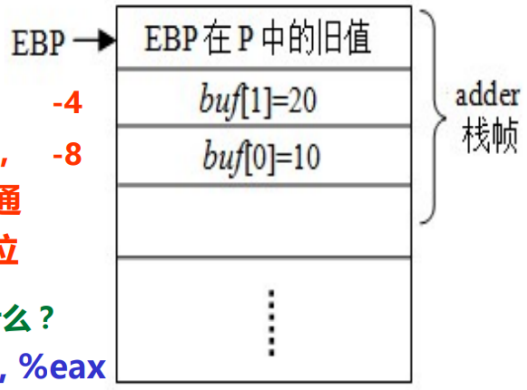
- **auto型数组的初始化和访问**

```
int adder ( )
{
   int buf[2] = {10, 20};
   int i, sum=0;
   for (i=0; i<2; i++)
         sum+=buf[i];
   return sum;
}
```

分配在栈中，故数组首址通过EBP来定位

EDX、ECX各是什么？

addl (%edx, %ecx, 4), %eax



对buf进行初始化的指令是什么？

```
movl $10, -8(%ebp)    //buf[0]的地址为R[ebp]-8，将10赋给buf[0]
movl $20, -4(%ebp)    //buf[1]的地址为R[ebp]-4，将20赋给buf[1]
```

若buf首址在EDX中，则获得buf首址的对应指令是什么？

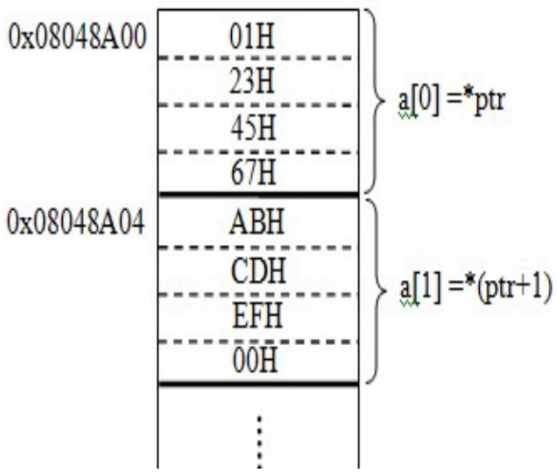
```
leal -8(%ebp), %edx  //buf[0]的地址为R[ebp]-8，将buf首址送EDX
```

- **数组与指针**

✓ 在指针变量目标数据类型与数组类型相同的前提下，指针变量可以指向数组或数组中任意元素

✓ 以下两个程序段功能完全相同，都是使ptr指向数组a的第0个元素a[0]。a的值就是其首地址，即a=&a[0]，因而a=ptr，从而有&a[i]=ptr+i=a+i以及a[i]=ptr[i]=*(ptr+i)=*(a+i)。

```
（1）int  a[10];
       int  *ptr=&a[0];
（2）int  a[10], *ptr;
       ptr=&a[0];
```



小端方式下a[0]=?,a[1]=?

a[0]=0x67452301, a[1]=0x0efcdab

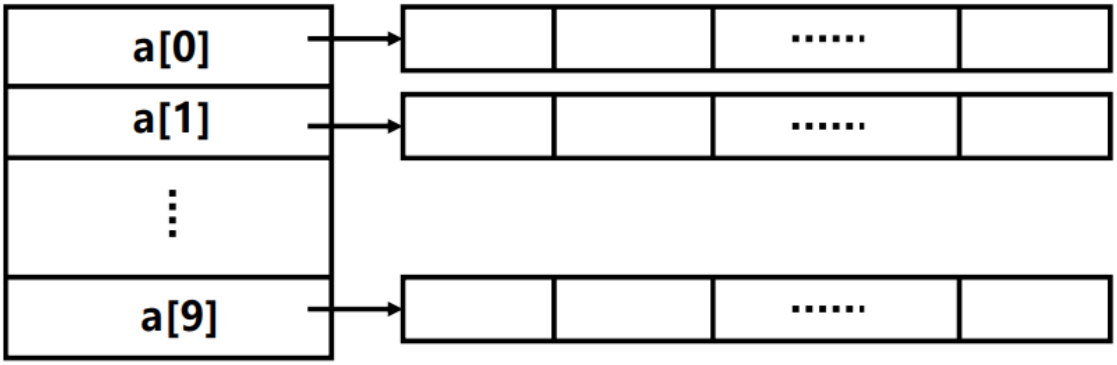
数组首址0x8048A00在ptr中，ptr+i 并不是用0x8048A00加 i 得到，而是等于0x8048A00+4*i

指针数组

由若干指向同类目标的指针变量组成的数组

```
存储类型 数据类型 *指针数组名[元素个数];

//定义了一个指针数组a，有10个元素，每个元素都是一个指向int型的指针
int *a[10];
```

- **一个指针数组可以实现一个二维数组。**



定长数组

声明一个16X16的整型数组

```
#define N 16
typedef int fix_matrix[N][N];
```

嵌套数组

声明

```
int A[5][3]

A = |a00 a01 a02|
    |a10 a11 a12|
    |a20 a21 a22|
    |a30 a31 a32|
    |a40 a41 a42|
```

在内存是以行优先排列

```
A[0][0]   A[0][1]   A[0][2]   A[1][0]   ...
a00       a01       a02       a10       a11 a12

A[0]: a00 a01 a02
```

变长数组

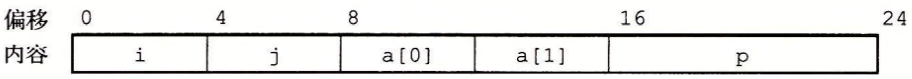
```
int A[expr1][expr2]

int var_ele(long n, int A[n][n], long i, long j)
{
    return A[i][j];
}
```

参数n必须在`A[n][n]`之前，这样函数才可以在遇到该数组时计算出维度

结构体与联合

结构体

```
struct rec{
    int i;
    int j;
    int a[2];
    int *p;
}
```



通过其`基址+偏移`来访问结构体内容

其占据的内存空间是最大的类型所需的字节倍数，即

初始化

```
struct cont_info
{
   char  id[8];
   char name[12];
   unsigned post;
   char address[100];
   char phone[20];
} x = {“0000000”, “ZhangS”, 210022, “273 long street, High Building #3015”,
“12345678”};

struct cont_info x={“0000000”, “ZhangS”, 210022, “273 long street, High Building
#3015”, “12345678”};
```

若x分配在地址0x8049200开始的区域，则

```
x=&(x.id)=0x8049200
&x(x.name)=0x8049200+8
//0x8049208~0x804920D: ZhangS
//0x804920E:\0
//0x804920F~0x8049213:[空字符]
&x(x.post)=0x8049200+8+12
    ...
```

访问

```
//初始化内容见上方
s1.post = 123333;
strcpy(s1.id, "123");
```

结构体作为参数时

按地址传参调用

将结构体成员所在的地址传到参数区

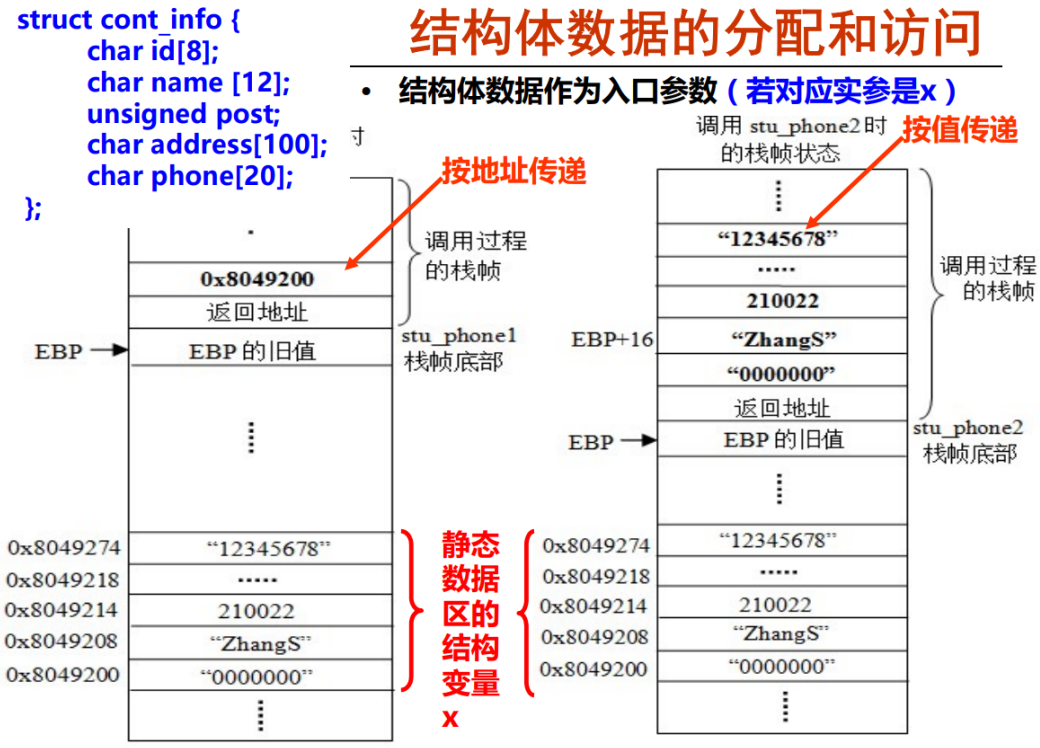
```
void stu_phone1 ( struct cont_info *s_info_ptr)
{
    printf ("%s phone number: %s", (*s_info_ptr).name, (*s_info_ptr).phone);
}

//(*stu_info).name
//stu_info->name
```

按值传参调用

```
void stu_phone2 ( struct cont_info s_info)
{
    printf ("%s phone number: %s", s_info.name, s_info.phone);
}

//stu_info.name
```

结构体所有成员值作为实参存储到参数区，即将结构体成员都复制到栈中，更新后的成员数据也无法在调用过程中使用。

由于按值传递有一个整体的复制过程，增加了程序运行的开销



联合

联合体各成员共享存储空间，按最大长度成员的长度算，即用不同的字段来引用相同的内存块

> 联合提供了一种方式，能够规避C语言的类型系统，允许以多种系统来引用一个对象

```
struct S{
    cahr c;
    int i[2];
    double v;
};

union U{
    char c;
    int i[2];
    double v;
}
```

union和struct的区别

类型	c的偏移量	i的偏移量	v的偏移量	大小
S	0	4	16	24
U	0	0	0	8

字节对齐

Windows：如果数据类型需要K个字节，那么地址都必须是K的倍数，即int型地址是4的倍数，short-2、double、long long-8、float-4、char不对齐

Linux：2字节数据类型的地址必须是2的倍数，int、double、float的地址(即偏移)必须是4的倍数

对各种不同长度的存放的补齐策略：

> 1字节：char
>
> 2字节：short 地址最低1比特为`0`
>
> 4字节：int, float... 地址最低2比特为`00`
>
> 8字节：double, long, char*,... 地址最低3比特为`000`
>
> 16字节：long double 地址最低4比特为`0000`

对结构体来说，所占空间是最大的类型所需字节的倍数

```
struct S1{
    char c;
    int i;
    char d;
}

struct S2{
    int i;
    char c;
    char d;
}
```

灰色部分表示补齐的字节，由此通过调整变量声明的顺序，有可能节省一定的空间

对齐方式设定

```
#pragma pack(n)
```

- 为编译器指定**结构体**或**类**内部的**成员变量**对齐方式
- 当`自然边界比n大`，则按n字节对齐（自然边界：int-4字节、short-2字节、float-4字节）
- 当`缺省`或`#pragma pack()`，按自然边界对齐

```
__attribute__((aligned(m)))
```

- 为编译器指定一个结构体或类或联合体或一个单独的变量(对象)的对齐方式
- 按m字节对齐(m必须是2的幂次方)，且占用空间大小也是m的整数倍，以保证在申请连续存储空间时各元素也按m字节对齐

```
__attribute__((packed))
```

- 不按边界对齐，称为紧凑方式

```
#include<stdio.h>
#pragma pack(4)
typedef struct {
    uint32_t    f1;
    uint8_t     f2;
    uint8_t     f3;
    uint32_t    f4;
    uint64_t    f5;
}__attribute__((aligned(1024))) ts;
int main()
{
    printf("Struct size is: %d, aligned on 1024\n",sizeof(ts));
    printf("Allocate f1 on address: 0x%x\n",&(((ts*)0)->f1));
    printf("Allocate f2 on address: 0x%x\n",&(((ts*)0)->f2));
    printf("Allocate f3 on address: 0x%x\n",&(((ts*)0)->f3));
    printf("Allocate f4 on address: 0x%x\n",&(((ts*)0)->f4));
    printf("Allocate f5 on address: 0x%x\n",&(((ts*)0)->f5));
    return 0;
}
```

输出：

```
Struct size is: 1024, aligned on 1024
Allocate f1 on address: 0x0
Allocate f2 on address: 0x4
Allocate f3 on address: 0x5
Allocate f4 on address: 0x8
Allocate f5 on address: 0xc
```