

程序的机器级表示：数据

施朱鸣

10 月 22 日



Outline

数组

结构体

浮点数

练习



数组



内存分配

```
char string[12];  
int val[5];  
T A[L];  
T* p = new T[L];
```

在内存中分配 $L \times \text{sizeof}(T)$ bytes 的连续空间。

数据类型是给编译器看的，分配的空间并没有记录自己的数据类型。



数组元素访问

可以用 [] 和指针运算来访问，C 语言指针计算会自动调整跨度
汇编会使用之前介绍过的寻址公式 $Imm(r_b, r_i, s)^1$

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

¹可以复习书 121 页

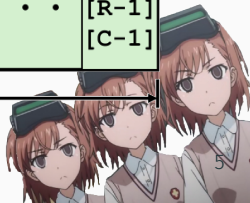
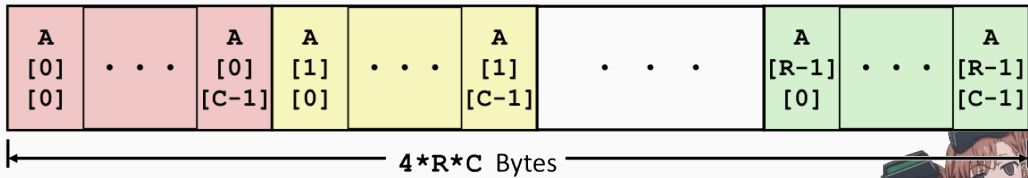


多维数组

$$\begin{pmatrix} A[0][0] & \cdots & A[0][C-1] \\ \vdots & & \vdots \\ A[R-1][0] & \cdots & A[R-1][C-1] \end{pmatrix}$$

在内存中被拉直，靠前的维度优先安排（先行后列）连续排列。

```
int A[R][C];
```



多维数组的访问

对于 $A[i][j]$, 公式为

$$A + i \times (C \times K) + j \times K$$

编译器利用寻址公式计算的具体方法根据优化情况而定, 下面是一个例子



多维数组的访问效率

```
// multiarray.c
int a[2][2] = {1,1,1,1};
void ij() {
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
            a[i][j]++;
}
void ji() {
    for (int j = 0; j < 2; j++)
        for (int i = 0; i < 2; i++)
            a[i][j]++;
}
```



多维数组的访问效率

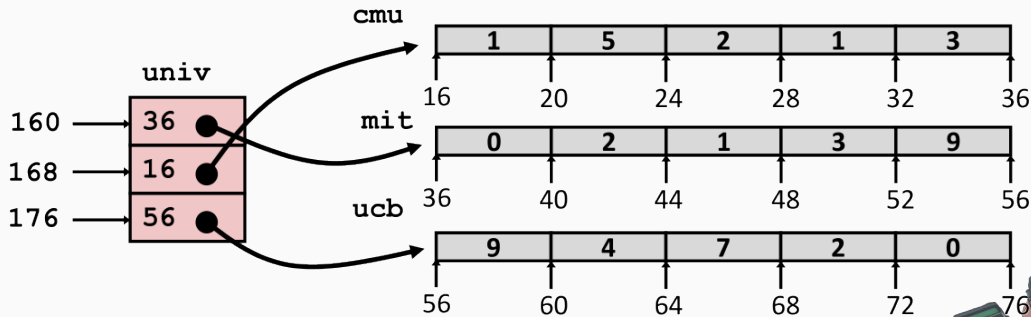
gcc -Og -S multiarray.c 发现指令相同，猜测与优化程度、内存管理相关

5- ij:	5+ ji:
6 .LFB0:	6 .LFB0:
7 .cfi_startproc	7 .cfi_startproc
8 movl \$0, %edi	8 movl \$0, %edi
9 jmp .L2	9 jmp .L2
10 .L3:	10 .L3:
11 leaq a(%rip), %rcx	11 leaq a(%rip), %rcx
12- movslq %eax, %rdx	12+ movslq %edi, %rdx
13- movslq %edi, %rsi	13+ movslq %eax, %rsi
14 leaq (%rdx,%rsi,2), %rsi	14 leaq (%rdx,%rsi,2), %rsi
15 movl (%rcx,%rsi,4), %edx	15 movl (%rcx,%rsi,4), %edx
16 addl \$1, %edx	16 addl \$1, %edx
17 movl %edx, (%rcx,%rsi,4)	17 movl %edx, (%rcx,%rsi,4)
18 addl \$1, %eax	18 addl \$1, %eax
19 .L4:	19 .L4:
20 cmpl \$1, %eax	20 cmpl \$1, %eax
21 jle .L3	21 jle .L3
22 addl \$1, %edi	22 addl \$1, %edi
23 .L2:	23 .L2:
24 cmpl \$1, %edi	24 cmpl \$1, %edi
25 jg .L6	25 jg .L6
26 movl \$0, %eax	26 movl \$0, %eax
27 jmp .L4	27 jmp .L4
28 .L6:	28 .L6:
29 rep ret	29 rep ret
30 .cfi_endproc	30 .cfi_endproc



多层数组

其实就是指针数组，访问第 n 层的元素需要访问 n 次内存，时间效率低



从定长数组到变长数组

固定维度数组：编译器知道数组每个维度的大小 C

$$A + i \times (C \times K) + j \times K$$

编译的时候就把 $C \times K$ 算好了

```
/* Get element a[i][j] */  
int fix_ele(fix_matrix a, size_t i, size_t j) {  
    return a[i][j];  
}
```

```
# a in %rdi, i in %rsi, j in %rdx  
salq    $6, %rsi          # 64*i  
addq    %rsi, %rdi         # a + 64*i  
movl    (%rdi,%rdx,4), %eax # M[a + 64*i + 4*j]  
ret
```



从定长数组到变长数组

固定维度数组：每个维度的大小 C 在运行时根据传入的参数确定

$$A + i \times (C \times K) + j \times K$$

$C \times K$ 只能在运行时计算，时间开销更大。当然，数组本身的连续内存空间大小还是确定的。元素的大小 K 也是确定的。只是拓展了这个函数的适用性。

```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n], size_t i, size_t j)
{
    return a[i][j];
}
```

```
# n in %rdi, a in %rsi, i in %rdx, j in %rcx
imulq    %rdx, %rdi          # n*i
leaq     (%rsi,%rdi,4), %rax  # a + 4*n*i
movl     (%rax,%rcx,4), %eax  # a + 4*n*i + 4*j
ret
```



正确理解“变长”

```
// vararray.c
# include<stdio.h>
int main(){
    int n, p[n];
    scanf("%d",&n);
    int q[n];
    printf("p:%lu and q:%lu",sizeof(p),sizeof(q));
    return 0;
}
```

> gcc vararray.c && echo 3 | ./a.out'

> p:131068 and q:12



结构体

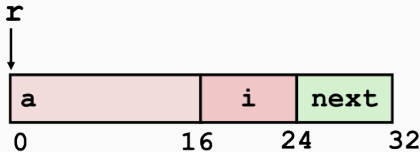


结构体的定义

按照声明的顺序，各元素顺序排列在一片连续的内存空间中

空间的大小由编译决定，机器级访问的方法类似于数组元素的访问，通过指针计算

```
struct rec {  
    int a[4];  
    size_t i;  
    struct rec *next;  
};
```

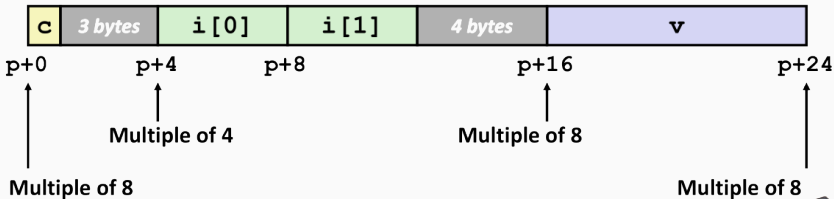


结构体的对齐

对齐的规则：

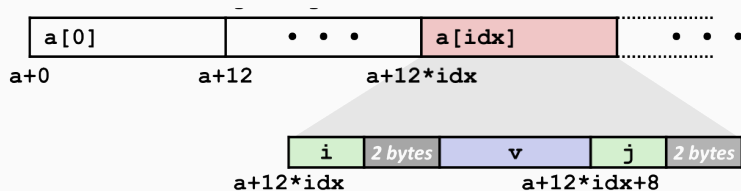
- 结构体内部对齐：每个 K 字节的基本对象的地址必须是 K 的倍数
- 结构体整体对其：结构体整体长度是结构体中最大的元素长度的整数倍

编译器会用 0 补足空隙，保证对齐



结构体元素的访问

以不变应万变：指针运算，遇到对齐补的 0 记得跳过



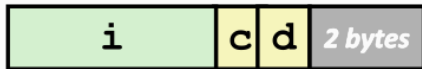
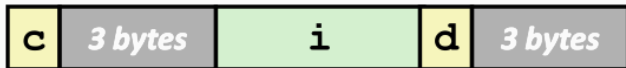
```
short get_j(int idx)
{
    return a[idx].j;
}
```

```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(,%rax,4),%eax
```



结构体存储的优化

根据对其规则调整结构体元素的顺序

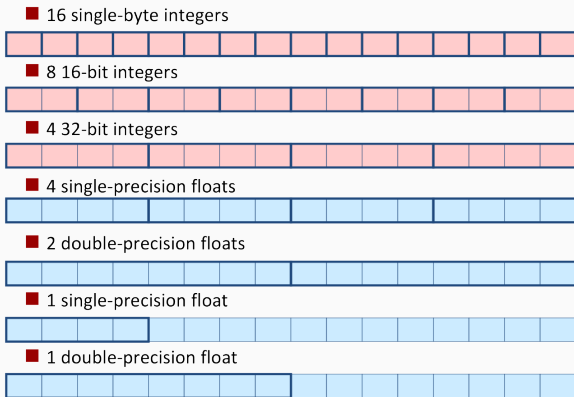


浮点数



浮点寄存器 XMM

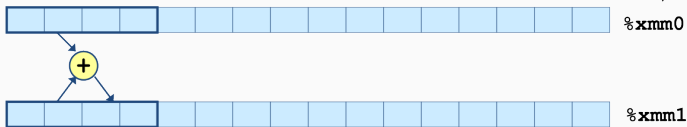
- 一共 16 个，每个 16 bytes，命名为 %xmm0 到 %xmm15
- 不止可以存浮点数，也不止可以放一个数



浮点计算指令

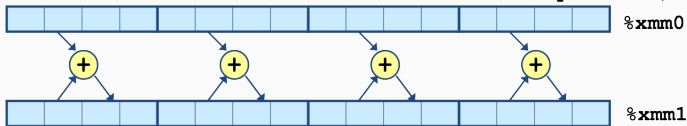
■ Scalar Operations: Single Precision

`addss %xmm0, %xmm1`



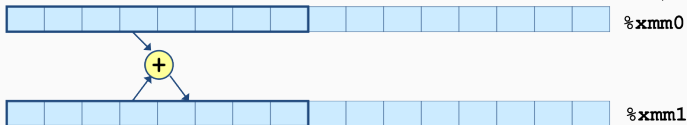
■ SIMD Operations: Single Precision

`addps %xmm0, %xmm1`



■ Scalar Operations: Double Precision

`addsd %xmm0, %xmm1`



全部浮点数参数时的传参

规则可以类比非浮点的计算

- 参数依次放入%xmm0, %xmm1 等
- 返回值放入%xmm0 到
- 所有的 xmm 寄存器都是 caller-saved

如果有非浮点参数和返回值呢？



全部浮点数参数时的传参

规则可以类比非浮点的计算

- 参数依次放入%xmm0, %xmm1 等
- 返回值放入%xmm0 到
- **所有的 xmm 寄存器都是 caller-saved**

如果有非浮点参数和返回值呢？

非浮点的参数用普通寄存器依次存放，浮点的参数用%xmm 依次存放。

xmm 寄存器之间的移动和内存与 xmm 的移动有浮点运算的一套指令



浮点相关的移动指令²

把指令前面的 v 去掉。其他浮点运算指令见书上 210 页。

指令	源	目的	描述
vmovss	M_{32}	X	传送单精度数
vmovss	X	M_{32}	传送单精度数
vmovsd	M_{64}	X	传送双精度数
vmovsd	X	M_{64}	传送双精度数
vmovaps	X	X	传送对齐的封装好的单精度数
vmovapd	X	X	传送对齐的封装好的双精度数

²教材 206 页



练习



练习题 3.38

练习题 3.38 考虑下面的源代码，其中 M 和 N 是用 `# define` 声明的常数：

```
long P[M][N];
long Q[N][M];

long sum_element(long i, long j) {
    return P[i][j] + Q[j][i];
}
```

在编译这个程序中，GCC 产生如下汇编代码：

```
    long sum_element(long i, long j)
    i in %rdi, j in %rsi
1   sum_element:
2       leaq    0(,%rdi,8), %rdx
3       subq    %rdi, %rdx
4       addq    %rsi, %rdx
5       leaq    (%rsi,%rsi,4), %rax
6       addq    %rax, %rdi
7       movq    Q(,%rdi,8), %rax
8       addq    P(,%rdx,8), %rax
9       ret
```

运用逆向工程技能，根据这段汇编代码，确定 M 和 N 的值。



练习题 3.44



练习题 3.44 对下面每个结构声明，确定每个字段的偏移量、结构总的大小，以及在 x86-64 下它的对齐要求：

- A. `struct P1 { int i; char c; int j; char d; };`
- B. `struct P2 { int i; char c; char d; long j; };`
- C. `struct P3 { short w[3]; char c[3] };`
- D. `struct P4 { short w[5]; char *c[3] };`
- E. `struct P5 { struct P3 a[2]; struct P2 t };`



致谢



致谢

LaTeX 代码开源在 <https://github.com/ShiZhuming/pku-ics>

祝大家写 lab 顺利，谢谢聆听！

