Machine Prog (Assembly Code) ——Procedures & Data

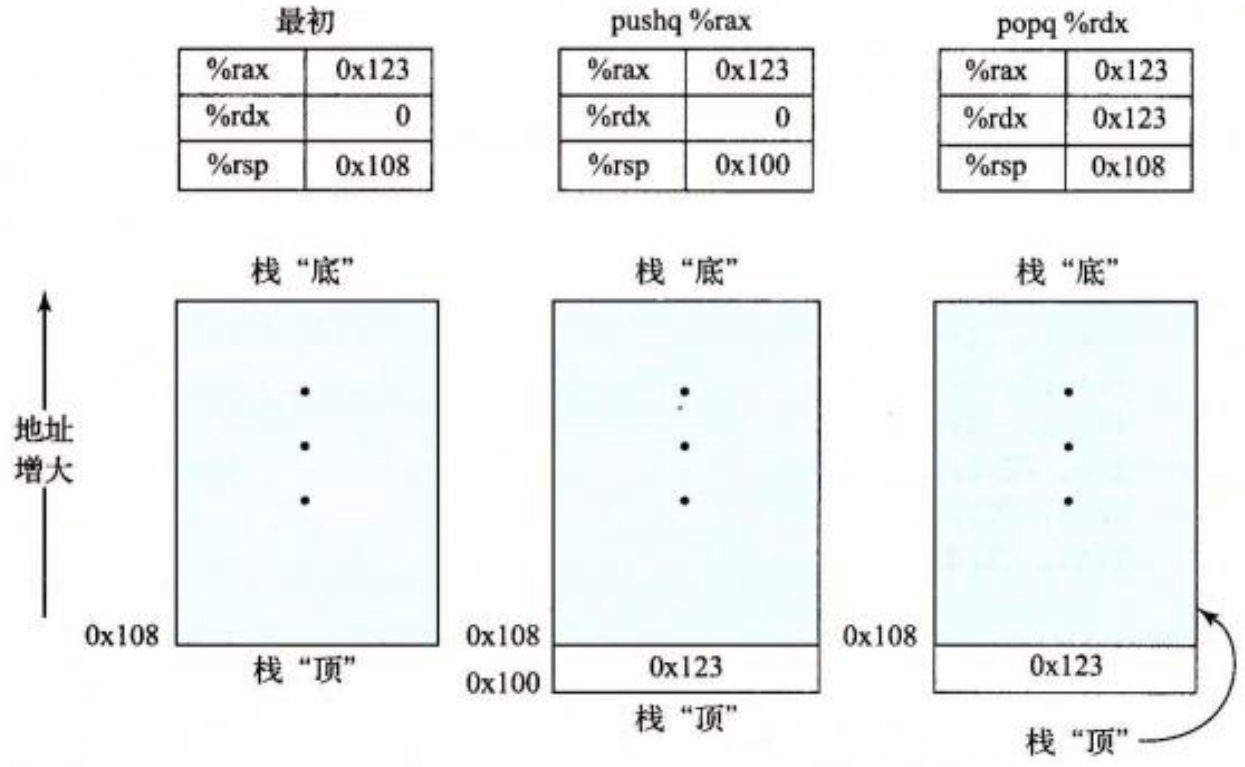
余文凯 康子熙 赵廷昊 许珈铭

2023.10.11

Procedures (CS:APP Ch. 3.7)

赵廷昊

push pop



- 栈——全局存储
- push数据压入栈 **sub+mov**
- pop弹出数据 mov+add
- 栈指针%rsp保存栈顶元素地址
- 栈顶元素地址是所有栈中元素地址中最低的
- 栈要以8bit为单位对齐

pushq用在callee保存的寄存器进行压栈保存

movq用在函数分配好的栈帧，局部变量的存放（subq $ %rsp）

过程调用

- 传递控制
- 传递数据
- 分配和释放内存

调用函数

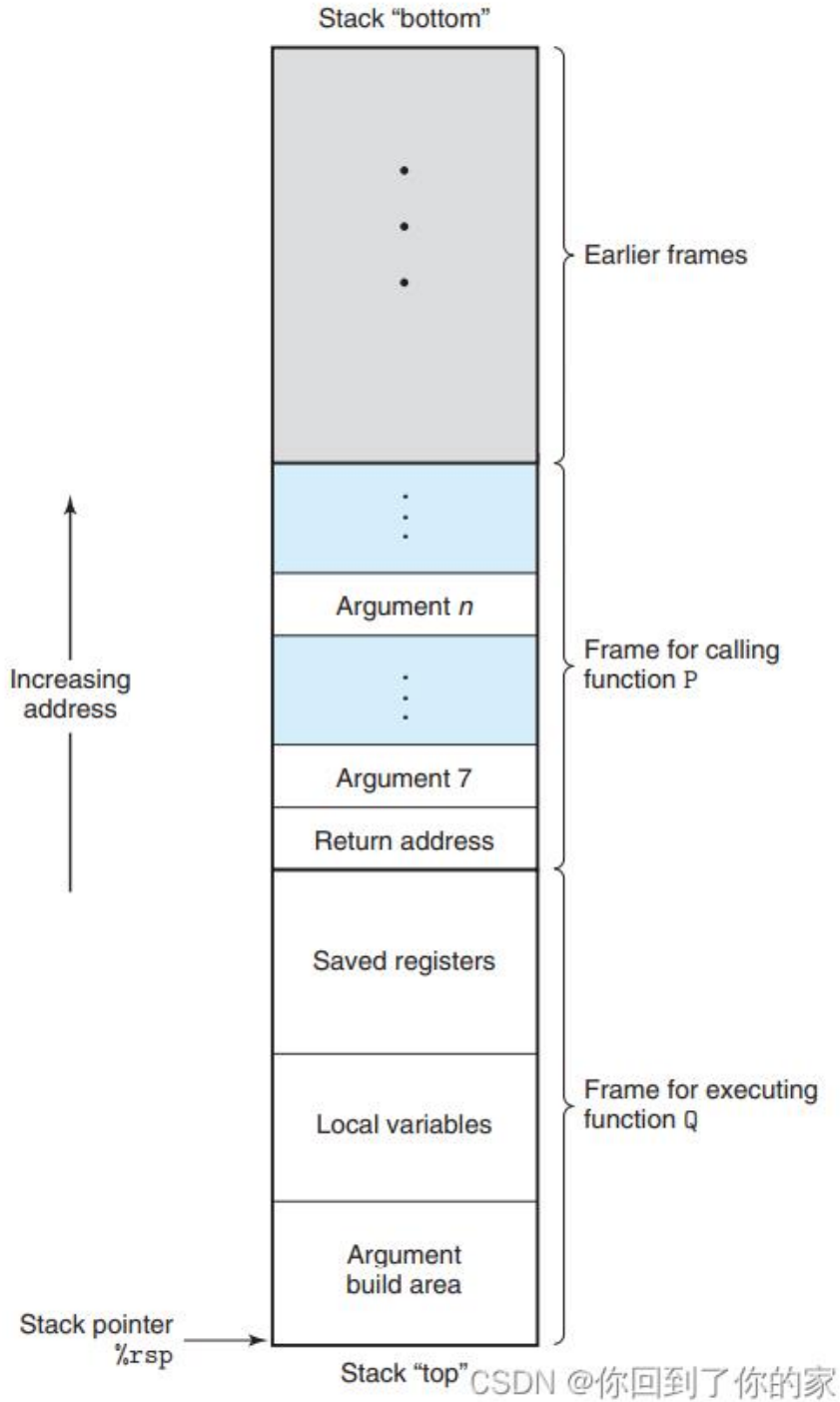
程序计数器地址转移

寄存器与栈

栈帧

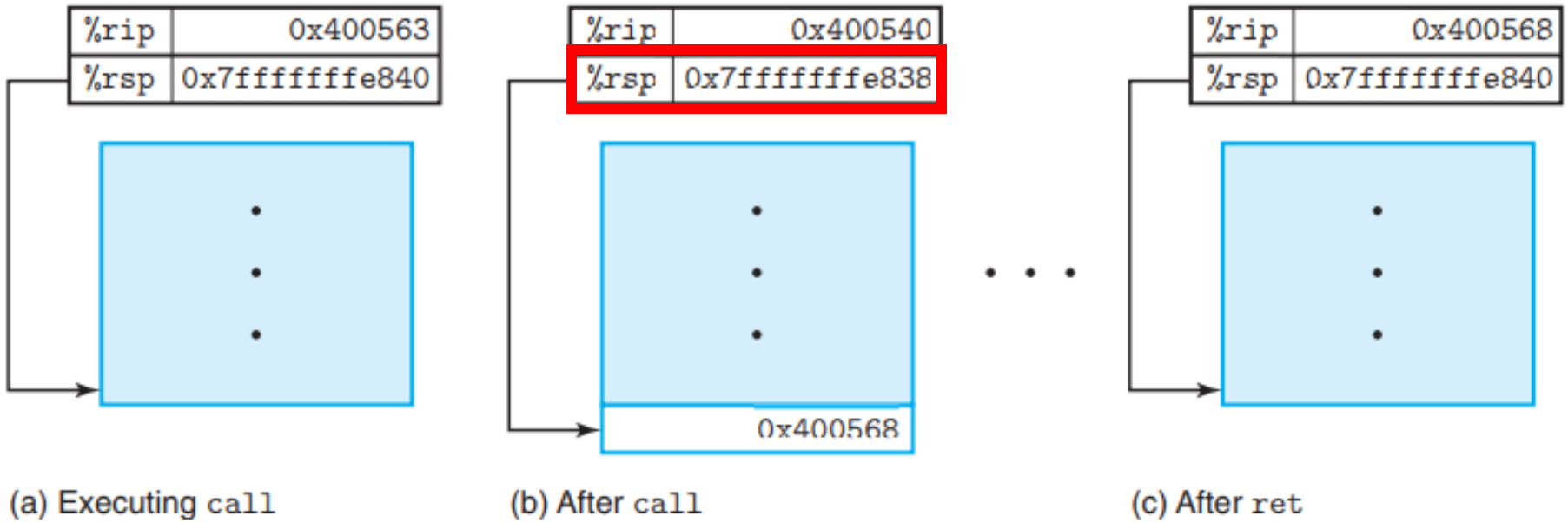
栈帧结构

- 一个栈帧 (stack frame)
  - 单个过程调用所分配的栈的部分
- 大多数过程的栈帧是定长的
- 调用函数P
  - 参数
  - 返回地址——P与Q的分界
- 正在执行的函数Q
  - 被保存的寄存器
  - 局部变量
  - 参数构造



转移控制

- call Q　　把地址A压入栈中 将PC设置为Q的起始地址
- retq　　从栈中弹出A并把PC设为A

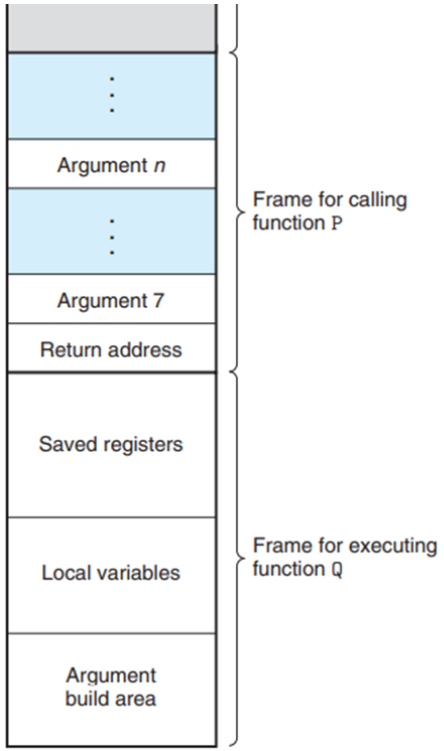


数据传送

函数参数传递

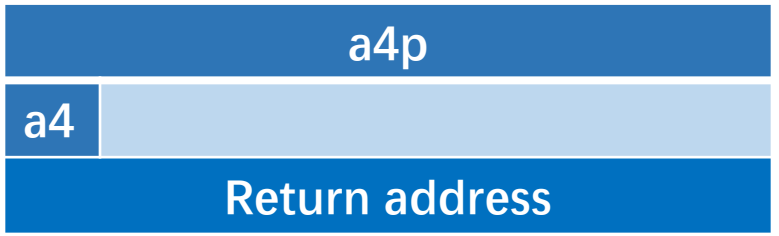
- 在x86-64下，最多可以通过寄存器传递六个integral arguments
- 当一个函数有超过六个integral arguments时，其他参数通过栈从下往上进行存储
- 返回值保存在%rax中

Operand size (bits)	Argument number					
	1	2	3	4	5	6
64	%rdi	%rsi	%rdx	%rcx	%r8	%r9
32	%edi	%esi	%edx	%ecx	%r8d	%r9d
16	%di	%si	%dx	%cx	%r8w	%r9w
8	%dil	%sil	%dl	%cl	%r8b	%r9b



例

栈的对齐

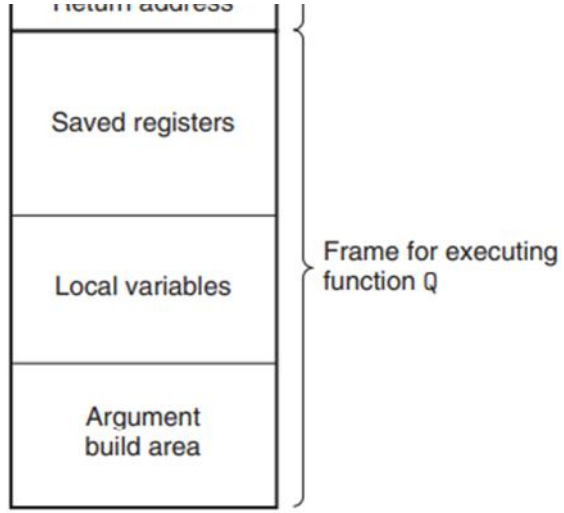



```
void proc(long  a1, long  *a1p,
          int   a2, int   *a2p,
          short a3, short *a3p,
          char  a4, char  *a4p)
{
    *a1p += a1;
    *a2p += a2;
    *a3p += a3;
    *a4p += a4;
}
```

```
void proc(a1, a1p, a2, a2p, a3, a3p, a4, a4p)
Arguments passed as follows:
  a1  in %rdi          (64 bits)
  a1p in %rsi          (64 bits)
  a2  in %edx          (32 bits)
  a2p in %rcx          (64 bits)
  a3  in %r8w          (16 bits)
  a3p in %r9           (64 bits)
  a4  at %rsp+8        ( 8 bits)
  a4p at %rsp+16       (64 bits)
1 proc:
2   movq    16(%rsp), %rax    Fetch a4p   (64 bits)
3   addq    %rdi, (%rsi)      *a1p += a1  (64 bits)
4   addl    %edx, (%rcx)      *a2p += a2  (32 bits)
5   addw    %r8w, (%r9)       *a3p += a3  (16 bits)
6   movl    8(%rsp), %edx     Fetch a4    ( 8 bits)
7   addb    %dl, (%rax)       *a4p += a4  ( 8 bits)
8   ret                       Return
```

栈上局部存储

- 过程调用 Q 也使用栈存储任何不能存储于寄存器中的局部变量
  - 没有足够的寄存器保存所有的局部变量
  - 一些局部变量是数组或结构体，必须通过数组和结构体引用才能访问
  - 取址操作符 ‘&’ 被应用于局部变量，因此我们必须能为它生成一个地址
  - 如果是new申请的空间则存放在堆中
  - 堆和栈的区别（转过无数次的文章） - 知乎 (zhihu.com)



例

使用了地址运算符

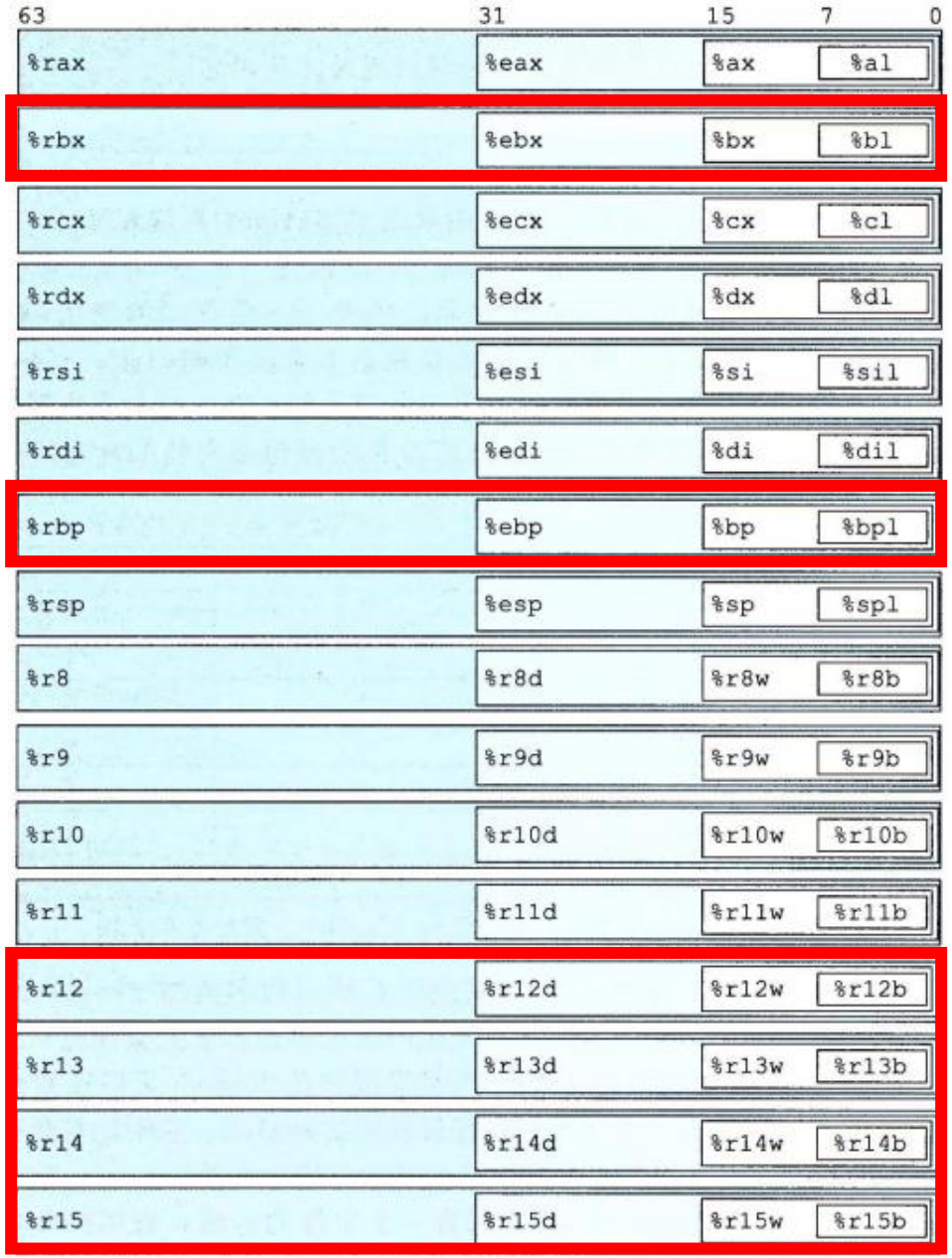
```
long swap_add(long *xp, long *yp)
{
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}

long caller()
{
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
}
```

```
    long caller()
1   caller:
2     subq    $16, %rsp          Allocate 16 bytes for stack frame
3     movq    $534, (%rsp)       Store 534 in arg1
4     movq    $1057, 8(%rsp)     Store 1057 in arg2
5     leaq    8(%rsp), %rsi      Compute &arg2 as second argument
6     movq    %rsp, %rdi         Compute &arg1 as first argument
7     call    swap_add           Call swap_add(&arg1, &arg2)
8     movq    (%rsp), %rdx       Get arg1
9     subq    8(%rsp), %rdx      Compute diff = arg1 - arg2
10    imulq   %rdx, %rax         Compute sum * diff
11    addq    $16, %rsp          Deallocate stack frame
12    ret                        Return
```

寄存器局部存储

- 寄存器%rbx、%rbp和%r12~r15为被调用者保存寄存器
  - 确保当Q返回P时这些寄存器的值和P调用Q时是一样的

- 除了栈指针%rsp外的其他寄存器为调用者保存寄存器



例

(a) Calling function

```
long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

后续需要x的值

后续需要u的值

(b) Generated assembly code for the calling function

```
    long P(long x, long y)
    x in %rdi, y in %rsi
1   P:
2     pushq   %rbp          Save %rbp
3     pushq   %rbx          Save %rbx
4     subq    $8, %rsp      Align stack frame
5     movq    %rdi, %rbp    Save x
6     movq    %rsi, %rdi    Move y to first argument
7     call    Q             Call Q(y)
8     movq    %rax, %rbx    Save result
9     movq    %rbp, %rdi    Move x to first argument
10    call    Q             Call Q(x)
11    addq    %rbx, %rax    Add saved Q(y) to Q(x)
12    addq    $8, %rsp      Deallocate last part of stack
13    popq    %rbx          Restore %rbx
14    popq    %rbp          Restore %rbp
15    ret
```

作为callee
使用callee保存寄存器前保存内容

作为caller
调用Q前保存局部变量
到callee保存寄存器中

作为callee
恢复callee保存寄存器初始状态

pushq与movq的区别

递归过程

- 寄存器与栈使得递归过程调用中每个过程调用在栈中都有自己的私有空间——多个未完成调用的局部变量不会相互影响
- 与调用其他函数没有太大的差别

例

(a) C code

```
long rfact(long n)
{
    long result;
    if (n <= 1)
        result = 1;
    else
        result = n * rfact(n-1);
    return result;
}
```

(b) Generated assembly code

```
    long rfact(long n)
    n in %rdi
1   rfact:
2     pushq   %rbx
3     movq    %rdi, %rbx
4     movl    $1, %eax
5     cmpq    $1, %rdi
6     jle     .L35
7     leaq    -1(%rdi), %rdi
8     call    rfact
9     imulq   %rbx, %rax
10  .L35:
11    popq    %rbx
12    ret
```

```
Save %rbx
Use %rbx as callee-saved register
Set return value = 1
Compare n:1
If <=, goto done
Compute n-1
Call rfact(n-1)
Multiply result by n
done:
Restore %rbx
Return
```

保证了各个进程中n的值的独立性

递归调用

Data (CS:APP Ch. 3.8-Ch. 3.10.1)

康子熙

数组存储的基本原则

对于声明：T A[N]

数组元素i的地址：$X_A + L \cdot i$

在64位机器上，所有指针数据类型都由8个字节存储；32位机器为4字节

```
char    A[12];
char   *B[8];
int     C[6];
double *D[5];
```

These declarations will generate arrays with the following parameters:

Array	Element size	Total size	Start address	Element i
A	1	12	x_A	$x_A + i$
B	8	64	x_B	$x_B + 8i$
C	4	24	x_C	$x_C + 4i$
D	8	40	x_D	$x_D + 8i$

指针运算原则

为什么第7行运行结果是+2？

- 强制类型转换的优先级高于加法
- 指针进行加减法时指针指向的值的加减只与sizeof(T)有关

在gcc编译器中，void* 的算术运算会按照 char* 来处理，即每次 p++ 会使指针移动一个字节。但这并不是标准的行为，依赖于此行为是不可移植的

```
1   #include<iostream>
2   using namespace std;
3   int main() {
4       unsigned int A=0x11112222;
5       unsigned int B=0x33336666;
6       void *x = (void *)&A;
7       void *y = 2 + (void *)&B;
8       unsigned short P = *(unsigned short *)x;
9       unsigned short Q = *(unsigned short *)y;
10      printf("0x%08x\n", *(int *)x);
11      printf("0x%08x\n", *(int *)y);
12      printf("0x%04x\n", P);
13      printf("0x%04x\n", Q);
14      printf("0x%04x", Q + P);
15      return 0;
16  }
```

通用指针 void*

通用指针具有以下特点：

- 类型中立：可以将任何类型的指针赋值给void*，且不经过显式类型转换

```
int x = 10;
double y = 20.5;
void* p1 = &x;  // Valid
void* p2 = &y;  // Valid
```

-无法直接解引用：由于 void* 是类型不明确的，所以你不能直接解引用它。

```
int* px = (int*)p1;
int value = *px;
```

需要经过类型转换

使用场景：
- 动态内存分配函数，如 malloc() 和 free()，使用 void*。
- 实现泛型数据结构和函数时，可以使用 void* 来处理各种数据类型。

指针运算原则

```
void *p
int *new_p1 = (int *)p + 7;
   //new_p1 是一个 int 指针，值 = p + 28 --先类型转换后加减
int *new_p2 = (int *)(p + 7);
   //new_p2 是一个 int 指针，值 = p + 7
```

- 指针不能相加
- T * 类型指针相减 = 地址之差 / sizeof (T)
  - 不同类型指针不能相减
  - void* - void* 会报错：表达式必须是指向完整对象类型的指针

函数指针

格式：`int (*f)(int *)`

- 函数指针指向函数而非数据

`int (*f)(int *)` vs `int *f(int *)`

函数指针　　函数

指针的阅读——右-左法则 "right-left rule"

- 定位变量名

do {
    向右阅读，直到右括号，确定其类型（是否是数组）
    再向左阅读，直到左括号，确定其内容类型
} while (括号没拆干净)

// 右侧并列的括号可能是函数

例子：int (*(*vtable)[])()

C++强制类型转换——避免裸指针的互转！

- **static_cast()**
  - 非多态的类型之间的转换
  - 无法改变const性
  - 不执行运行时类型检查（确保其安全！）
- **dynamic_cast()**
  - 用于多态类型之间的安全向下转型
  - 若失败，返回nullptr（对于指针）或抛出异常（对于引用）
- **const_cast()**
  - 添加或删除对象的const性
  - 不能通过删改const性试图修改const变量
- **reinterpret_cast()**

C++ 引用

- 本质是指针的高层封装，提高了安全性与易用性
  - 非空保证
  - 不允许进行指针运算
  - 不可重新绑定

```
void f(int &a, int &b) {
  int c = a;
  a = b;
  b = c;
}
```

```
f(int&, int&):
        movl    (%rdi), %eax
        movl    (%rsi), %edx
        movl    %edx, (%rdi)
        movl    %eax, (%rsi)
        ret
```

高维数组

对于声明：T D[R][C]

数组元素D[i][j]的地址：$X_D + L(C \cdot i + j)$

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

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

```
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
```

对于定长数组，优化器可以使用leaq来达到乘法运算的效果

高维数组

定义	sizeof(a)	sizeof(*a)	sizeof(**a)	sizeof(***a)	sizeof(****a)
int *a[5]	40	8	4	/	/
int (*a[5])	40	8	4	/	/
int (*a)[5]	8	20	4	/	/
int **a[5]	40	8	8	4	/
int **a[3][5]	120	40	8	8	4
int (*a[3])[5]	24	8	20	4	/
int (**a[3])[5]	24	8	8	20	4
int (*(*a[3])[5])	24	8	40	8	4
int (*(*a)[3])[5]	8	24	8	20	4

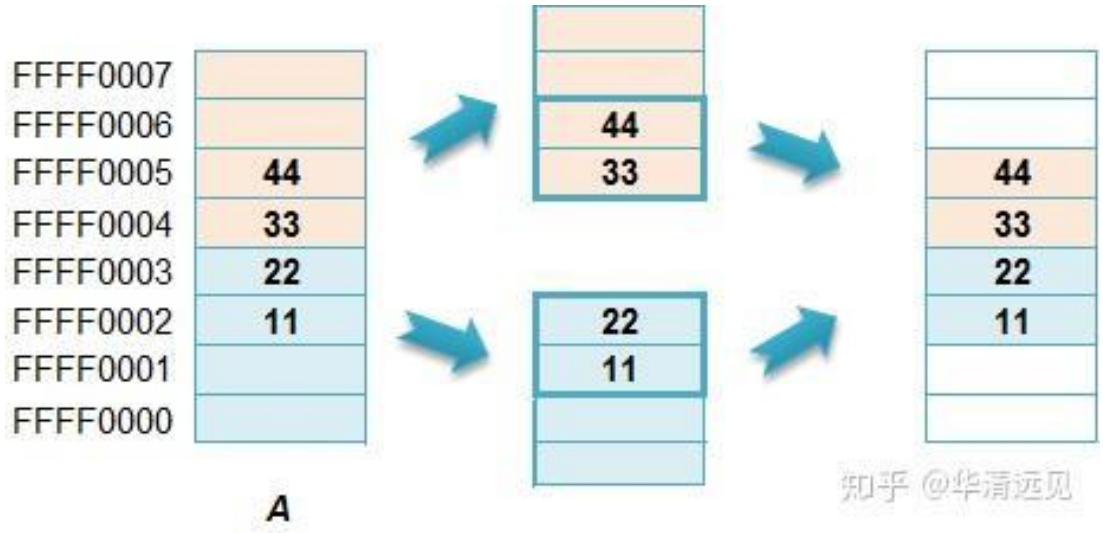
对齐

对齐要求某种类型对象的地址必须是某个值K(2/4/8)的倍数，K由sizeof决定

- **internal padding (depends on elements' size)**
- **external padding (determinates structure's size)**

提高数据访问速度：对齐的数据通常可以更快地从内存中加载和存储，因为它们遵循硬件的自然边界。例如，如果一个32位整数从4字节边界开始，那么它可以在单次内存操作中读取，而不需要多次操作。

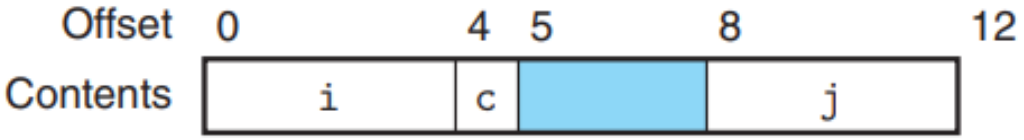
增强向量化和SIMD效率：对于使用SIMD（单指令多数据）指令的代码，对齐的数据通常提供更好的性能。这是因为SIMD指令经常需要对齐的数据，或者在处理对齐的数据时工作得更快。

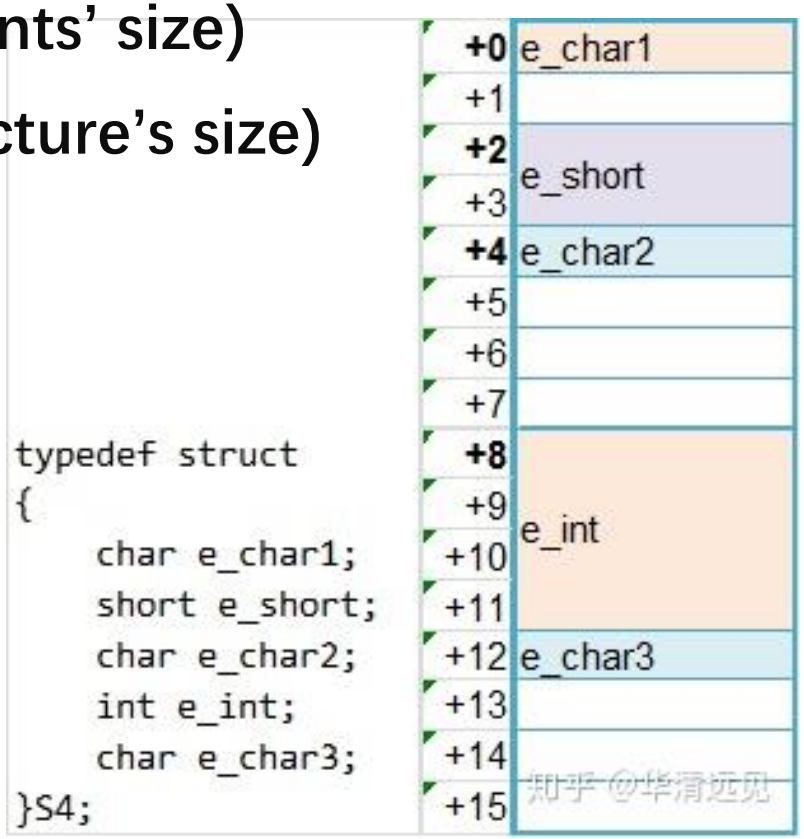


结构体 Struct

一个结构体中不同子变量在内存中几乎相邻（对齐可能会影响）

- **internal padding (depends on elements' size)**
- **external padding (determinates structure's size)**





- 数组
  - C++引用
  - 指针运算
  - 类型转换
  - 高维数组
  - 指针练习
- 数据结构
  - 对齐
  - **结构体**
  - 联合

结构体 Struct

Q9

6、有如下定义的结构，在 x86-64 下，下述结论中错误的是？

```
struct {
    char c;
    union {
        char vc;
        double value;
        int vi;
    } u;
    int i;
} sa;
```

A. sizeof(sa) == 24

B. (&sa.i - &sa.u.vi) == 8

C. (&sa.u.vc - &sa.c) == 8

D. 优化成员变量的顺序，可以做到“sizeof(sa) == 16”

答：（ B ）

联合 Union

- union 的大小是所有子变量大小的最大值
- 所有子变量在内存中起始位置相同
  - 通常情况下各个子变量是互斥的（不会同时被用到）
  - 使用枚举来表示联合中存储的数据是哪种类型，这样就可以在运行时知道如何正确地访问它
- 可用于同位级表示的互转
  - **注意大小端！**

```
double uu2double(unsigned word0, unsigned word1)
{
    union {
        double d;
        unsigned u[2];
    } temp;

    temp.u[0] = word0;
    temp.u[1] = word1;
    return temp.d;
}
```

联合 Union

Q12

5. 以下代码的输出结果是

```
union {
    double d;
    struct {
        int i;
        char c[4];
    } s;
} u;
u.d = 1;
printf("%d\n", u.s.c[2]);
```

A) 0　　B) -16　　C) 240　　D) 191

B

- 数组
  - C++引用
  - 指针运算
  - 类型转换
  - 高维数组
  - 指针练习
- 数据结构
  - 对齐
  - 结构体
  - **联合**

Practice

余文凯

The End