CS 261
Fall 2016

Mike Lam, Professor

x86-64 Data Structures

Topics

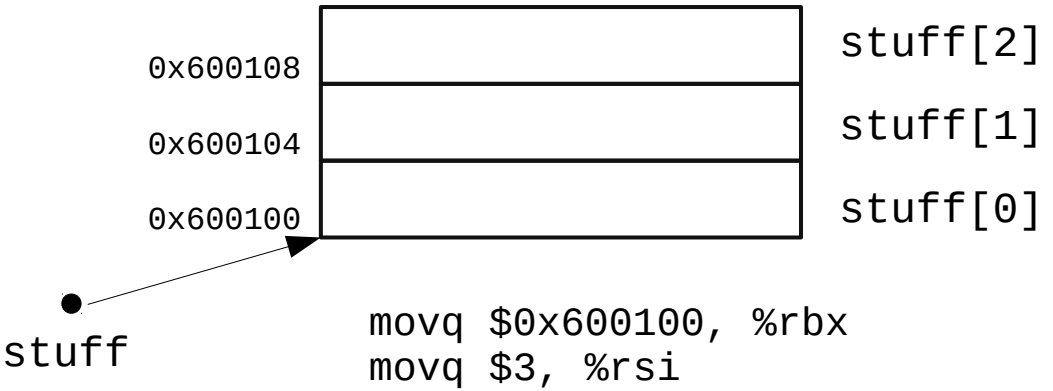
- Homogeneous data structures
  - Arrays
  - Nested / multidimensional arrays
- Heterogeneous data structures
  - Structs / records
  - Unions

Arrays

- An array is simply a block of memory
  - Fixed-sized *homogeneous* elements
  - Contiguous layout
  - Known length (but not stored as part of the array!)

```
uint32_t stuff[3];
```

3 elements
each element is 4 bytes wide
*total size is 3 * 4 = 12 bytes*




```
for (int i = 0; i < 3; i++) {
    stuff[i] = 5;
}
```

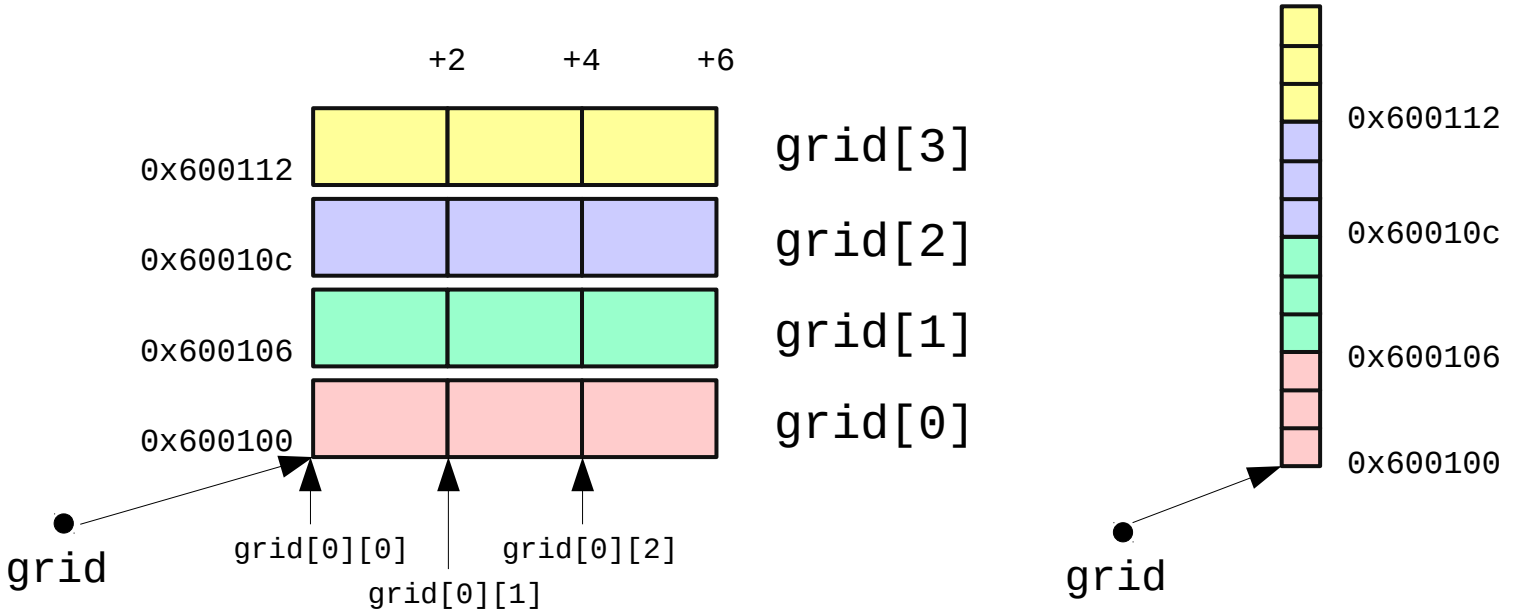
```
    movq $0x600100, %rbx
    movq $3, %rsi
    xorq %rdi, %rdi

loop:
    movq $5, (%rbx, %rdi, 4)
    addq $1, %rdi
    cmpq %rsi, %rdi
    jl loop
```

Arrays and pointers

- Array name is essentially a pointer to first element (base)
  - The *i*th element is at address (base + size * *i*)
- C pointer arithmetic uses intervals of the element width
  - No need to explicitly multiply by size in C
  - "`stuff+0`" or "`stuff`" is the address of the first element
  - "`stuff+1`" is the address of the second element
  - "`stuff+2`" is the address of the third element
- Indexing = pointer arithmetic plus dereferencing
  - "`stuff[i]`" means "`*(stuff + i)`"
  - In assembly, use the scaled index addressing mode
    - (*base*, *index*, *scale*) → e.g., `(%rbx, %rdi, 4)` for 32-bit elements

Nested / multidimensional arrays

- Generalizes cleanly to multiple dimensions
  - Think of the elements of outer dimensions as being arrays of inner dimensions
  - “Row-major” order: outer dimension specified first
  - E.g., “`int16_t grid[4][3]`” is a 4-element array of 3-element arrays of 16-bit integers
  - 2D: Address of (*i*,*j*)th element is $(\text{base} + \text{size}(\text{cols} * i + j))$
  - 3D: Address of (*i*,*j*,*k*)th element is $(\text{base} + \text{size}((n_{d1} * n_{d2}) * i + n_{d2} * j + k))$



Compiler optimizations

(a) Original C code

```
1   /* Compute i,k of variable matrix product */
2   int var_prod_ele(long n, int A[n][n], int B[n][n], long i, long k) {
3       long j;
4       int result = 0;
5
6       for (j = 0; j < n; j++)
7           result += A[i][j] * B[j][k];
8
9       return result;
10  }
```

(b) Optimized C code

```
/* Compute i,k of variable matrix product */
int var_prod_ele_opt(long n, int A[n][n], int B[n][n], long i, long k) {
    int *Arow = A[i];
    int *Bptr = &B[0][k];
    int result = 0;
    long j;
    for (j = 0; j < n; j++) {
        result += Arow[j] * *Bptr;
        Bptr += n;
    }
    return result;
}
```

```
    Registers: n in %rdi, Arow in %rsi, Bptr in %rcx
               4n in %r9, result in %eax, j in %edx
1   .L24:                                   loop:
2     movl    (%rsi,%rdx,4), %r8d           Read Arow[j]
3     imull   (%rcx), %r8d                  Multiply by *Bptr
4     addl    %r8d, %eax                    Add to result
5     addq    $1, %rdx                      j++
6     addq    %r9, %rcx                     Bptr += n
7     cmpq    %rdi, %rdx                    Compare j:n
8     jne     .L24                          If !=, goto loop
```

Structs

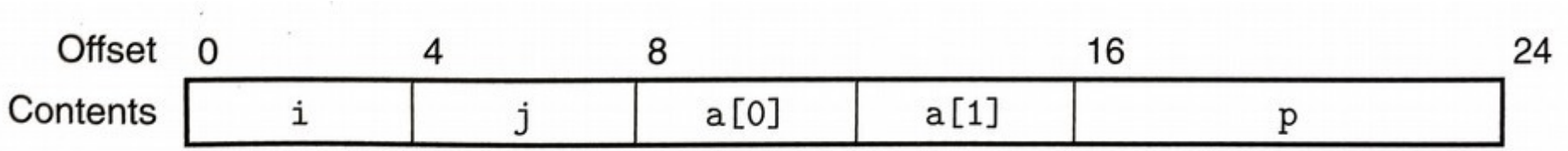
- C structs are also just regions of memory
  - "Structured" *heterogeneous* regions--they're split into fields
  - Contiguous layout (w/ occasional gaps for alignment)
  - Offset of each field can be determined by the compiler
  - Sometimes called "records" generally

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

```
x.i = 1;
x.j = 2;
x.a[0] = 3;
x.a[1] = 4;
x.p = NULL;
```

(%rbx = &x *and* %rdi = 1)

```
movl $1, (%rbx)
movl $2, 4(%rbx)
movl $3, 8(%rbx)
movl $4, 8(%rbx, %rdi, 4)
movq $0, 16(%rbx)
```



Union

- C unions are also just regions of memory
  - Can store one “thing”, but it could be multiple sizes depending on what kind of “thing” it currently is
  - All “fields” start at offset zero
  - Generally a bad idea! (circumvents the type system in C)
  - Can be used to do OOP in C (i.e., polymorphism)

```
typedef enum { CHAR, INT, FLOAT } objtype_t;

typedef struct {
    objtype_t type;
    union {
        char c;
        int i;
        float f;
    } data;
} obj_t;
```

```
obj_t foo;

foo.type = INT;
foo.data.i = 65;

printf(“%c”, foo.data.c);  ← VALID!
```

Alignment

- Alignment restrictions require addresses be *n*-divisible
  - E.g., 4-byte alignment means all addresses must be divisible by 4
  - Specified using an assembler directive
  - Improves memory performance if the hardware matches
  - Can be avoided in C using "`attribute (packed)`" (as in `elf.h`)

```
struct {
    int i;
    char c;
    int j;
} rec;
```