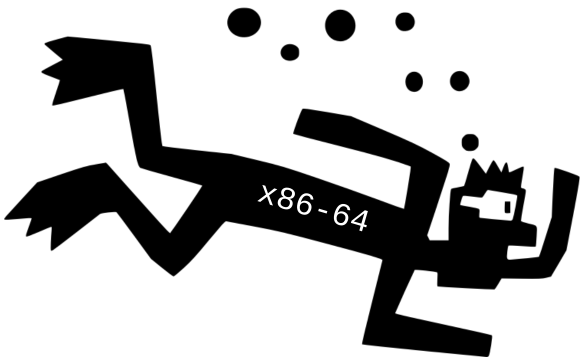
CS 261
Fall 2022

Mike Lam, Professor

Q. Why do assembly programmers need to know how to swim?

A. Because they work below C level!



x86-64 Miscellaneous Topics

Topics

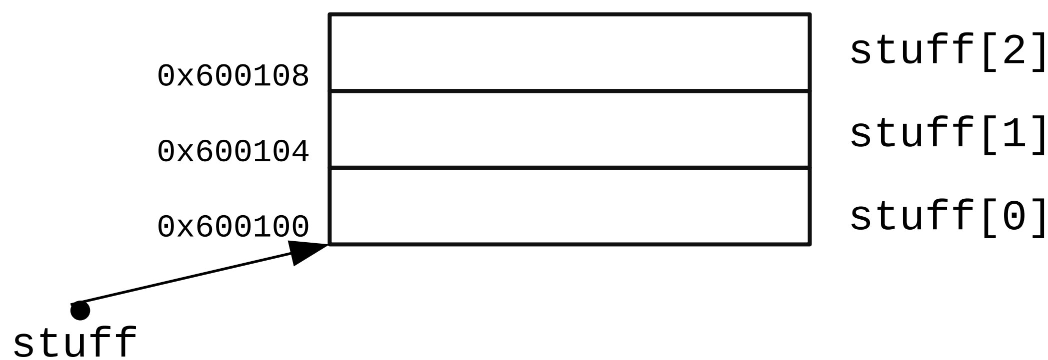
- Homogeneous data structures
  - Arrays
  - Nested / multidimensional arrays
- Heterogeneous data structures
  - Structs / records
  - Unions
- Floating-point code

Arrays

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

```
int32_t stuff[3];
```

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




```
stuff[0] = 7
stuff[1] = 7
stuff[2] = 7
```

```
movq $0x600100, %rbx
movl $7, (%rbx)
movl $7, 4(%rbx)
movl $7, 8(%rbx)
```

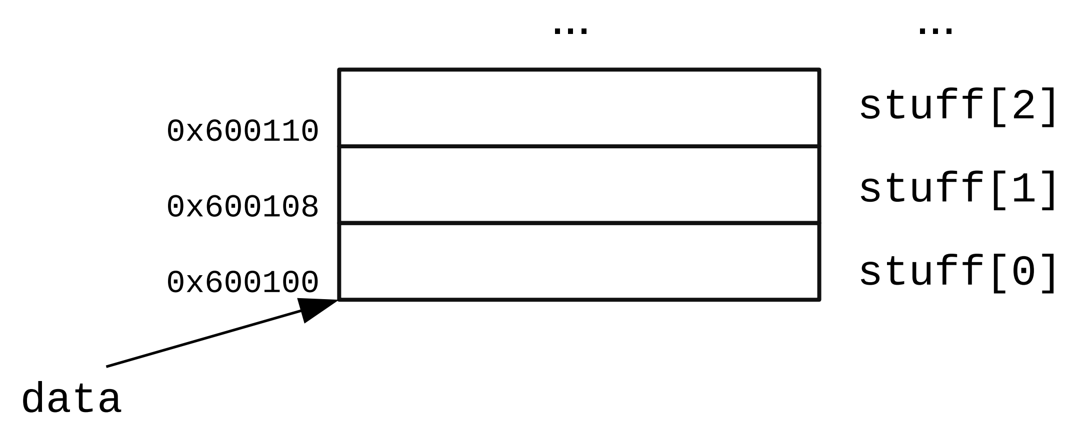
Arrays and pointers

- 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

Question

- Fill in the blank to correctly translate the following C code into x86-64:

```
int64_t data[10];
```



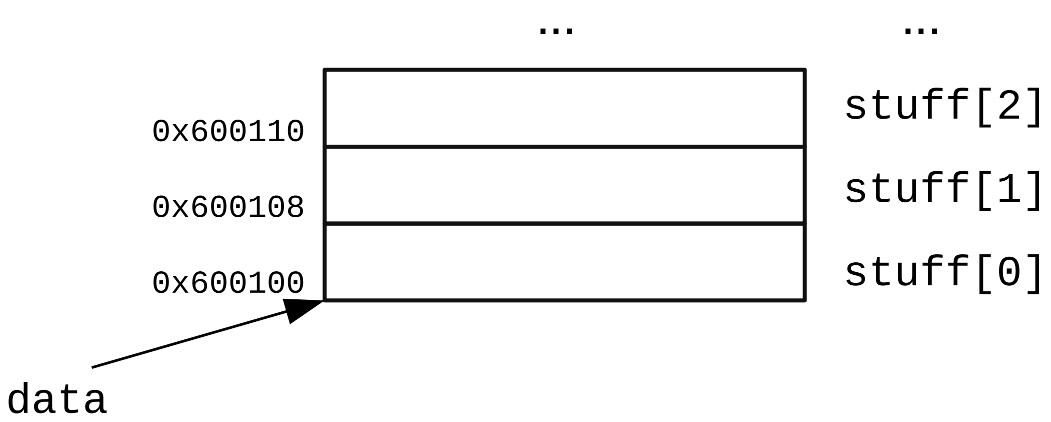

```
for (int i = 0; i < 10; i++) {
    data[i] = 0;
}
```

```
  movq $0x600100, %rbx
  movq $0, %rdx
  jmp L2
L1:
  movq $0, ___________
  incq %rdx
L2:
  cmpq $10, $rdx
  jl L1
```

Question

- Fill in the blank to correctly translate the following C code into x86-64:

```
int64_t data[10];
```

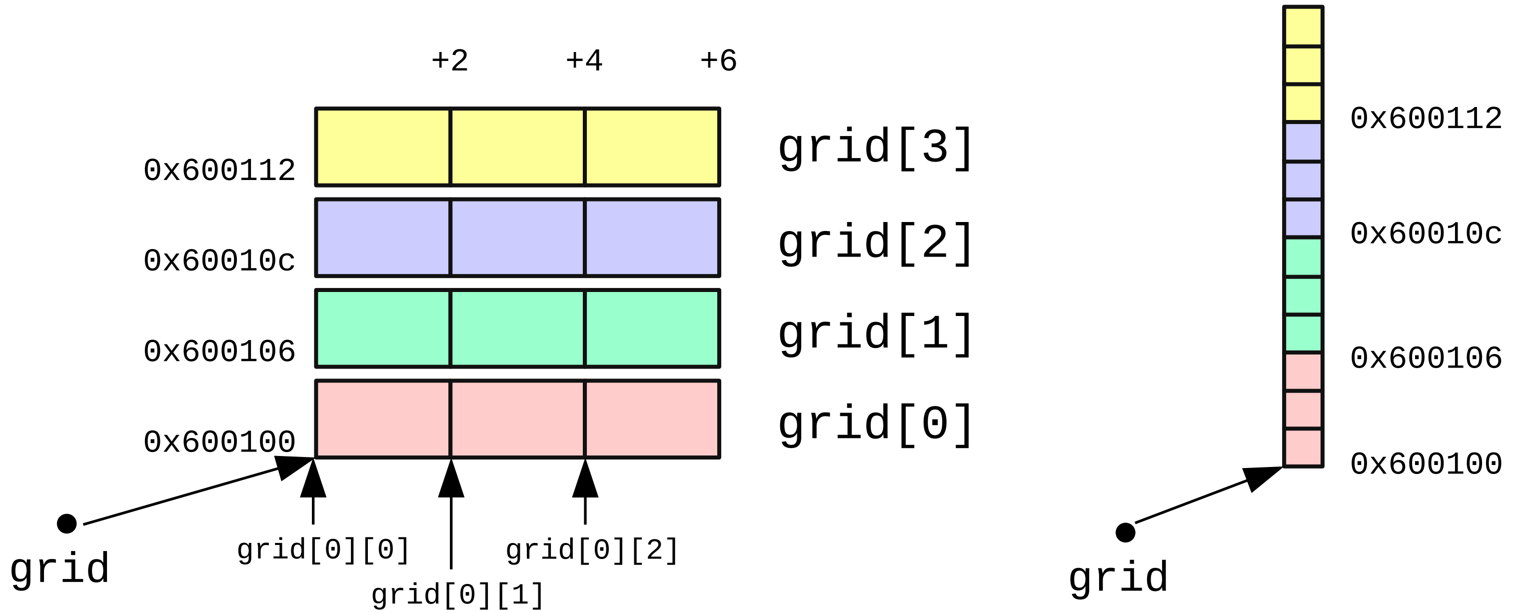



```
for (int i = 0; i < 10; i++) {
    data[i] = 0;
}
```

```
  movq $0x600100, %rbx
  movq $0, %rdx
  jmp L2
L1:
  movq $0, (%rbx, %rdx, 8)
  incq %rdx
L2:
  cmpq $10, $rdx
  jl L1
```

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 $(\text{base} + \text{size}((n_{d1} * n_{d2}) * i + n_{d2} * j + k))$



Structs

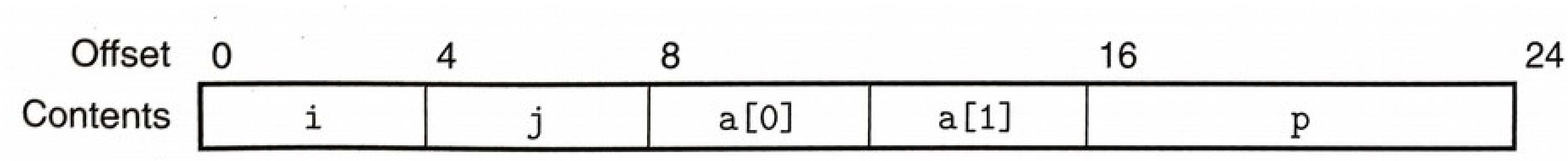
- C structs are also just regions of memory
  - “Structured” *heterogeneous* regions--they’re split into fields
  - Contiguous layout (w/ occasional gaps for alignment)
  - Offset of each field can be determined by the compiler
  - Sometimes called “records” generally

```
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)

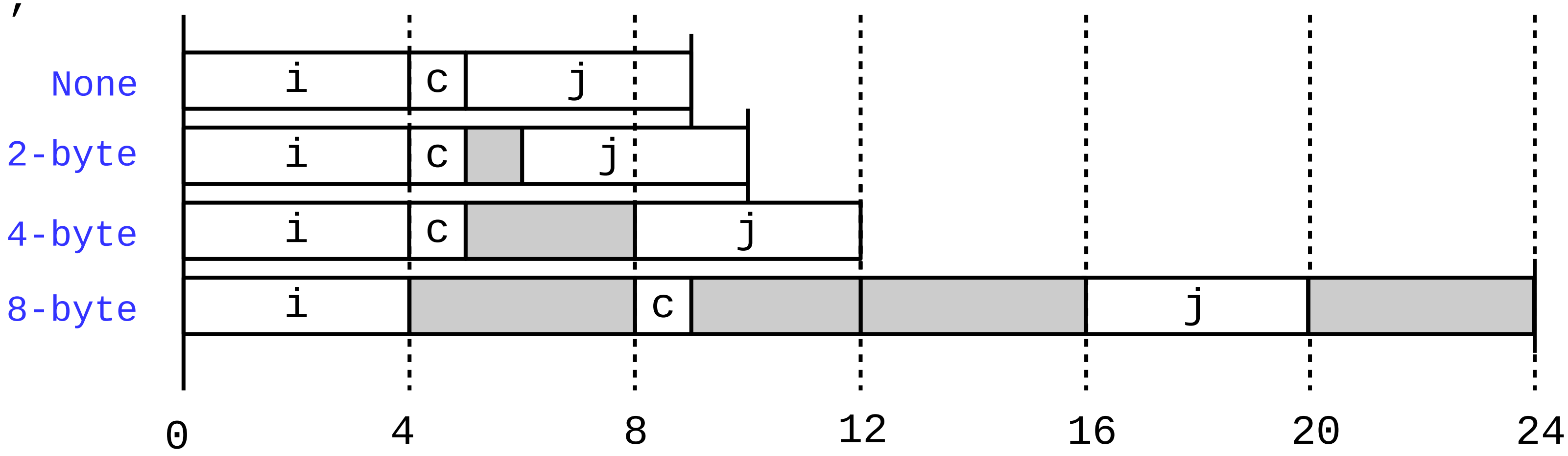
```
movl $1, (%rbx)
movl $2, 4(%rbx)
movl $3, 8(%rbx)
movl $4, 8(%rbx, %rdi, 4)
movq $0, 16(%rbx)
```



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;
```



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 (so context is even more important!)
  - 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!
```

Aside: Enums

- Enumerations are types where all values are listed
  - Declared in C using `enum` keyword
  - In C, the actual values are stored as integers
  - Can assign integer values if desired
  - Primary advantage: named constants

```
typedef enum {
   MON = 1, TUE, WED, THU, FRI, SAT, SUN
} day_t;

// essentially the same as: int midterm_day = 3;
day_t midterm_day = WED;
```

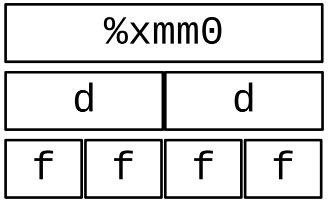
Floating-point code

- x87: extension of x86 for floating-point arithmetic
  - Originally for the 8087 floating-point co-processor
  - Adds new floating-point "stack" registers `ST(0)` – `ST(7)`
    - 80-bit extended double format (15 exponent and 63 significand bits)
  - Push/pop with `FLD` and `FST` instructions
  - Arithmetic: `FADD`, `FMUL`, `FSQRT`, etc.
  - Largely deprecated now in favor of new SIMD architectures

Floating-point code

- Single-Instruction, Multiple-Data (SIMD)
  - Performs the same operation on multiple pairs of elements
  - Also known as vector instructions
- Various floating-point SIMD instruction sets
  - MMX, **SSE**, **SSE2**, SSE3, SSE4, SSE5, **AVX**, **AVX2**
  - 16 new extra-wide XMM (128-bit) or YMM (256-bit) registers for holding multiple elements
    - Floating-point arguments passed in `%xmm0-%xmm7`
    - Return value in `%xmm0`
    - All registers are caller-saved

Floating-point code



- SSE (Streaming SIMD Extensions)
  - 128-bit XMM registers
    - Can store two 64-bit doubles or four 32-bit floats
  - New instructions for movement and arithmetic
    - General form: <*op*><*s*|*p*><s|*d*>
    - <*s*|*p*>: s=scalar (single data) p=packed (multiple data)
    - <*s*|*d*>: s=single (32-bit) d=double (64-bit)
    - E.g., “`addsd`” = add scalar 64-bit doubles
    - E.g., “`mulps` = multiply packed 32-bit floats
- AVX (Advanced Vector Extensions)
  - 256-bit YMM registers
    - Can store four 64-bit doubles or eight 32-bit floats
  - Similar instructions as SSE (but with “v” prefix, e.g., `vmulps`)

SSE/AVX

- **Movement**
  - `movss / movsd`
  - `movaps / movapd`
- **Conversion**
  - `cvtsi2ss / cvtsi2sd`
  - `cvtss2si / cvtsd2si`
  - `cvtss2sd / cvtsd2ss`
- **Arithmetic**
  - `addss / addsd`
  - `addps / addpd`

    `… (sub, mul, div, max, min, sqrt)`
  - `andps / andpd`
  - `xorps / xorpd`
- **Comparison**
  - `ucomiss / ucomisd`

(AVX has "v___" opcodes)

Bitwise operations in SSE/AVX

- Assembly instructions provide low-level access to floating-point numbers
  - Some numeric operations can be done more efficiently with simple bitwise operations
- AKA: Floating-Point Hacks™
  - Set to zero (value XOR value)
  - Absolute value (value AND `0x7fffffff`)
  - Additive inverse (value XOR `0x80000000`)
- Lesson: Information = Bits + Context
    - *(even if it wasn't the intended context!)*

Preview: Y86-64 ISA

Byte	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
rrmovq rA, rB	2	0	rA	rB						
irmovq V, rB	3	0	F	rB	V					
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D					
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn	Dest							
cmovXX rA, rB	2	fn	rA	rB						
call Dest	8	0	Dest							
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

Not in CS:APP:
`iotrap` **id**

C	id

Operations

addq	6	0
subq	6	1
andq	6	2
xorq	6	3

Branches

jmp	7	0	jne	7	4
jle	7	1	jge	7	5
jl	7	2	jg	7	6
je	7	3			

Moves

rrmovq	2	0	cmovne	2	4
cmovle	2	1	cmovge	2	5
cmovl	2	2	cmovg	2	6
cmove	2	3			

Number	Register name
0	%rax
1	%rcx
2	%rdx
3	%rbx
4	%rsp
5	%rbp
6	%rsi
7	%rdi

8	%r8
9	%r9
10	%r10
11	%r11
12	%r12
13	%r13
14	%r14

Value	Name	Meaning
1	AOK	Normal operation
2	HLT	halt instruction encountered
3	ADR	Invalid address encountered
4	INS	Invalid instruction encountered

RF: Program registers

%rax	%rsp	%r8	%r12
%rcx	%rbp	%r9	%r13
%rdx	%rsi	%r10	%r14
%rbx	%rdi	%r11	

CC: Condition codes

ZF	SF	OF

Stat: Program status

PC

DMEM: Memory