Arrays, Structs, and Memory

10/18/16

Recall: Indexed Addressing Mode

- General form:

```
offset(%base, %index, scale)
```

- Translation: Access the memory at address…

```
base + (index * scale) + offset
```

- Example:

```
-0x8(%ebp, %ecx, 0x4)
```

Translate this array access to IA32

```
int *x;
x = malloc(10*sizeof(int));

...

x[i] = -12;
```

At this point, suppose that the variable `x` is stored at `%ebp+8`. And `i` is in `%edx`. Use indexed addressing to assign into the array.

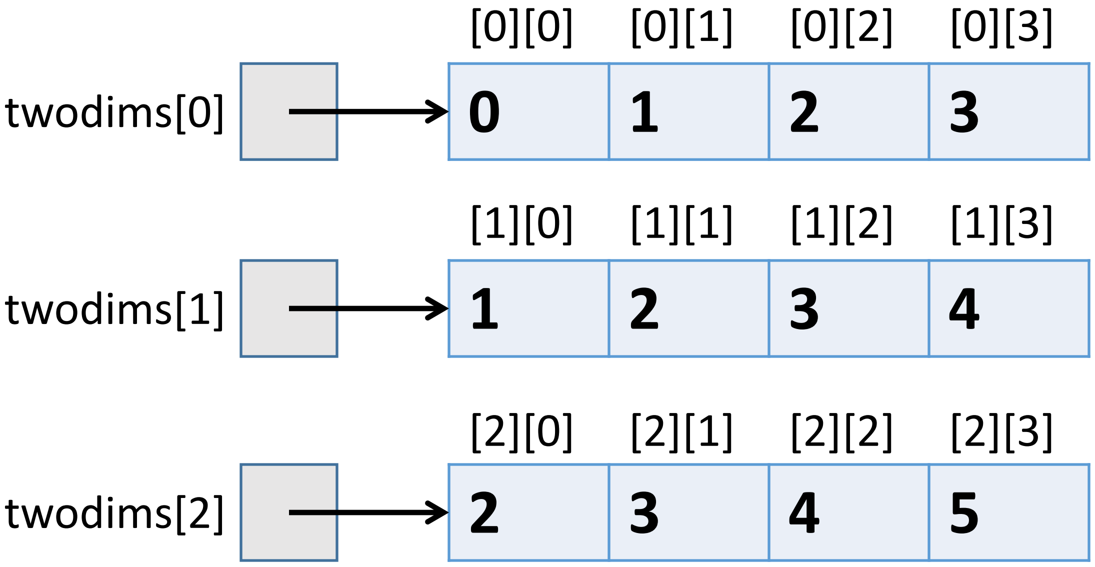
Two-dimensional Arrays

```
int twodims[3][4];
twodims[1][3] = 5;
```

- Technically an array of arrays of ints.
- “Give me three sets of four integers.”
- How are these organized in memory?

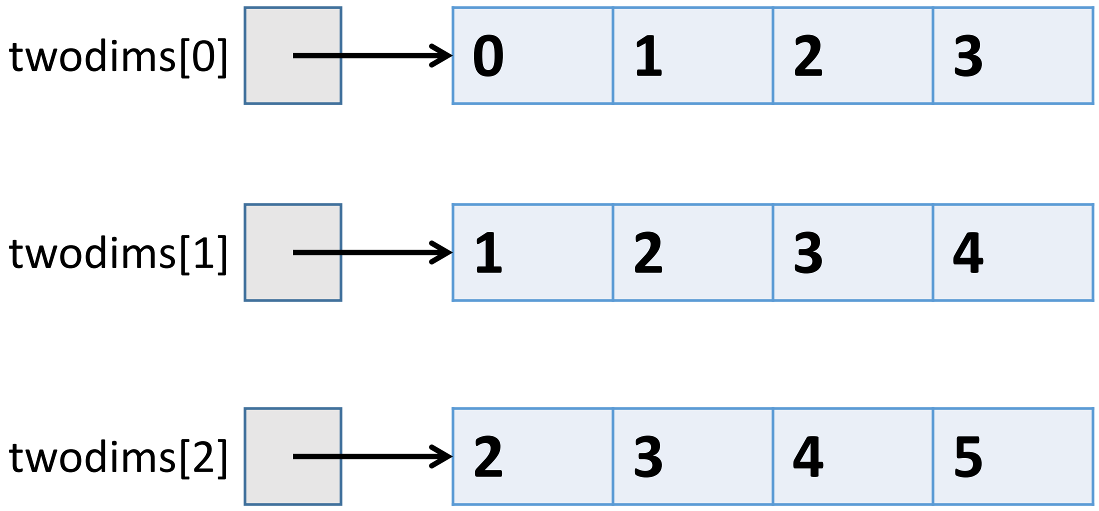
Two-dimensional Arrays

```
int twodims[3][4];
for(i=0; i<3; i++) {
   for(j=0; j<4; j++) {
      twodims[i][j] = i+j;
   }
}
```



Two-dimensional Arrays: Matrix

```
int twodims[3][4];
for(i=0; i<3; i++) {
   for(j=0; j<4; j++) {
      twodims[i][j] = i+j;
   }
}
```



Memory Layout

`int twodims[3][4];`

- Matrix: 3 rows, 4 columns

0	**1**	**2**	**3**
1	**2**	**3**	**4**
2	**3**	**4**	**5**

`twodims[1][3]:`

base addr + row offset + col offset

`twodims + 1*ROWSIZE*4 + 3*4`

`0xf260 + 16 + 12 = 0xf27c`

0xf260	0	twodim[0][0]
0xf264	1	twodim[0][1]
0xf268	2	twodim[0][2]
0xf26c	3	twodim[0][3]
0xf270	1	twodim[1][0]
0xf274	2	twodim[1][1]
0xf278	3	twodim[1][2]
0xf27c	4	twodim[1][3]
0xf280	2	twodim[2][0]
0xf284	3	twodim[2][1]
0xf288	4	twodim[2][2]
0xf28c	5	twodim[2][3]

Memory Layout

```
int twodims[3][4];
```

- Matrix: 3 rows, 4 columns

0	**1**	**2**	**3**
1	**2**	**3**	**4**
2	**3**	**4**	**5**

<u>Row Major Order</u>:
all Row 0 buckets,
followed by
all Row 1 buckets

0xf260	0	twodim[0][0]
0xf264	1	twodim[0][1]
0xf268	2	twodim[0][2]
0xf26c	3	twodim[0][3]
0xf270	1	twodim[1][0]
0xf274	2	twodim[1][1]
0xf278	3	twodim[1][2]
0xf27c	4	twodim[1][3]
0xf280	2	twodim[2][0]
0xf284	3	twodim[2][1]
0xf288	4	twodim[2][2]
0xf28c	5	twodim[2][3]

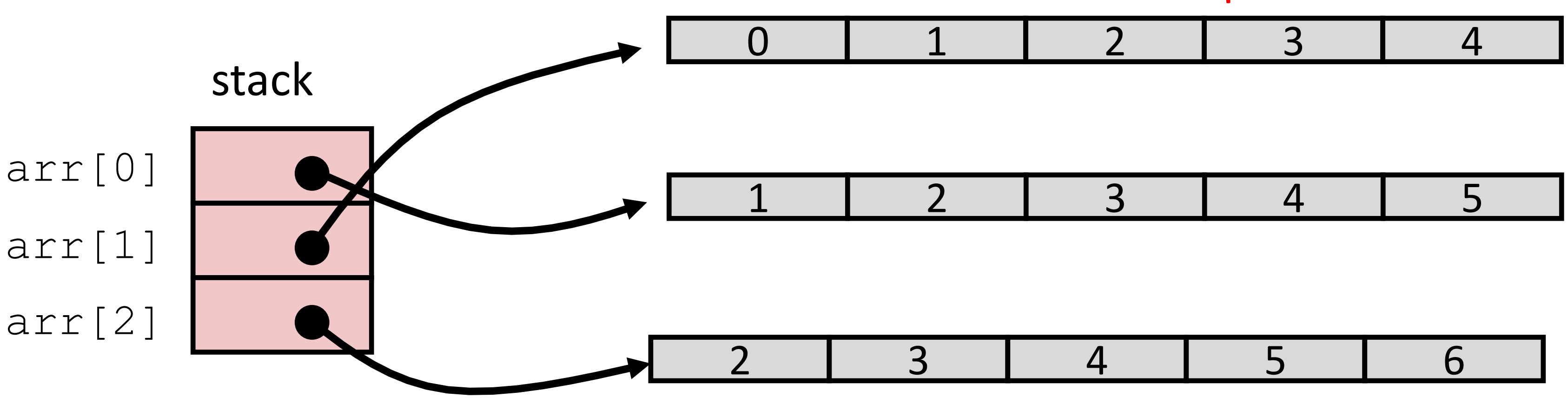
If we declared `int matrix[5][3];`, and the base of matrix is `0x3420`, what is the address of `matrix[3][2]`?

A. 0x3438
B. 0x3440
C. 0x3444
D. 0x344C
E. None of these

2D Arrays Another Way

```
char *arr[3]; // array of 3 char *'s
for(i=0; i<3; i++) {
  arr[i] = malloc(sizeof(char)*5);
  for(j=0; j<5; j++) {
    arr[i][j] = i+j;
  }
}
```

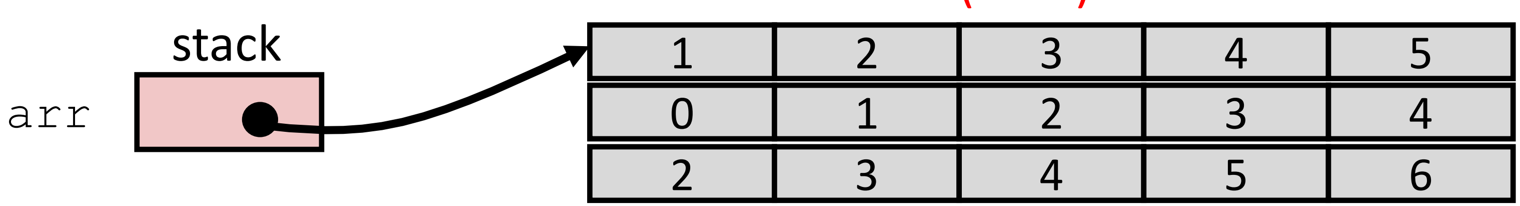
Heap: each malloc'ed array of 5 chars is contiguous, but three separately malloc'ed arrays, not necessarily → each has separate base address



2D Arrays Yet Another Way

```
char *arr;
arr = malloc(sizeof(char)*ROWS*COLS);
for(i=0; i< ROWS; i++) {
   for(j=0; j< COLS; j++) {
      arr[i*COLS+j] = i+j;
   }
}
```

Heap: all ROW*COLS buckets are contiguous
(allocated by a single malloc)
all buckets can be access from single
base address (addr)

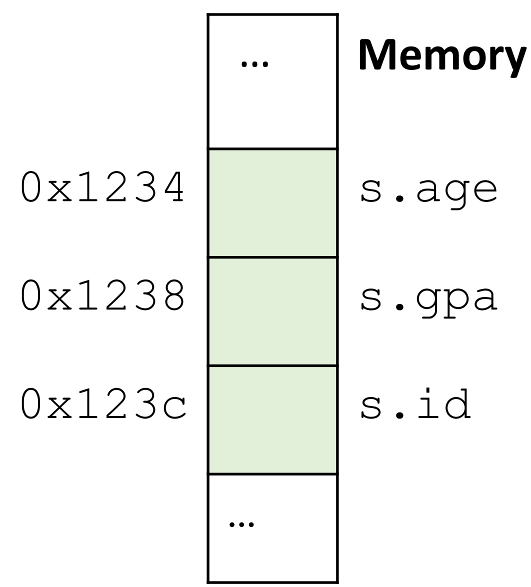


Structs

- Laid out contiguously by field
  - In order of field declaration.
  - May require some padding, for alignment.
- Struct fields accessible as a base + displacement
  - Compiler knows (constant) displacement of each field

```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
```



Data Alignment:

- Where (which address) can a field be located?

- <u>char (1 byte)</u>: can be allocated at any address:
  0x1230, 0x1231, 0x1232, 0x1233, 0x1234, …
- <u>short (2 bytes)</u>: must be aligned on 2-byte addresses:
  0x123**0**, 0x123**2**, 0x123**4**, 0x123**6**, 0x123**8**, …
- <u>int (4 bytes)</u>: must be aligned on 4-byte addresses:
  0x123**0**, 0x123**4**, 0x123**8**, 0x123**c**, 0x124**0**, …

Why do we want to align data on multiples of the data size?

A. It makes the hardware faster.

B. It makes the hardware simpler.

C. It makes more efficient use of memory space.

D. It makes implementing the OS easier.

E. Some other reason.

Data Alignment: Why?

- Simplify hardware
  - e.g., only read ints from multiples of 4
  - Don’t need to build wiring to access 4-byte chunks at any arbitrary location in hardware

- Inefficient to load/store single value across alignment boundary (1 vs. 2 loads)

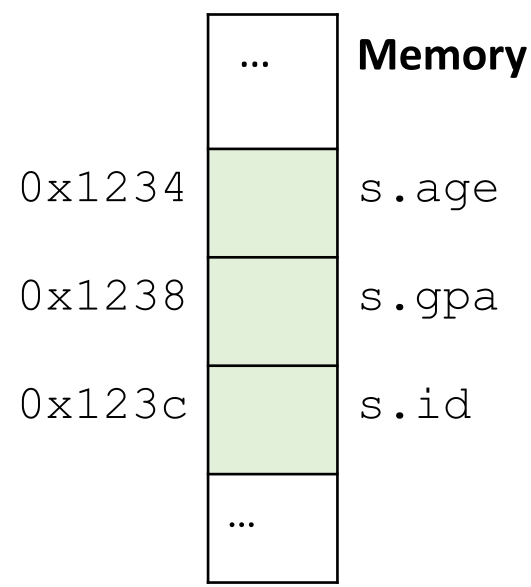
- Simplify OS:
  - Prevents data from spanning virtual pages
  - Atomicity issues with load/store across boundary

Structs

- Laid out contiguously by field
  - In order of field declaration.
  - May require some padding, for alignment.
- Struct fields accessible as a base + displacement
  - Compiler knows (constant) displacement of each field

```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
```



How much space do we need to store one of these structures?

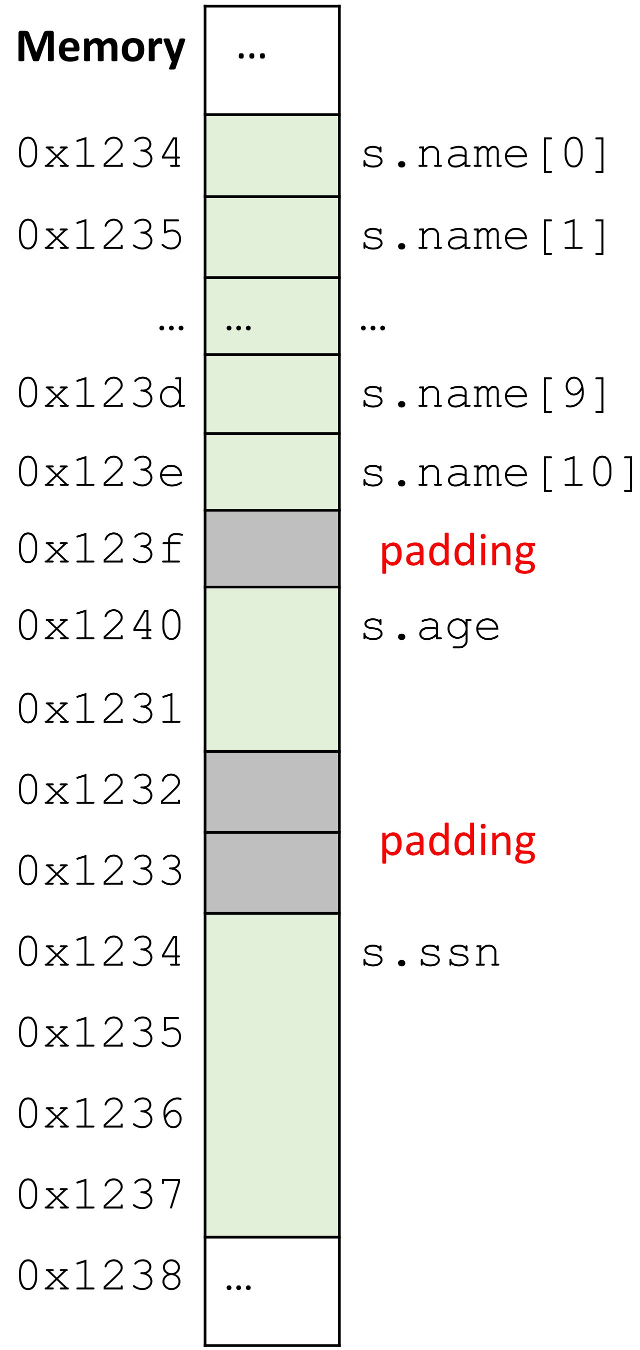
```
struct student{
  char name[11];
  short age;
  int id;
};
```

A. 17 bytes
B. 18 bytes
C. 20 bytes
D. 22 bytes
E. 24 bytes

Structs

```
struct student{
  char name[11];
  short age;
  int id;
};
```

- Size of data: 17 bytes
- Size of struct: 20 bytes



Use sizeof() when allocating structs with malloc()!

Alternative Layout

```
struct student{
   int id;
   short age;
   char name[11];
};
```

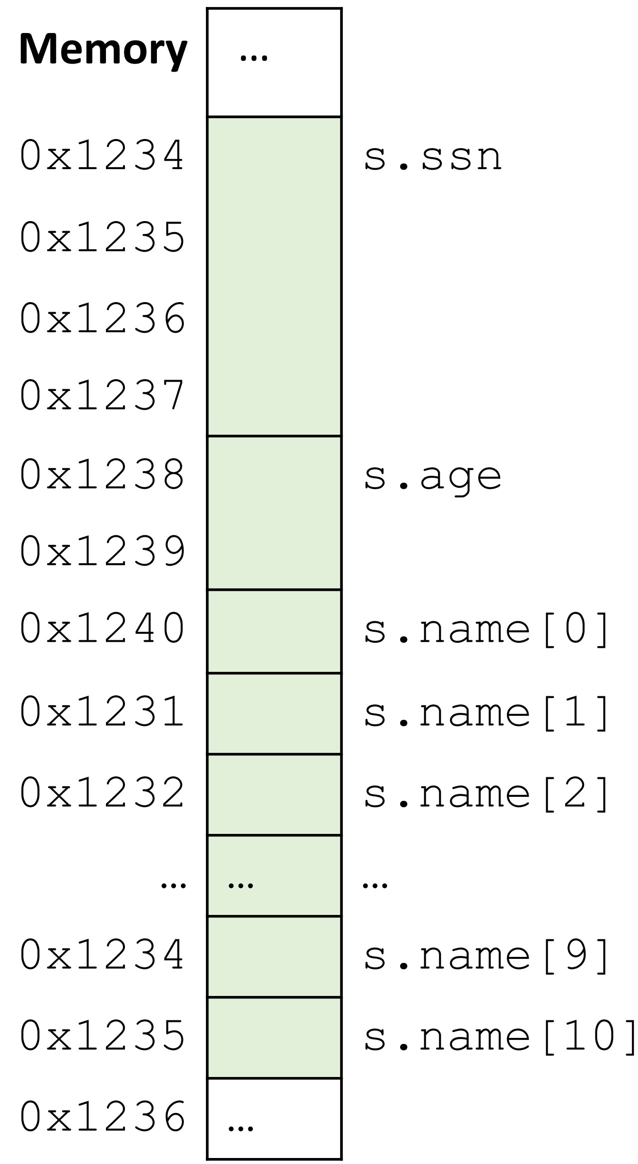
Same fields, declared in a different order.

Alternative Layout

```
struct student{
   int id;
   short age;
   char name[11];
};
```

- Size of data: 17 bytes
- Size of struct: 17 bytes!

In general, this isn't a big deal on a day-to-day basis. Don't go out and rearrange all your struct declarations.



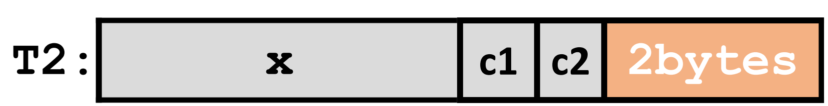
Cool, so we can get rid of this padding by being smart about declarations?

- Answer: Maybe.

- Rearranging helps, but often padding after the struct can't be eliminated.

```
struct T1 {
    char c1;
    char c2;
    int  x;
};
```

```
struct T2 {
    int x;
    char c1;
    char c2;
};
```

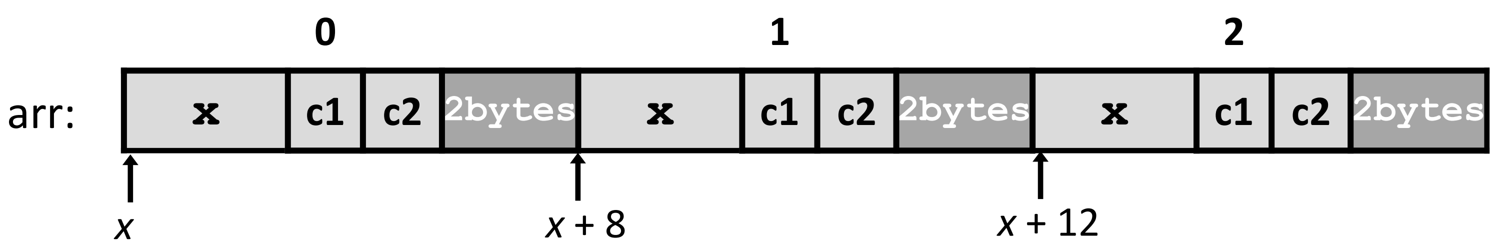


"External" Padding

- Array of Structs

  Field values in each bucket must be properly aligned:

```
struct T2 arr[3];
```



Buckets must be on a 4-byte aligned address

Which instructions would you use to access the age field of students[8]?

```
struct student {
  int id;
  short age;
  char name[11];
};

struct student students[20];

students[8].age = 21;
```

Assume the base of students is stored in register %edx.

Stack Padding

- Memory alignment applies elsewhere too.

```
void func1(){
  int x;
  char ch[5];
  short s;
  double y;
  ...
}
```

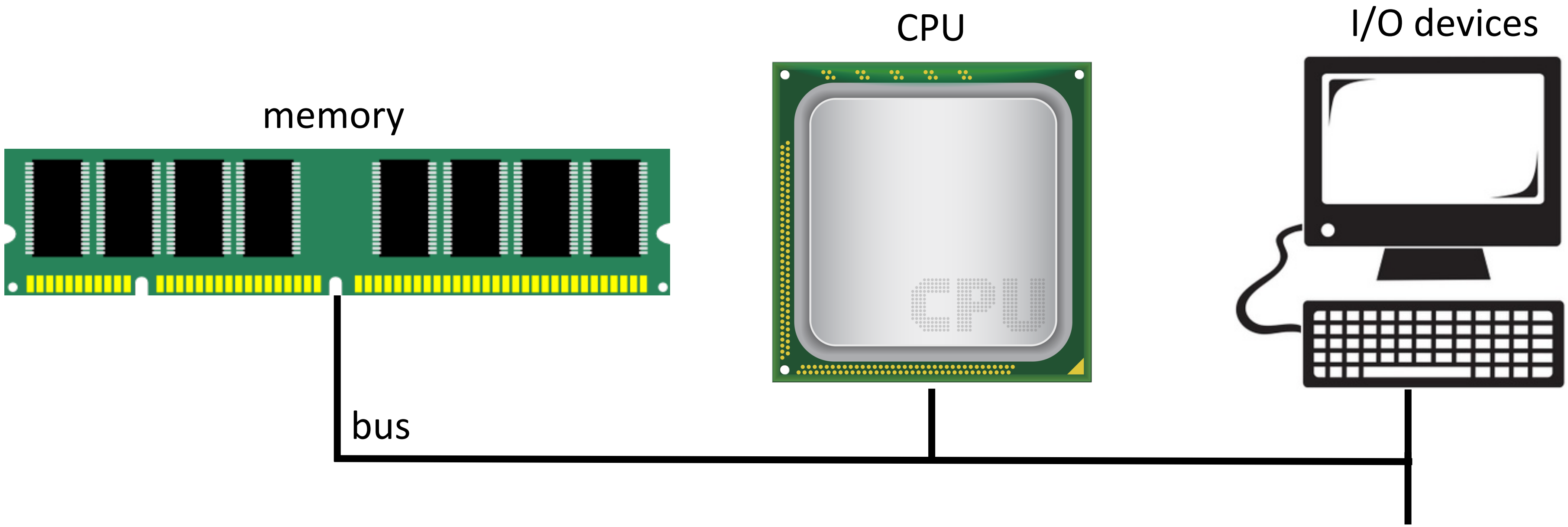
vs.

```
void func2(){
  double y;
  int x;
  short s;
  char ch[5];
  ...
}
```

What We’ve Learned

CS31: First Half

The Hardware Level



- Basic Hardware Units:
  - Processor
  - Memory
  - I/O devices
- Connected by buses.

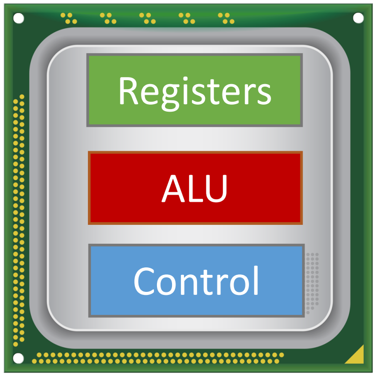
Foundational Concepts

- Von Neumann architecture
  - Programs are data.
  - Programs and other data are stored in main memory.

- Binary data representation
  - Data is encoded in binary.
    - Two’s complement
    - ASCII
    - etc.
  - Instructions are encoded in binary.
    - Opcode
    - Source and destination addresses

Architecture and Digital Circuits

- Circuits are built from logic gates.
  - Basic gates: AND, OR, NOT, …
- Three types of circuits:
  - Arithmetic/Logic
  - Storage
  - Control
- The CPU uses all three types of circuits.
- Clock cycle drives the system.
  - One instruction per clock cycle.
- ISA defines which operations are available.



Assembly Language

- Assembly instructions correspond closely to CPU operations.
- Compiler converts C code to assembly instructions.
- Types of instructions:
  - Arithmetic/logic: ADD, OR, …
  - Control Flow: JMP, CALL
  - Data Movement: MOV, (and fake data mvmt: LEAL)
  - Stack & Functions: PUSH, POP, CALL, LEAVE, RET
- Many ways to compile the same program.
  - Conventions govern choices that need to be consistent.
    - Location of function arguments, return address, etc.

C Programming Concepts

- Arrays, structs, and memory layout.
- Pointers and addresses.
- Function calls and stack memory.
- Dynamic memory on the heap.

Some of the (many) things we've left out…

- EE level: wires and transistors.
- Optimizing circuits: time and area.
  - Example: a ripple carry adder has a long critical path; can we shorten it?
- Architecture support for complex instructions.
  - Often an assembly instruction requires multiple CPU operations.
- Compiler design.
  - The compiler automates C →IA32 translation. How does this work? How can it be made efficient?

Midterm Info

- Arrive early on Thursday. We will start right at 1:15.
- Bring a pencil.
  - Please don't use a pen.
- Closed notes, but you may bring the following:
  - IA32 cheat sheet
  - IA32 stack diagram
- Q&A-style review session in lab tomorrow.
  - I will not prepare slides for this.
  - You need to prepare questions to make this useful.

Midterm Tips

- Don’t leave questions blank: a partial answer is better than none.
- If you don’t understand a question, ask for clarification during exam.
- If you’re not sure how to do problem, move on and come back later.
- Use a question’s point value as rough guide for how much time to spend on it.
- Review your answers before turning in the exam.
- Show your work for partial credit.