CS 31: Intro to Systems C Programming
L14: Arrays, Structs, Strings, and Pointers

Vasanta Chaganti & Kevin Webb
Swarthmore College
Nov 2, 2023
Announcements

• HW 3 is due today before class
Today

• Accessing *things* via an offset
  • Arrays, Structs, Unions
  • Connect accessing them in C with what we know about assembly

• How complex structures are stored in memory
  • Multi-dimensional arrays & Structs
So far: Primitive Data Types

• We’ve been using ints, floats, chars, pointers

• Simple to place these in memory:
  – They have an unambiguous size
  – They fit inside a register*
  – The hardware can operate on them directly

(*There are special registers for floats and doubles that use the IEEE floating point format.)
Composite Data Types

• Combination of one or more existing types into a new type. (e.g., an array of *multiple* ints, or a struct)
Addressing Mode: Indexed

The offset (%rcx) can also be scaled by a constant.

```
mov (%rax, %rcx, 4), %rdx
```

- Take the base address: %rax
- Multiply the offset by the scale: %rcx * 4
- Add the scaled offset to the base: %rax + %rcx * 4
- Now, index into memory at (%rax + %rcx * 4) and store the result in %rdx.

One register to keep track of base address.
One register to keep track of offset from base address.
Scale Constant
Let’s try an example

Suppose:

```c
int iptr; is stored in register %rax.
int i=2; is stored at %rbp-8
iptr[i] = 9; //iptr[2] = 9;
```

In assembly:

```assembly
mov -8(%rbp), %rcx
mov %rdx, (rax, rcx, 4)
```

<table>
<thead>
<tr>
<th>Registers:</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax 0x0824</td>
<td></td>
</tr>
<tr>
<td>rcx</td>
<td></td>
</tr>
<tr>
<td>rdx 9</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0824:</td>
<td>iptr[0]</td>
</tr>
<tr>
<td>0x0828:</td>
<td>iptr[1]</td>
</tr>
<tr>
<td>0x082C:</td>
<td>iptr[2]</td>
</tr>
<tr>
<td>0x0830:</td>
<td>iptr[3]</td>
</tr>
</tbody>
</table>

Let's try an example
Let’s try an example

Suppose:

```c
int iptr; is stored in register %rax.
int i=2; is stored at %rbp-8
iptr[i] = 9; //iptr[2] = 9;
```

In assembly:

```assembly
mov -8(%rbp), %rcx
mov %rdx, (rax, rcx, 4) = add (rcx *4) = add (2*4) = add 8
```

Registers:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>0x0824</td>
</tr>
<tr>
<td>rcx</td>
<td></td>
</tr>
<tr>
<td>rdx</td>
<td>9</td>
</tr>
</tbody>
</table>

Heap

- 0x0824: iptr[0]
- 0x0828: iptr[1]
- 0x082C: iptr[2]
- 0x0830: iptr[3]
What happens when we increment i?
What changes do we make in assembly?

Suppose:
- `int iptr;` is stored in register `%rax`.
- `int i=3;` is stored at `%rbp-8`.
- `iptr[i] = 10; // iptr[3] = 10;`

In assembly:
- `mov -8(%rbp), %rcx`
- `mov %rdx, (%rax, rcx, 4)`

= add (rcx * 4)
= add (2*4)
= add 8

Heap:
- `0x0824: iptr[0]`
- `0x0828: iptr[1]`
- `0x082C: iptr[2]`
- `0x0830: iptr[3]`
Suppose:

```plaintext
int iptr;

int i = 3;

iptr[i] = 10;  // iptr[3] = 10;
```

In assembly:

```assembly
mov -8(%rbp), %rcx
mov %rdx, (rax, rcx, 4)
```

What happens when we increment `i`?

What changes do we make in assembly?

Compiler can simply increment register `rcx` and access the next element of the array with the same assembly commands!
Two-dimensional Arrays

• Why stop at an array of ints?
  How about an array of arrays of ints?

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

• “Give me three sets of four integers.”

• How should these be organized in memory?
Two-dimensional Arrays

```c
int twodims[3][4];
for(i=0; i<3; i++) {
    for(j=0; j<4; j++) {
        twodims[i][j] = i+j;
    }
}
```
Two-dimensional Arrays

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

```
  |   |   |   |   |
---|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 1 | 2 | 3 | 4 |
| 2 | 3 | 4 | 5 |
```
Memory Layout

- Matrix: 3 rows, 4 columns

Row Major Order:
all Row 0 buckets, followed by
all Row 1 buckets, followed by
all Row 2 buckets, ...
Memory Layout

- Matrix: 3 rows, 4 columns

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
1 & 2 & 3 & 4 \\
2 & 3 & 4 & 5 \\
\end{array}
\]

twodim[1][3]:

\[
\text{base addr} + \text{row offset (} # \text{ rows} \times \text{rows} \times \text{sizeof(int)}\) + \text{col offset}
\]

twodim + 1*ROWSIZE*4 + 3*4

0xf260 + 16 + 12

= 0xf27c
Memory Layout

- Matrix: 3 rows, 4 columns

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

twodim[1][3]:

```
base_addr + row offset (# rows * rows * sizeof(int)) + col offset
```

twodim + 1*ROWSIZE*4 + 3*4

\[
\begin{align*}
0xf260 + 16 + 12 &= 0xf27c \\
\end{align*}
\]

You do not need to convert mem index into an address for the lab!
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

Here is a table for reference:

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3420</td>
<td>0</td>
<td>matrix[0][0]</td>
</tr>
<tr>
<td>0x3424</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>0x3480</td>
<td>2</td>
<td>matrix[5][3]</td>
</tr>
</tbody>
</table>

**base addr**
+ row offset (# rows * row_size * sizeof(data_type))
+ col offset
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

Mem_index = 3*3+2 = 11 (you need this for the lab)  
Mem. address = 0x3420 + 11*4 (2c) = 0x344c
Dynamic Two-dimensional Array

• Given the **row-major order** layout, a "two-dimensional array" is still just a contiguous block of memory:

• The malloc function returns... a pointer to a contiguous block of memory!

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Index</th>
<th>Array Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xf260</td>
<td>0</td>
<td>twodim[0][0]</td>
<td></td>
</tr>
<tr>
<td>0xf264</td>
<td>1</td>
<td>twodim[0][1]</td>
<td></td>
</tr>
<tr>
<td>0xf268</td>
<td>2</td>
<td>twodim[0][2]</td>
<td></td>
</tr>
<tr>
<td>0xf26c</td>
<td>3</td>
<td>twodim[0][3]</td>
<td></td>
</tr>
<tr>
<td>0xf270</td>
<td>1</td>
<td>twodim[1][0]</td>
<td></td>
</tr>
<tr>
<td>0xf274</td>
<td>2</td>
<td>twodim[1][1]</td>
<td></td>
</tr>
<tr>
<td>0xf278</td>
<td>3</td>
<td>twodim[1][2]</td>
<td></td>
</tr>
<tr>
<td>0xf27c</td>
<td>4</td>
<td>twodim[1][3]</td>
<td></td>
</tr>
<tr>
<td>0xf280</td>
<td>2</td>
<td>twodim[2][0]</td>
<td></td>
</tr>
<tr>
<td>0xf284</td>
<td>3</td>
<td>twodim[2][1]</td>
<td></td>
</tr>
<tr>
<td>0xf288</td>
<td>4</td>
<td>twodim[2][2]</td>
<td></td>
</tr>
<tr>
<td>0xf28c</td>
<td>5</td>
<td>twodim[2][3]</td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Two-dimensional Array

- For this example, with three rows and four columns:

```
int *matrix = malloc(3 * 4 * sizeof(int));
```

Caveat: the C compiler doesn't know that you're planning to use this block of memory with more one index (i.e., row and column).

Can't access: matrix[i][j]
Dynamic Two-dimensional Array

- For this example, with three rows and four columns:

```
int *matrix = malloc(3 * 4 * sizeof(int));

// Compute the offset manually
index = i * ROWSIZE + j;
matrix[index] = ...
```
Two-dimensional array alternative

• (Dynamically) Allocate an array of pointers. For each pointer, (dynamically) allocate an array.

• How do we get an array of pointers?
Two-dimensional array alternative

• If we want a dynamic array of ints:
  – declare int *array = malloc(N * sizeof(int))

• So... if we want an array of int pointers:
  – declare int **array = malloc(...)

Two-dimensional array alternative

• If we want a dynamic array of ints:
  – declare int *array = malloc(N * sizeof(int))

• So... if we want an array of int pointers:
  – declare int **array = malloc(N * sizeof(int *))
  – The type of array[0], array[1], etc. is: int *
  – For each one of those, we can malloc an array of ints:
    • array[0] = malloc(M * sizeof(int))
int **two_d_array;

two_d_array = malloc(sizeof(int *) * N);
for (i=0; i < N; i++) {
    two_d_array[i] = malloc(sizeof(int) * M);
}
Two-dimensional arrays

- We'll use BOTH methods in future labs.
Structs

• Multiple values (fields) stored together
  – Defines a new type in C's type system

• Laid out contiguously by field (with a caveat we'll see later)
  – In order of field declaration.
Structs

Laid out contiguously by field (with a caveat we'll see later)
– In order of field declaration.

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

struct student s;
```
Structs

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;
```
Structs

Struct fields accessible as a base + displacement
– Compiler knows (constant) displacement of each field

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

```c
struct student s;
```
Struct fields accessible as a **base + displacement**

In assembly:  `mov reg_value, 16(reg_base)`

Where:
- `reg_value` is a register holding the value to store (say, 12)
- `reg_base` is a register holding the base address of the struct

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

struct student s;
s.id = 12;
```

Given the starting address of a struct...

The id field is always at an offset of 8 forward from the start.
Structs

• Laid out contiguously by field
  – In order of field declaration.
  – May require some padding, for data alignment.

<table>
<thead>
<tr>
<th>Memory</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1234</td>
<td>s.age</td>
</tr>
<tr>
<td>0x1238</td>
<td>s.gpa</td>
</tr>
<tr>
<td>0x123c</td>
<td>s.id</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Data Alignment:

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

• **char (1 byte):** can be allocated at any address:
  
  0x1230, 0x1231, 0x1232, 0x1233, 0x1234, ...

• **short (2 bytes):**
  
  – must be aligned on 2-byte addresses:
  
  – 0x1230, 0x1232, 0x1234, 0x1236, 0x1238, ...

• **int (4 bytes):**
  
  – must be aligned on 4-byte addresses:
  
  – 0x1230, 0x1234, 0x1238, 0x123c, 0x1240, ...
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.
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.

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

struct student s;
```
struct student{
    char name[11];
    short age;
    int id;
};
How much space do we need to store one of these structures? Why?

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

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

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

Same fields, declared in a different order.
In general, this isn’t a big deal on a day-to-day basis. Don’t go out and rearrange all your struct declarations.

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

size of data: 17 bytes

size of struct: 17 bytes
Aside: Network Headers

• In networks, we attach metadata to packets
  – Things like destination address, port #, etc.

• Common for these to be a specific size/format
  – e.g., the first 20 bytes must be laid out like ...

• Naïvely declaring a struct might introduce padding, violate format.
Cool, so we can get rid of this struct padding by being smart about declarations?

A. Yes (why?)

B. No (why not?)
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.

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

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

T1: `c1 c2 2bytes x`
T2: `x c1 c2 2bytes`
"External" Padding

Array of Structs: Field values in each bucket must be properly aligned:

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

Buckets must be on a 8-byte aligned address

![Diagram showing array structure and alignment](image-url)
struct student {
    int id;
    short age;
    char name[11];
};

struct student s;

s.id = 406432;
s.age = 20;
strcpy(s.name, "Alice");
struct student {
    int id;
    short age;
    char name[11];
};
struct student *s = malloc(sizeof(struct student));

How do we get to the id and age?
struct student {
  int id;
  short age;
  char name[11];
};

struct student *s = malloc(sizeof(struct student));

What about this?

How do we get to the id and age?

Option 1: Works but ugly

(*s).id = 406432;
(*s).age = 20;
strcpy((*s).name, “Alice”);

Option 2: Use struct pointer dereference!

s->id = 406432;
s->age = 20;
strcpy(s->name, “Alice”);
Memory alignment applies elsewhere too!

\[
\begin{align*}
\text{int } x; & \quad \text{vs.} \quad \text{double } y; \\
\text{char } \text{ch}[5]; & \\
\text{short } s; & \\
\text{double } y; &
\end{align*}
\]

In nearly all cases, you shouldn't stress about this. The compiler will figure out where to put things.

Exceptions: networking, OS
• Declared like a struct, but only contains one field, rather than all of them.

• Struct: field 1 and field 2 and field 3 ...
• Union: field 1 or field 2 or field 3 ...

Intuition: you know you only need to store one of N things, don’t waste space.
**Unions**

```c
struct my_struct {
    char ch[2];
    int i;
    short s;
}
```

```c
union my_union {
    char ch[2];
    int i;
    short s;
}
```

*my_struct* in memory

*my_union* in memory

Same memory used for all fields!
Unions

```c
my_union u;

u.i = 7;
```

```c
union my_union {
    char ch[2];
    int i;
    short s;
}
```

Same memory used for all fields!

my_union in memory
Unions

```c
my_union u;

u.i = 7;
u.s = 2;
```

```c
union my_union {  
    char ch[2];  
    int i;  
    short s;  
}
```

Same memory used for all fields!

my_union in memory
Unions

```c
my_union u;

u.i = 7;
u.s = 2;
u.ch[0] = 'a';

u.i = 5;
```

Reading `i` or `s` here would be bad!

Same memory used for all fields!
Unions

my_union u;

u.i = 7;
u.s = 2;
u.ch[0] = ‘a’;

u.i = 5;

union my_union {
    char ch[2];
    int i;
    short s;
}

Same memory used for all fields!

Reading i or s here would be bad!
Unions

- You probably won’t use these often.
- Use when you need mutually exclusive types.
- Can save memory.

```c
union my_union {
    char ch[2];
    int i;
    short s;
}
```

Same memory used for all fields!

my_union in memory
Strings

• Strings are *character arrays*

• Layout is the same as:
  – char name[10];

• Often accessed as (char *)
String Functions

- C library has many built-in functions that operate on char *’s:
  - strcpy, strdup, strlen, strcat, strcmp, strstr

```c
char name[10];
strcpy(name, "CS 31");
```
String Functions

• C library has many built-in functions that operate on char *’s:
  – strcpy, strdup, strlen, strcat, strcmp, strstr

```c
char name[10];
strcpy(name, “CS 31”);
```

• Null terminator (\0) ends string.
  – We don’t know/care what comes after
String Functions

• C library has many built-in functions that operate on char *’s:
  – strcpy, strdup, strlen, strcat, strcmp, strstr

• Seems simple on the surface.
  – That null terminator is tricky, strings error-prone.
  – Strings used everywhere!

• You will implement use these functions in a future lab.
Up next…

• New topic: Storage and the Memory Hierarchy
Transition

• First half of course: hardware focus
  – How the hardware is constructed
  – How the hardware works
  – How to interact with hardware / ISA

• Up next: performance and software systems
  – Memory performance
  – Operating systems
  – Standard libraries (strings, threads, etc.)
Efficiency

• How to Efficiently Run Programs

• Good algorithm is critical...

• Many systems concerns to account for too!
  – The memory hierarchy and its effect on program performance
  – OS abstractions for running programs efficiently
  – Support for parallel programming
Efficiency

• How to **Efficiently** Run Programs

• Good algorithm is critical...

• Many systems concerns to account for too!
  – The memory hierarchy and its effect on program performance
  – OS abstractions for running programs efficiently
  – Support for parallel programming
Suppose you’re designing a new computer architecture. Which type of memory would you use? Why?

A. low-capacity (~1 MB), fast, expensive

B. medium-capacity (a few GB), medium-speed, moderate cost

C. high-capacity (100’s of GB), slow, cheap

D. something else (it must exist)

trade-off between capacity and speed
Classifying Memory

- Broadly, two types of memory:
  1. Primary storage: CPU instructions can access any location at any time (assuming OS permission)
  2. Secondary storage: CPU can’t access this directly
Random Access Memory (RAM)

• Any location can be accessed directly by CPU
  – Volatile Storage: lose power → lose contents

• Static RAM (SRAM)
  – Latch-Based Memory (e.g. RS latch), 1 bit per latch
  – Faster and more expensive than DRAM
    • “On chip”: Registers, Caches

• Dynamic RAM (DRAM)
  – Capacitor-Based Memory, 1 bit per capacitor
    • “Main memory”: Not part of CPU
Memory Technologies

• Static RAM (SRAM)
  – 0.5ns – 2.5ns, $2000 – $5000 per GB

• Dynamic RAM (DRAM)
  – 50ns – 100ns, $20 – $75 per GB
  (Main memory, “RAM”)

We’ve talked a lot about registers (SRAM) and we’ll cover caches (SRAM) soon. Let’s look at main memory (DRAM) now.
Dynamic Random Access Memory (DRAM)

Capacitor based:
- cheaper and slower than SRAM
- capacitors are leaky (lose charge over time)
- Dynamic: value needs to be refreshed (every 10-100ms)

Example: DIMM
(Dual In-line Memory Module):
Connecting CPU and Memory

• Components are connected by a **bus**:  
  • A bus is a collection of parallel wires that carry address, data, and control signals.  
  • Buses are typically shared by multiple devices.
How A Memory Read Works

(1) CPU places address A on the memory bus.

Load operation: \texttt{mov (Address A), %rax}
How A Memory Read Works

(2) Main Memory reads address A from memory, fetches value at that address and puts it on the bus

Sending the value back to the CPU
(3) CPU reads value from the bus, and copies it into register rax. A copy also goes into the on-chip cache memory.
How a Memory Write Works

1. CPU writes A to bus, memory reads it
2. CPU writes value to bus, memory reads it
3. Memory stores value at address A
Secondary Storage

• Disk, Tape Drives, Flash Solid State Drives, ...

• Non-volatile: retains data without a charge

• Instructions **CANNOT** directly access data on secondary storage
  – No way to specify a disk location in an instruction
  – Operating System moves data to/from memory
Secondary Storage

- CPU
  - ALU
  - Register (5)
  - CPU Cache

- Memory Module Slots

- Memory Bus

- I/O Bus (e.g., PCI)
  - SATA Controller
  - USB Controller
  - IDE Controller
  - ... (Secondary Storage Devices)
  - I/O Controller

- Path is much longer
What’s Inside A Disk Drive?

- Spindle
- Arm
- Actuator
- Platters
- Controller Electronics (includes processor & memory)
- R/W head
- Device Driver (part of OS code) interacts with Controller to R/W to disk

Data Encoded as points of magnetism on Platter surfaces
Data blocks located in some **Sector** of some **Track** on some **Surface**

1. Disk Arm moves to correct **track** (seek time)
2. Wait for **sector** spins under R/W head (rotational latency)
3. As sector spins under head, data are Read or Written (transfer time)
Memory Technology

- **Static RAM (SRAM)**
  - 0.5ns – 2.5ns, $2000 – $5000 per GB

- **Dynamic RAM (DRAM)**
  - 50ns – 100ns, $20 – $75 per GB

  Solid-state disks (flash): 100 us – 1 ms, $2 - $10 per GB

- **Magnetic disk**
  - 5ms – 15ms, $0.20 – $2 per GB

Like walking:
- Down the hall
- Across campus (to Cleveland / Indianapolis)
- To Seattle

1 ms == 1,000,000 ns
The Memory Hierarchy

- **Local secondary storage (disk)**
  - ~100 M cycles to access

- **Main memory (DRAM)**
  - ~100 cycles to access

- **Cache(s) (SRAM)**
  - ~10’s of cycles to access

- **Registers**
  - 1 cycle to access

- **On Chip Storage**

- **CPU instructions can directly access**

- **Smaller, Faster, Costlier per byte**

- **Larger, Slower, Cheaper per byte**
Where does accessing the network belong?

- Larger
- Slower
- Cheaper per byte

Local secondary storage (disk)
- ~100 M cycles to access

On Chip Storage
- ~100 cycles to access

Cache(s) (SRAM)
- ~10’s of cycles to access

Main memory (DRAM)
- ~100 cycles to access

Registers
- 1 cycle to access

CPU instrs can directly access

A: Here
B: Here
C: Somewhere else
The Memory Hierarchy

- **Local secondary storage (disk)**
  - Larger
  - Slower
  - Cheaper per byte
  - ~100 M cycles to access

- **Remote secondary storage (tapes, Web servers / Internet)**
  - Slower than local disk to access

- **Main memory (DRAM)**
  - ~100 cycles to access

- **Cache(s) (SRAM)**
  - ~10’s of cycles to access

- **On Chip Storage**
  - CPU instrs can directly access
  - Registers
  - 1 cycle to access

- **On Chip Storage**
  - CPU can access

**Summary**

- Smaller, Faster, Costlier per byte
- Larger, Slower, Cheaper per byte
Abstraction Goal

• Reality: There is no one type of memory to rule them all!

• Abstraction: hide the complex/undesirable details of reality.

• Illusion: We have the speed of SRAM, with the capacity of disk, at reasonable cost.
Motivating Story / Analogy

• You work at a video rental store (remember Blockbuster?)

• You have a huge warehouse of movies
  – 10-15 minutes to find movie, bring to customer
  – Customers don’t like waiting...

• You have a small office in the front with shelves, you choose what goes on shelves
  – < 30 seconds to find movie on front shelf
Goal: strategically put movies on office shelf to reduce trips to warehouse.

~30 seconds to find movie

~10 minutes to find movie
Quick vote: Which movie should we place on the shelf for tonight?

A. Eternal Sunshine of the Spotless Mind
B. The Godfather
C. Pulp Fiction
D. Rocky V
E. There’s no way for us to know.
Problem: Prediction

• We can’t know the future...

• So... are we out of luck?
  What might we look at to help us decide?

• The past is often a pretty good predictor...
Repeat Customer: Bob

• Has rented “Eternal Sunshine of the Spotless Mind” ten times in the last two weeks.

• You talk to him:
  – He just broke up with his girlfriend
  – Swears it will be the last time he rents the movie (he’s said this the last six times)
Quick vote: Which movie should we place on the shelf for tonight?

A. Eternal Sunshine of the Spotless Mind
B. The Godfather
C. Pulp Fiction
D. Rocky V
E. There’s no way for us to know.
Repeat Customer: Alice

• Alice rented Rocky a month ago

• You talk to her:
  – She’s really likes Sylvester Stalone

• Over the next few weeks she rented:
  – Rocky II, Rocky III, Rocky IV
Quick vote: Which movie should we place on the shelf for tonight?

A. Eternal Sunshine of the Spotless Mind
B. The Godfather
C. Pulp Fiction
D. Rocky V
E. There’s no way for us to know.