CS 61C: Great Ideas in Computer Architecture: *Malloc Examples, Introduction to Machine Language*

Instructor:
Michael Greenbaum

http://inst.eecs.Berkeley.edu/~cs61c/su11

Agenda

- Malloc Review and Examples
- Administrivia
- Machine Languages
- Break
- Data Transfer Instructions
- Instructions for Decisions
- Summary

When to Use Malloc

- malloc returns a pointer to *dynamically sized, persistent* memory.

```
int* foo() {
    int ret[10];
    return ret;
}
```

```
int* bar(int numInts) {
    int *ret = malloc(numInts *
                      sizeof(int));
    return ret;
}
```

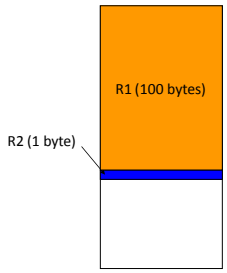
- `foo` returns a pointer to memory that is no longer valid after the function returns!
  - Size of allocated array must be a constant (except in C99)
- `bar` returns a pointer to memory that remains allocated until free is called on that address.

Malloc Internals

- Many calls to malloc and free with many different size blocks. Where to place them?
- Want system to be fast, minimal memory overhead.
- Want to avoid *Fragmentation*, the tendency of free space on the heap to get separated into small chunks.
- Some clever algorithms to do this, we will not cover them here.

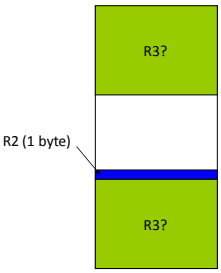
Fragmentation

- An example
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed
  - Request R3 for 50 bytes
  - What if R3 was a request for 101 bytes?



Fragmentation

- **An example**
  - **Request R1 for 100 bytes**
  - **Request R2 for 1 byte**
  - **Memory from R1 is freed**
  - **Request R3 for 50 bytes**
  - **What if R3 was a request for 101 bytes?**



Segmentation Fault vs. Bus Error

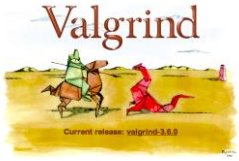
- http://www.hyperdictionary.com/
- Bus Error
  - A fatal failure in the execution of a machine language instruction resulting from the processor detecting an anomalous condition on its bus. Such conditions include invalid address alignment (accessing a multi-byte number at an odd address), accessing a physical address that does not correspond to any device, or some other device-specific hardware error. A bus error triggers a processor-level exception which Unix translates into a "SIGBUS" signal which, if not caught, will terminate the current process.
- Segmentation Fault
  - An error in which a running Unix program attempts to access memory not allocated to it and terminates with a segmentation violation error and usually a core dump.

Common Memory Problems

- Using uninitialized values
- Using memory that you don't own
  - Deallocated stack or heap variable
  - Out of bounds reference to stack or heap array
  - Using NULL or garbage data as a pointer
- Improper use of free/realloc by messing with the pointer handle returned by malloc/calloc
- Memory leaks (you allocated something you forgot to later free)

Debugging Tools

- Runtime analysis tools for finding memory errors
  - Dynamic analysis tool: collects information on memory management while program runs
  - Contrast with static analysis tool like `lint`, which analyzes source code without compiling or executing it
  - No tool is guaranteed to find ALL memory bugs – this is a very challenging programming language research problem
- *You will be introduced to Valgrind in Lab #3!*



http://valgrind.org

Using Uninitialized Values

- What is wrong with this code?

```
void foo(int *pi) {
    int j;
    *pi = j;
}

void bar() {
    int i=10;
    foo(&i);
    printf("i = %d\n", i);
}
```


Using Uninitialized Values

- What is wrong with this code?

```
void foo(int *pi) {
    int j;
    *pi = j;
    /* j is uninitialized, copied into *pi */
}

void bar() {
    int i=10;
    foo(&i);
    printf("i = %d\n", i);
    /* Using i, which now has junk value */
 }
```


Valgrind Output (Highly Abridged!)

```
==98863== Memcheck, a memory error detector
==98863== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info

==98863== Conditional jump or move depends on uninitialised value(s)
==98863==    at 0x100031A1E: __vfprintf (in /usr/lib/libSystem.B.dylib)
==98863==    by 0x100073BCA: vfprintf_l (in /usr/lib/libSystem.B.dylib)
==98863==    by 0x1000A11A6: printf (in /usr/lib/libSystem.B.dylib)
==98863==    by 0x100000EEE: main (slide21.c:13)
==98863==  Uninitialised value was created by a stack allocation
==98863==    at 0x100000EB0: foo (slide21.c:3)
==98863==
```

Valgrind Output (Highly Abridged!)

```
==98863== HEAP SUMMARY:
==98863==    in use at exit: 4,184 bytes in 2 blocks
==98863==   total heap usage: 2 allocs, 0 frees, 4,184 bytes allocated
==98863==
==98863== LEAK SUMMARY:
==98863==    definitely lost: 0 bytes in 0 blocks
==98863==    indirectly lost: 0 bytes in 0 blocks
==98863==      possibly lost: 0 bytes in 0 blocks
==98863==    still reachable: 4,184 bytes in 2 blocks
==98863==         suppressed: 0 bytes in 0 blocks
==98863== Reachable blocks (those to which a pointer was found) are not shown.
==98863== To see them, rerun with: --leak-check=full --show-reachable=yes
```


Using Memory You Don't Own

- What is wrong with this code?

```
typedef struct node {
   struct node* next;
   int val;
} Node;

int findLastNodeValue(Node* head) {
   while (head->next != NULL) {
      head = head->next;
   }
   return head->val;
}
```


Using Memory You Don't Own

- Following a NULL pointer to mem addr 0!

```
typedef struct node {
   struct node* next;
   int val;
} Node;

int findLastNodeValue(Node* head) {
   while (head->next != NULL) {
   /* What if head happens to be NULL? */
      head = head->next;
   }
   return head->val;
}
```


Using Memory You Don't Own

- What is wrong with this code?

```
struct Profile {
   char *name;
   int age;
}

struct Profile person = malloc(sizeof(struct Profile));
char* name = getName();
person.name = malloc(sizeof(char)*strlen(name));
strcpy(person.name,name);
… //Do stuff (that isn't buggy)
free(person);
free(person.name);
```


Using Memory You Don't Own

- What is wrong with this code?

```
struct Profile {
   char *name;
   int age;
}

struct Profile person = malloc(sizeof(struct Profile));
char* name = getName();
person.name = malloc(sizeof(char)*(strlen(name)+1));
strcpy(person.name,name);
… //Do stuff (that isn't buggy)
free(person);
free(person.name);
```

strlen only counts # of characters in a string. Need to add 1 for null terminator!

Accessing memory after you've already freed it. These statements should be in the reverse order

Using Memory You Don't Own

- What's wrong with this code?

```
char *append(const char* s1, const char *s2) {
   const int MAXSIZE = 128;
   char result[128];
   int i=0, j=0;
   for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
    result[i] = s1[j];
   }
   for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
    result[i] = s2[j];
   }
   result[++i] = '\0';
   return result;
}
```

Using Memory You Don't Own

- Beyond stack read/write

```
char *append(const char* s1, const char *s2) {
  const int MAXSIZE = 128;
  char result[128];
  int i=0, j=0;
  for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
   result[i] = s1[j];
  }
  for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
   result[i] = s2[j];
  }
  result[++i] = '\0';
  return result;
}
```

`result` is a local array name – stack memory allocated

Function returns pointer to stack memory – won't be valid after function returns

Using Memory You Haven't Allocated

- What is wrong with this code?

```
void StringManipulate() {
  const char *name = "Safety Critical";
  char *str = malloc(10);
  strncpy(str, name, 10);
  str[10] = '\0';
  printf("%s\n", str);
}
```


Using Memory You Haven't Allocated

- Reference beyond array bounds

```
void StringManipulate() {
  const char *name = "Safety Critical";
  char *str = malloc(10);
  strncpy(str, name, 10);
  str[10] = '\0';
  /* Write Beyond Array Bounds */
  printf("%s\n", str);
  /* Read Beyond Array Bounds */
}
```


Faulty Heap Management

- What is wrong with this code?

```
int *p;
void foo() {
  p = malloc(8*sizeof(int));
  …
  free(p);
}

void main() {
  p = malloc(4*sizeof(int));
  foo();
}
```


Faulty Heap Management

- Memory leak

```
int *p;
void foo() {
  p = malloc(8*sizeof(int));
  /* Oops, lost the address of the memory p previously
  pointed to */
  …
  free(p);
}

void main() {
  p = malloc(4*sizeof(int));
  foo();

}
```


Faulty Heap Management

- What is wrong with this code?

```
 int *plk = NULL;
 void genPLK() {
  plk = malloc(2 * sizeof(int));
  … … …
  plk++;
}
```

Faulty Heap Management

- Potential memory leak – handle has been changed, do you still have copy of it that can correctly be used in a later free?

```
int *plk = NULL;
void genPLK() {
  plk = malloc(2 * sizeof(int));
  … … …
  plk++; /* Potential leak: pointer variable
  incremented past beginning of block! */
}
```


Faulty Heap Management

- What is wrong with this code?

```
void FreeMemX() {
  int fnh = 0;
  free(&fnh);
}

void FreeMemY() {
  int *fum = malloc(4 * sizeof(int));
  free(fum+1);
  free(fum);
  free(fum);
}
```


Faulty Heap Management

- Can't free non-heap memory; Can't free memory that hasn't been allocated

```
void FreeMemX() {
  int fnh = 0;
  free(&fnh); /* Oops! freeing stack memory */
}

void FreeMemY() {
  int *fum = malloc(4 * sizeof(int));
  free(fum+1);
  /* fum+1 is not a proper handle; points to middle
  of a block */
  free(fum);
  free(fum);
  /* Oops! Attempt to free already freed memory */
}
```


Agenda

- Malloc Review and Examples
- Administrivia
- Machine Languages
- Break
- Data Transfer Instructions
- Instructions for Decisions
- Summary

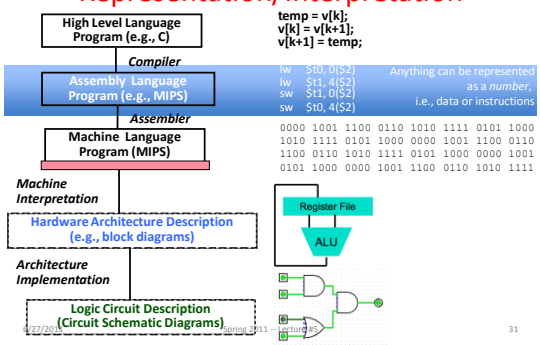
Administrivia

- HW2 posted...
  - WARNING, this homework is VERY LARGE. It is actually two weeks of homework compacted into one week. We've decided to give you a single assignment over a week rather than two assignments due twice this week. More flexibility, but more responsibility. Start today.
  - Problem 1 from HW2 is larger than ALL OF HW1.
- Lab 3 posted, Project 1 posted mid-week, due a week from this Sunday.

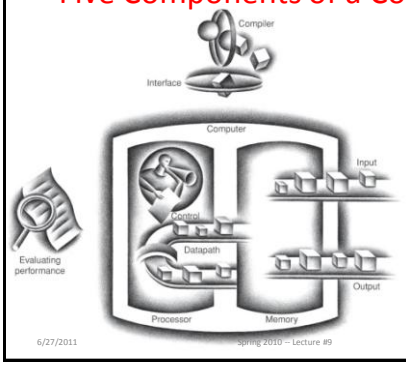
Agenda

- Malloc Review and Examples
- Administrivia
- Machine Languages
- Break
- Data Transfer Instructions
- Instructions for Decisions
- Summary

Levels of Representation/Interpretation



Five Components of a Computer



- Control
- Datapath
- Memory
- Input
- Output

The Language a Computer Understands

- Word a computer understands: *instruction*
- Vocabulary of all words a computer understands: *instruction set* (aka *instruction set architecture* or *ISA*)
- Different computers may have *different* vocabularies (i.e., different ISAs)
  - iPhone not same as Macbook
- Or the *same* vocabulary (i.e., same ISA)
  - iPhone and iPad computers have same instruction set

The Language a Computer Understands

- Why not all the same? Why not all different? What might be pros and cons?

The Language a Computer Understands

- Why not all the same? Why not all different? What might be pros and cons?
  - Single ISA (*to rule them all*):
    - Leverage common compilers, operating systems, etc.
    - BUT fairly easy to retarget these for different ISAs (e.g., Linux, gcc)
  - Multiple ISAs:
    - Specialized instructions for specialized applications
    - Different tradeoffs in resources used (e.g., functionality, memory demands, complexity, power consumption, etc.)
    - Competition and innovation is good, especially in emerging environments (e.g., mobile devices)

Instruction Set in CS61c

- MIPS
  - Invented by John Hennessy @ Stanford
  - MIPS is a real world ISA
    - Standard instruction set for networking equipment
- Elegant example of *Reduced Instruction Set Computer* (RISC) instruction set

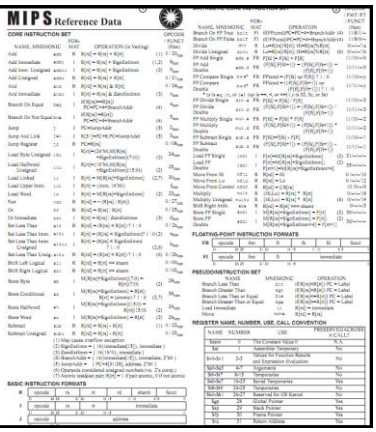
RISC Design Principles

- Basic RISC principle: "A simpler CPU (the hardware that interprets machine language) is a faster CPU" (CPU → *Core*)
- Focus of the RISC design is reduction of the number and complexity of instructions in the ISA
- A number of the more common strategies include:
  - Fixed instruction length, generally a single word; Simplifies process of fetching instructions from memory
  - Simplified addressing modes; Simplifies process of fetching operands from memory
  - Fewer and simpler instructions in the instruction set; Simplifies process of executing instructions
  - Only load and store instructions access memory; E.g., no add memory to register, add memory to memory, etc.
  - *Let the compiler do it.* Use a good compiler to break complex high-level language statements into a number of simple assembly language statements

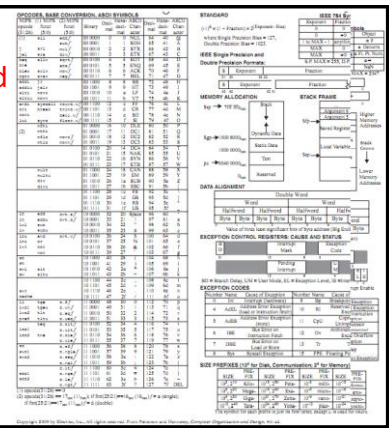
Other Instruction Sets

- Most popular RISC is ARM (Advanced RISC Machine)
  - In every smart phone-like device |(e.g., iPhone, iPad, iPod, …)
- Another popular Instruction Set is used in Macbook and PCs: Intel 80x86 in Core i3, Core i5, Core i7, …
  - 20x ARM versus 80x86 (i.e., 05 billion vs. .3 billion)

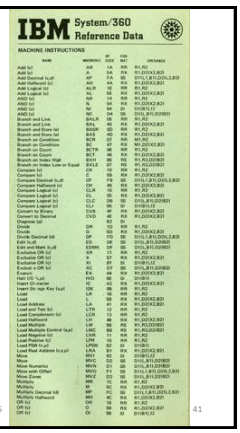
MIPS Green Card



MIPS Green Card



Inspired by the IBM 360 "Green Card"



MIPS Instructions

- Every computer does arithmetic
- *Instruct* a computer to do addition:

```
add a, b, c
```

  - Put into `a` the sum of `b` and `c`.
- 3 operands: 2 sources + 1 destination for sum
- One operation per MIPS instruction
- How do you write the same operation in C?

Guess More MIPS instructions

- Subtract `c` from `b` and put difference in `a`?

```
sub a, b, c
```

- Multiply `b` by `c` and put product in `a`?

```
mul a, b, c
```

- Divide `b` by `c` and put quotient in `a`?

```
div a, b, c
```


Guess More MIPS instructions

- C operator &: `c & b` with result in `a`?

```
and a, b, c
```

- C operator |: `c | b` with result in `a`?

```
or a, b, c
```

- C operator <<: `c << b` with result in `a`?

```
sll a, b, c
```

- C operator >>: `c >> b` with result in `a`?

```
srl a, b, c
```


Example Instructions

- MIPS instructions are inflexible, rigid:
  - Just one arithmetic operation per instruction
  - Always with three operands
- How write this C expression in MIPS?

```
a = b + c + d + e
```


Example Instructions

- How write this C expression in MIPS?

```
a = b + c + d + e
add t1, d, e
add t2, c, t1
add a, b, t2
```


Comments in MIPS

- Can add comments to MIPS instruction by putting # that continues to end of line of text

```
add a, b, c # b + c is placed in a
add a, a, d # b + c + d is now in a
add a, a, e # b + c + d + e is in a
```


C to MIPS

- What is MIPS code that performs same as?

```
a = b + c; add a, b, c
d = a - e; sub d, a, e
```

- What is MIPS code that performs same as?

```
f = (g + h) - (i + j);
add t1, i, j
add t2, g, h
sub f, t2, t1
```

Agenda

- Malloc Review and Examples
- Administrivia
- Machine Languages
- Break
- Data Transfer Instructions
- Instructions for Decisions
- Summary

Computer Hardware Operands

- High-Level Programming languages: could have millions of variables
- Instruction sets have fixed, smaller number
- Called *registers*
  - "Bricks" of computer hardware
  - Used to construct computer hardware
  - Visible to compiler (or MIPS programmer).
- MIPS Instruction Set has 32 registers

Why Just 32 Registers?

- RISC Design Principle: *Smaller is faster*
  - But you can be too small …
- Hardware would likely be slower with 64, 128, or 256 registers
- 32 is enough for compiler to translate typical C programs, and not run out of registers very often
  - ARM instruction set has only 16 registers
  - May be faster, but compiler may run out of registers too often (aka "spilling registers to memory")

Names of MIPS Registers

- For registers that hold programmer variables: `$s0`, `$s1`, `$s2`, …
- For registers that hold temporary variables: `$t0`, `$t1`, `$t2`, …
- You'll learn about the others later.
- Suppose variables `f`, `g`, `h`, `i`, and `j` are assigned to the registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively. What is MIPS for

```
f = (g + h) - (i + j);
```


Names of MIPS Registers

- Suppose variables `f`, `g`, `h`, `i`, and `j` are assigned to the registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively. What is MIPS for

```
f = (g + h) - (i + j);
  add $t1, $s3, $s4
  add $t2, $s1, $s2
  sub $s0, $t2, $t1
```


Size of Registers

- *Bit* is the atom of Computer Hardware: contains either 0 or 1
  - True "alphabet" of computer hardware is 0, 1
- MIPS registers are 32 bits wide
- MIPS calls this quantity a *word*

Agenda

- Malloc Review and Examples
- Administrivia
- Machine Languages
- Break
- Data Transfer Instructions
- Instructions for Decisions
- Summary

Data Structures vs. Simple Variables

- C variables map onto registers; what about large data structures like arrays?
- Remember *memory*, our big array indexed by addresses.
- But MIPS instructions only operate on registers!
- Data transfer instructions transfer data between registers and memory:
  - Memory to register
  - Register to memory

Transfer from Memory to Register

- MIPS instruction: *Load Word*, abbreviated `lw`
- Load Word Syntax:

  lw 1,3(2)

  - where
    1) register that will receive value
    2) register containing pointer to memory
    3) numerical offset from 2) in bytes
- Adds 3) to address stored in 2), loads **FROM** this address in memory and puts result in 1).

Transfer from Memory to Register

Data flow

Example: `lw $t0,12($s0)`

This instruction will take the pointer in `$s0`, add `12` bytes to it, and then load the value from the memory pointed to by this calculated sum into register `$t0`

- Notes:
  - `$s0` is called the base register
  - `12` is called the offset
  - offset is generally used in accessing elements of array or structure: base reg points to beginning of array or structure (note offset must be a constant known at assembly time)

Transfer from Register to *Memory*

- MIPS instruction: *Store Word*, abbreviated `sw`
- Syntax similar to `lw`.

Data flow

- Example: `sw $t0,12($s0)`

  This instruction will take the pointer in `$s0`, add `12` bytes to it, and then store the value from register `$t0` into that memory address
- Remember: "Store **INTO** memory"

Memory Addresses are in Bytes

- Lots of data is smaller than 32 bits, but rarely smaller than 8 bits – works fine if everything is a multiple of 8 bits
- 8 bit item is called a *byte* (1 word = 4 bytes)
- Memory addresses are really in *bytes*, not words
- Word addresses are 4 bytes apart
  - Word address is same as leftmost byte

Addr of lowest byte in word is addr of word

...	...	...	...
12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

lw/sw Example

- Assume A is an array of 100 words, variables `g` and `h` map to registers `$s1` and `$s2`, the starting address, or base address, of the array A is in `$s3`

```
A[10] = h + A[3];
```

- Turns into

```
lw   $t0,12($s3)   # Temp reg $t0 gets A[3]
add  $t0,$s2,$t0   # t0 = h + A[3]
sw   $t0,40($s3)   # A[10] = h + A[3]
```


Speed of Registers vs. Memory

- Given that
  - Registers: 32 words (128 Bytes)
  - Memory: Billions of bytes (2 GB to 8 GB on laptop)
- and the RISC principle is
  - Smaller is faster
- How much faster are registers than memory??
- About 100-500 times faster!

Agenda

- Malloc Review and Examples
- Administrivia
- Machine Languages
- Break
- Data Transfer Instructions
- Instructions for Decisions
- Summary

Computer Decision Making

- Based on computation, do something different
- In programming languages: if-statement
- Sometimes combined with gotos and labels
- MIPS: conditional instruction is

```
beq reg1,reg2,L1
```

  means go to statement labeled L1 if value in reg1 == value in reg2
- `beq` stands for *branch if equal*
- Other instruction: `bne` for *branch if not equal*

Making Decisions in C or Java

```
if (i == j) f = g + h; else f = g – h;
```

- If false, skip over "then" part to "else" part => use conditional branch `bne`
- Otherwise, (its true) do "then" part and skip over "else" part => need an always branch instruction ("*unconditional branch*")
- MIPS name for this instruction: *jump (`j`)*

Making Decisions in MIPS

```
if (i == j) f = g + h; else f = g – h;
```

- `f` => `$s0`, `g` => `$s1`, `h` => `$s2`, `i` => `$s3`, `j` => `$s4`
- If false, skip "then" part to "else" part
- Otherwise, (its true) do "then" part and skip over "else" part

```
       bne $s3,$s4,Else    # go to Else part if i ≠ j
       add $s0,$s1,$s2     # f = g + h (Then part)
       j Exit              # go to Exit
Else:  sub $s0,$s1,$s      # f = g – h (Else part)
Exit:
```

And In Conclusion …

- Computer words and vocabulary are called *instructions* and *instruction set* respectively
- MIPS is example RISC instruction set in this class
- Rigid format: 1 operation, 2 source operands, 1 destination
  - `add`, `sub`, `mul`, `div`, `and`, `or`, `sll`, `srl`
  - `lw`, `sw` to move data to/from registers from/to memory
- Simple mappings from arithmetic expressions, array access, `if-then-else` in C to MIPS instructions

Bonus slides

- These are extra slides that used to be included in lecture notes, but have been moved to this, the "bonus" area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

Bonus

Using Memory You Don't Own

- What is wrong with this code?

```
int* init_array(int *ptr, int new_size) {
    ptr = realloc(ptr, new_size*sizeof(int)); //Returns new block of memory
    memset(ptr, 0, new_size*sizeof(int));
    return ptr;
}

int* fill_fibonacci(int *fib, int size) {
    int i;
    init_array(fib, size);
    /* fib[0] = 0; */ fib[1] = 1;
    for (i=2; i<size; i++)
      fib[i] = fib[i-1] + fib[i-2];
    return fib;
}
```


Using Memory You Don't Own

```
int* init_array(int *ptr, int new_size) {
    ptr = realloc(ptr, new_size*sizeof(int));
    memset(ptr, 0, new_size*sizeof(int));
    return ptr;
}

int* fill_fibonacci(int *fib, int size) {
    int i;
    /* oops, forgot: fib = */ init_array(fib, size);
    /* fib[0] = 0; */ fib[1] = 1;
    for (i=2; i<size; i++)
      fib[i] = fib[i-1] + fib[i-2];
    return fib;
}
```

Remember: `realloc` may move entire block

What if array is moved to new location?