CSE 31
Computer Organization

Lecture 7 – C Memory Management (3)
MIPS: Arithmetic

Announcement

- Lab #3 this week
  - Due in one week
- HW #2 (at CatCourses)
  - Written homework, NOT from zyBooks
  - Type your answers or scan and submit trough CatCourses
  - Due Monday (9/24)
- Midterm exam Wednesday (9/26)
  - Lectures 1 – 7
  - HW #1 and #2
  - Closed book
  - 1 sheet of note (8.5” x 11”)
  - Sample exam online
- Reading assignment
  - Chapter 2.1 – 2.9 of zyBooks (Reading Assignment #2)
    - Make sure to do the Participation Activities
    - Due Wednesday (9/19)

Automatic Memory Management

- Dynamically allocated memory is difficult to track
  - Why not track it automatically?
- If we can keep track of what memory is in use, we can reclaim everything else.
  - Unreachable memory is called garbage, the process of reclaiming it is called garbage collection.
- So how do we track what is in use?

Tracking Memory Usage

- Techniques depend heavily on the programming language and rely on help from the compiler.
- Start with all pointers in global variables and local variables (root set).
- Recursively examine dynamically allocated objects we see a pointer to.
  - We can do this in constant space by reversing the pointers on the way down
- How do we recursively find pointers in dynamically allocated memory?

Tracking Memory Usage

- Again, it depends heavily on the programming language and compiler.
- Could have only a single type of dynamically allocated object in memory
  - E.g., simple Lisp/Scheme system with only `cons` cells
- Could use a strongly typed language (e.g., Java)
  - Don't allow conversion (casting) between arbitrary types.
  - C/C++ are not strongly typed.
- Here are 3 schemes to collect garbage

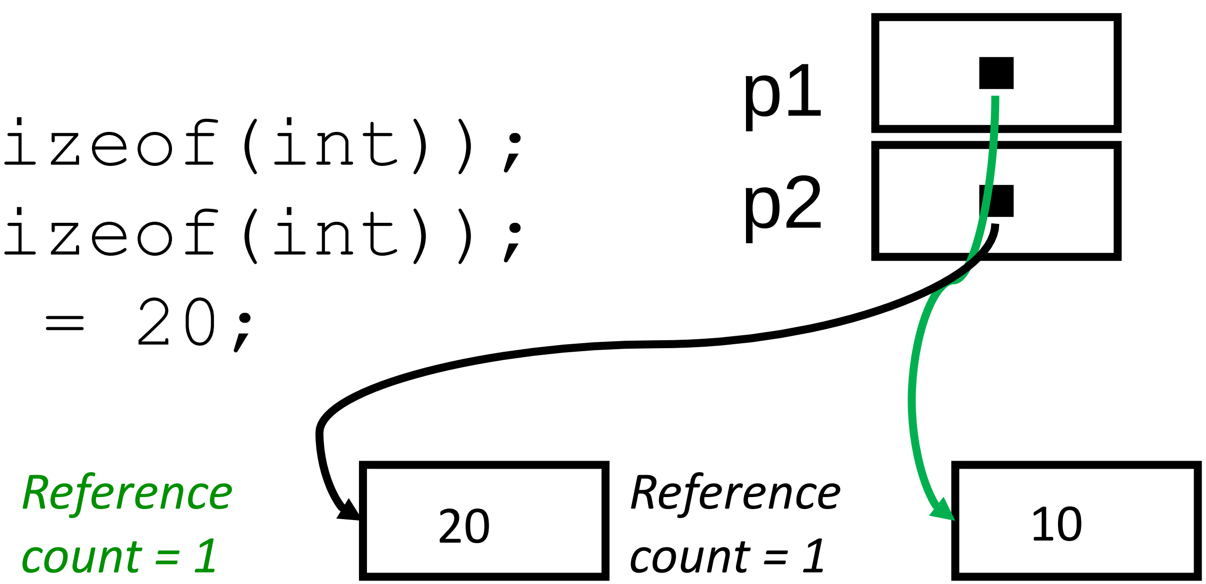
Scheme 1: Reference Counting

- For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
- When the count reaches 0, reclaim the memory.
- Simple assignment statements can result in a lot of work, since may update reference counts of many items

Reference Counting Example

- For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
  - When the count reaches 0, reclaim.

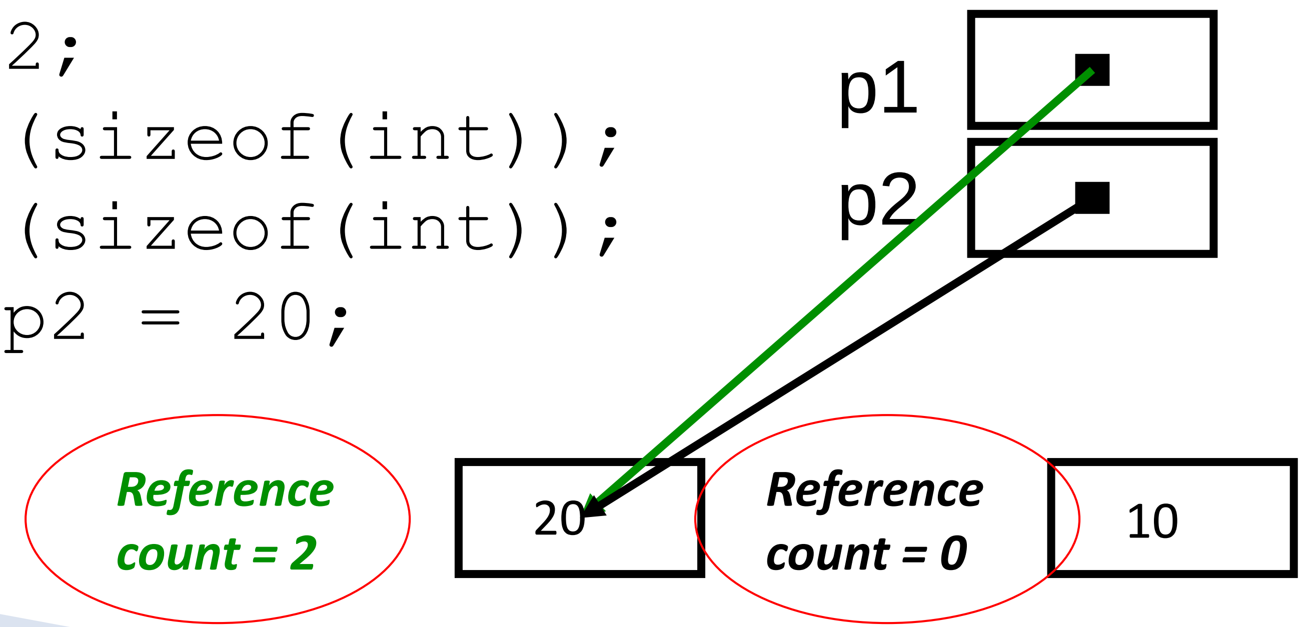
```
int *p1, *p2;
p1 = malloc(sizeof(int));
p2 = malloc(sizeof(int));
*p1 = 10; *p2 = 20;
```



Reference Counting Example

- For every chunk of dynamically allocated memory, keep a count of number of pointers that point to it.
  - When the count reaches 0, reclaim.

```
int *p1, *p2;
p1 = malloc(sizeof(int));
p2 = malloc(sizeof(int));
*p1 = 10; *p2 = 20;
p1 = p2;
```



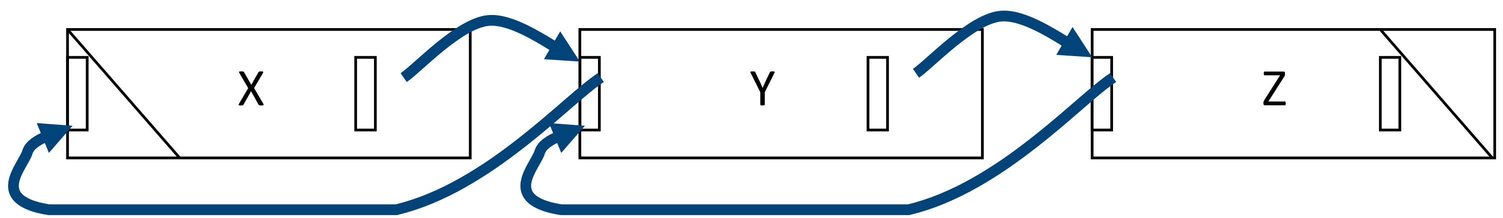
Reference Counting (p1, p2 are pointers)

```
p1 = p2;
```

- Increment reference count for `p2`
- If `p1` held a valid value, decrement its reference count
- If the reference count for `p1` is now 0, reclaim the storage it points to.
  - If the storage pointed to by `p1` held other pointers, decrement all of their reference counts, and so on…
- Must also decrement reference count when local variables cease to exist.

Reference Counting Flaws

- Extra overhead added to assignments, as well as ending a block of code.
- Does not work for circular structures!
  - E.g., doubly linked list:



Scheme 2: Mark and Sweep Garbage Collection

- Keep allocating new memory until memory is exhausted, then try to find unused memory.
- Consider objects in a graph, chunks of memory (objects) are graph nodes, pointers to memory are graph edges.
  - Edge from A to B $\rightarrow$ A stores pointer to B
- Can start with the root set, perform a graph traversal, find all usable memory!
- 2 Phases:
  1. Mark used nodes
  2. Sweep free ones, returning list of free nodes

Mark and Sweep

- Graph traversal is relatively easy to implement recursively

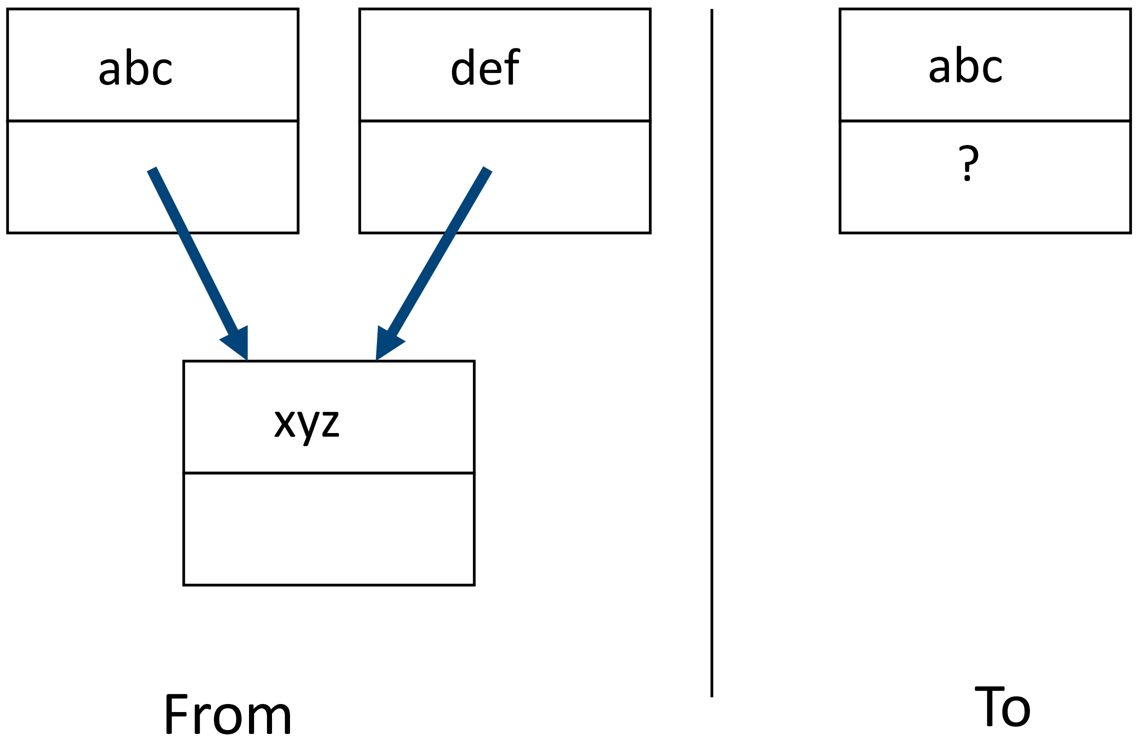
```
void traverse(struct graph_node *node) {
    /* visit this node */
    foreach child in node->children {
        traverse(child);
    }
}
```

- But with recursion, state is stored on the execution stack.
  - Garbage collection is invoked when not much memory left
- As before, we could traverse in constant space (by reversing pointers)

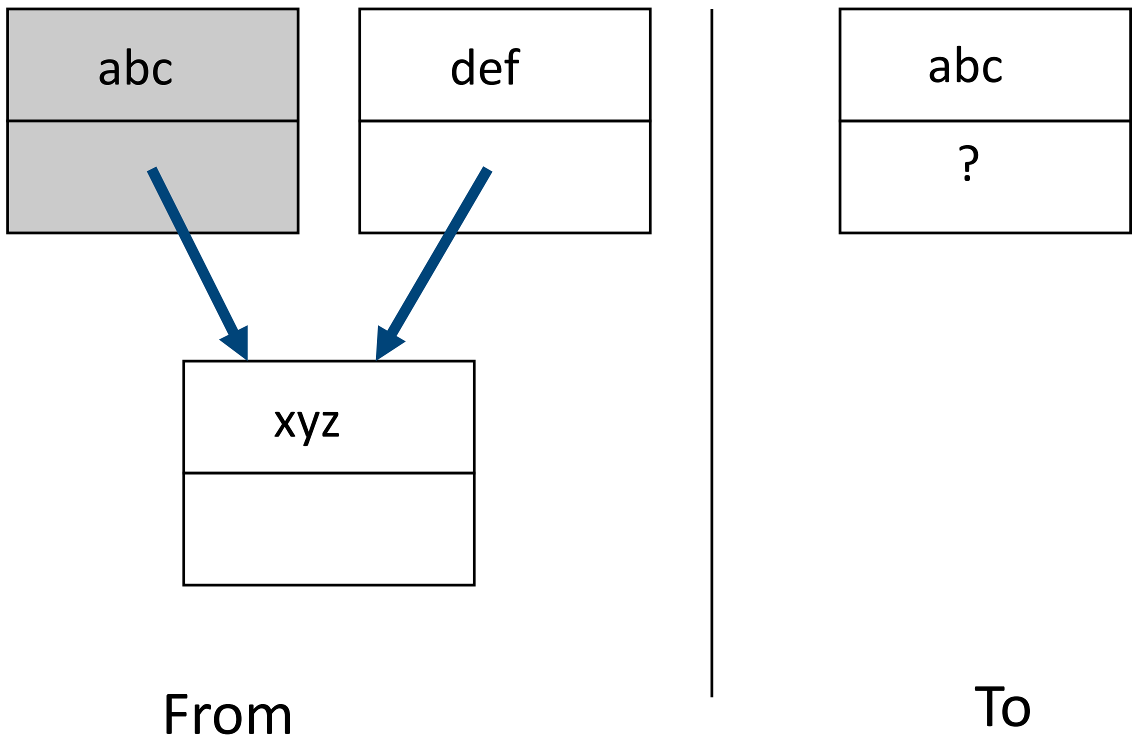
Scheme 3: Copying Garbage Collection

- Divide memory into two spaces, only one in use at any time.
- When active space is exhausted, traverse the active space, copying all objects to the other space, then make the new space active and continue.
  - Only reachable objects are copied!
- Use “forwarding pointers” to keep consistency
  - Simple solution to avoiding having to have a table of old and new addresses, and to mark objects already copied

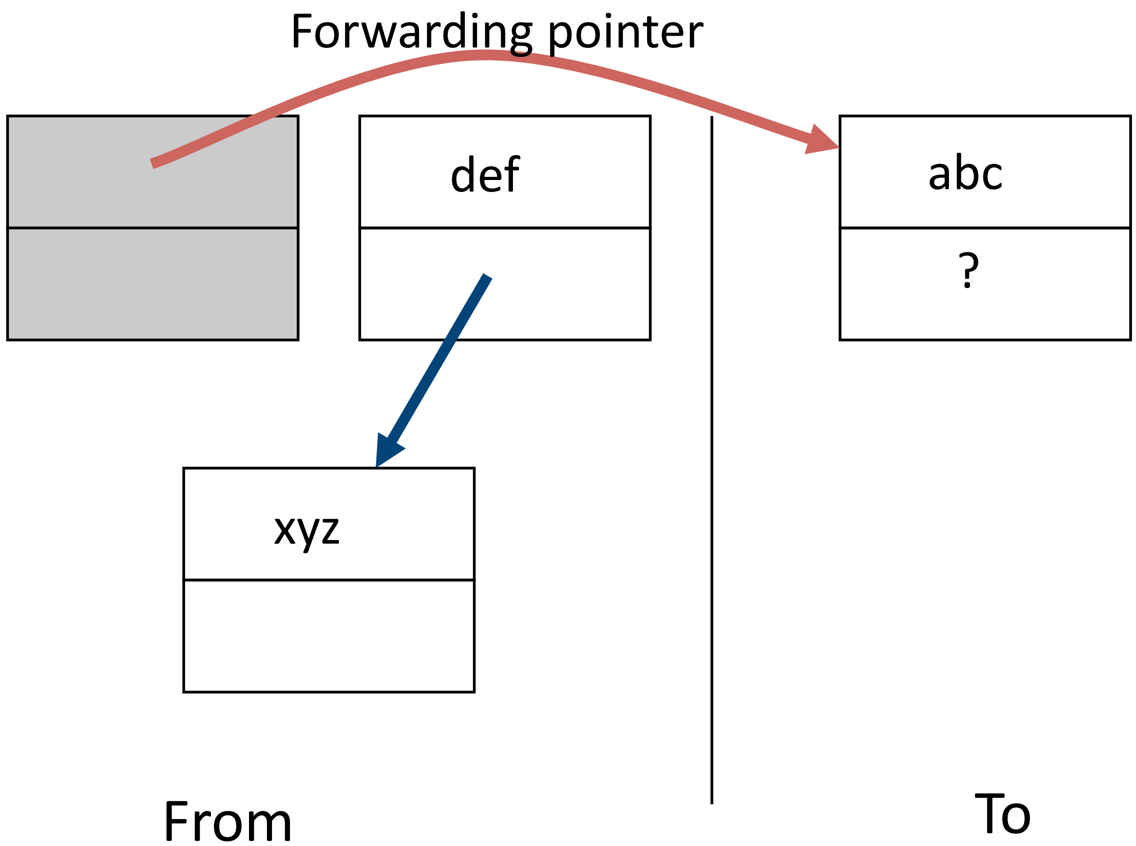
Forwarding Pointers: 1st copy "abc"



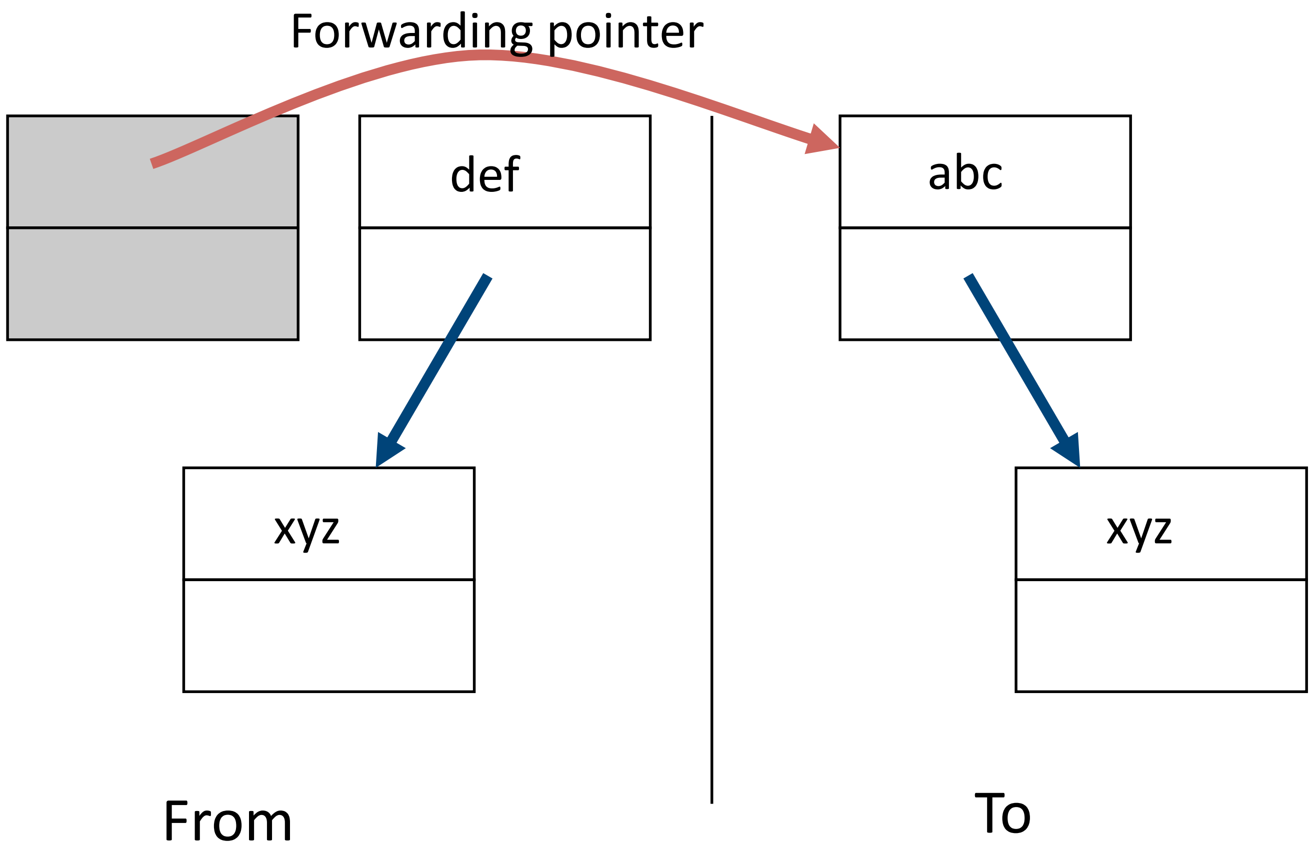
Forwarding Pointers: leave ptr to new abc



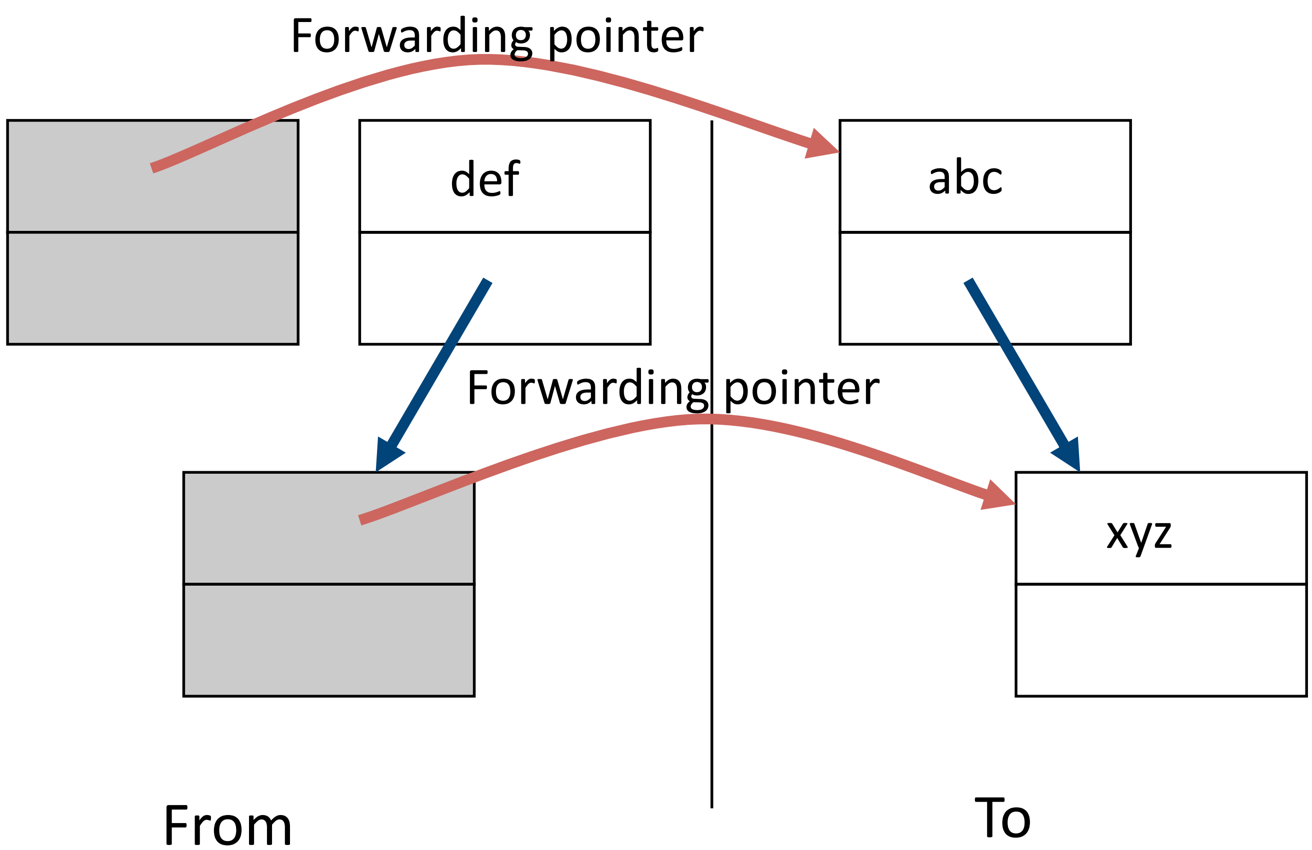
Forwarding Pointers : now copy “xyz”



Forwarding Pointers: leave ptr to new xyz

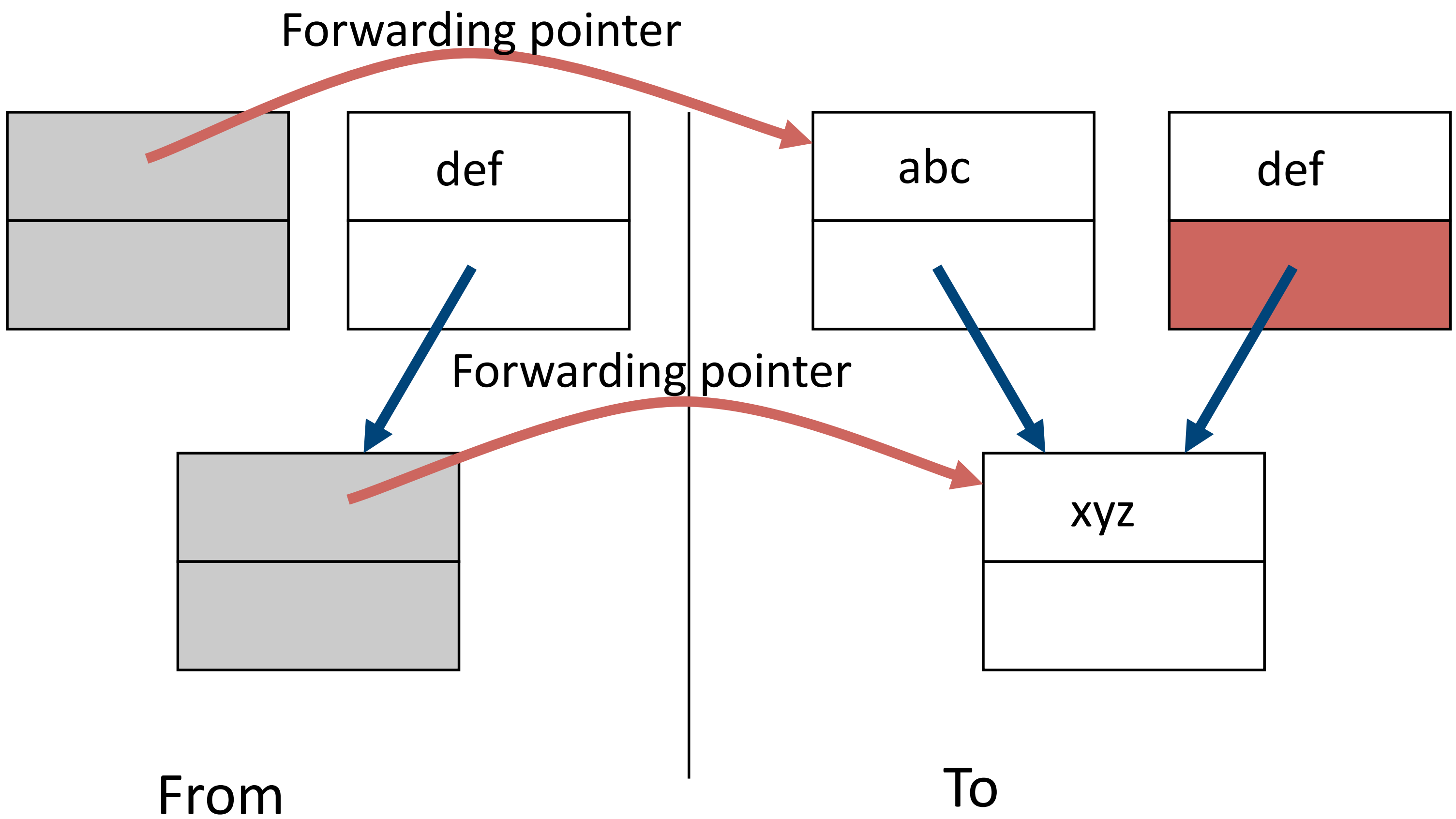


Forwarding Pointers: now copy "def"



Since xyz was already copied,
def uses xyz's forwarding pointer
to find its new location

Forwarding Pointers



Since xyz was already copied,
def uses xyz's forwarding pointer
to find its new location

Summary

- Several techniques for managing heap via malloc and free: best-, first-, next-fit
  - 2 types of memory fragmentation: internal & external; all suffer from some kind of frag.
  - Each technique has strengths and weaknesses, none is definitively best
- Automatic memory management relieves programmer from managing memory.
  - All require help from language and compiler
  - Reference Count: not for circular structures
  - Mark and Sweep: complicated and slow, works
  - Copying: Divides memory to copy good stuff

Assembly Language

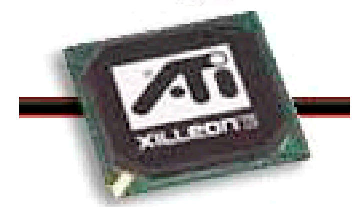
- Basic job of a CPU: execute lots of instructions.
- Instructions are the primitive operations that the CPU may execute.
- Different CPUs implement different sets of instructions. The set of instructions a particular CPU implements is an Instruction Set Architecture (ISA).
  - Examples: Intel 80x86 (Pentium 4), IBM/Motorola PowerPC (Macintosh), MIPS (**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages), Intel IA64, ...

Instruction Set Architectures

- Early trend was to add more and more instructions to new CPUs to do elaborate operations
  - VAX architecture had an instruction to multiply polynomials!
- RISC philosophy (Cocke IBM, Patterson, Hennessy, 1980s) – **R**educed **I**nstruction **S**et **C**omputing
  - Keep the instruction set small and simple, makes it easier to build fast hardware.
  - Let software do complicated operations by composing simpler ones.

MIPS Architecture

- MIPS – semiconductor company that built one of the first commercial RISC architectures
- We will study the MIPS architecture in some detail in this class (also used in upper division courses)
- Why MIPS instead of Intel 80x86?
  - MIPS is simple, elegant. Don't want to get bogged down in gritty details.
  - MIPS widely used in embedded apps, x86 little used in embedded, and more embedded computers than PCs



Most HP LaserJet workgroup printers are driven by MIPS-based™ 64-bit processors.

Assembly Variables: Registers (1/4)

- Unlike HLL like C or Java, assembly cannot use variables
  - Why not?
    - Keep Hardware Simple
- Assembly operands are registers
  - Limited number of special locations built directly into the hardware
  - Operations can only be performed on these!
- Benefit: Since registers are directly in hardware, they are very fast (faster than 1 billionth of a second)

Assembly Variables: Registers (2/4)

- Drawback: Since registers are in hardware, there are a predetermined number of them
  - Solution: MIPS code must be very carefully put together to efficiently use registers
- 32 registers in MIPS
  - Why 32?
    - Smaller is faster
- Each MIPS register is 32 bits wide
  - Groups of 32 bits called a word in MIPS
  - Basic unit of data storage

Assembly Variables: Registers (3/4)

- Registers are numbered from 0 to 31
- Each register can be referred to by number or name
- Number references:

  `$0, $1, $2, … $30, $31`

Assembly Variables: Registers (4/4)

- **By convention**, each register also has a name to make it easier to code
- For now:

  `$16 - $23` ➔ `$s0 - $s7`

  (correspond to C variables)

  `$8 - $15` ➔ `$t0 - $t7`

  (correspond to temporary variables)

  Later will explain other 16 register names
- In general, use names to make your code more readable

C, Java variables vs. registers

- In C (and most High Level Languages) variables declared first and given a type
  - Example:

```
int fahr, celsius;
char a, b, c, d, e;
```

- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match `int` and `char` variables).
- In Assembly Language, the registers have no type; **operation** determines how register contents are treated

Comments in Assembly

- Another way to make your code more readable!
- Hash (#) is used for MIPS comments
  - anything from hash mark to end of line is a comment and will be ignored
  - This is just like the C99 //
- Note: Different from C.
  - C comments have format
    `/* comment: Cannot use this with MIPS! */`
    so they can span many lines

Assembly Instructions

- In assembly language, each statement (called an Instruction), executes exactly one of a short list of simple commands
- Unlike in C (and most other High Level Languages), each line of assembly code contains at most 1 instruction
- Instructions are related to operations (=, +, -, *, /) in C or Java
- Ok, ready for MIPS?

MIPS Addition and Subtraction (1/4)

- Syntax of Instructions:

  Format: 1,2,3,4

  where:

  1) operation by name

  2) operand getting result ("destination")

  3) 1st operand for operation ("source1")

  4) 2nd operand for operation ("source2")

- Syntax is rigid:
  - 1 operator, 3 operands
  - Why?
    - Keep Hardware simple via regularity

Addition and Subtraction of Integers (1/3)

- Addition in Assembly
  - Example: `add $s0,$s1,$s2` (in MIPS)
    Equivalent to: `a = b + c` (in C)
  where MIPS registers `$s0,$s1,$s2` are associated with C variables `a, b, c`
- Subtraction in Assembly
  - Example: `sub $s3,$s4,$s5` (in MIPS)
    Equivalent to: `d = e - f` (in C)
  where MIPS registers `$s3,$s4,$s5` are associated with C variables `d, e, f`

Addition and Subtraction of Integers (2/3)

- How do the following C statement work in MIPS?

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

- Break into multiple instructions

```
add $t0, $s1, $s2 # temp = b + c
add $t0, $t0, $s3 # temp = temp + d
sub $s0, $t0, $s4 # a = temp - e
```

  - Notice: A single line of C may break up into several lines of MIPS.
  - Notice: Everything after the hash mark on each line is ignored (comments)

Addition and Subtraction of Integers (3/3)

- How do we do this?

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

- Use intermediate temporary register

```
add $t0,$s1,$s2    # temp = g + h
add $t1,$s3,$s4    # temp = i + j
sub $s0,$t0,$t1    # f=(g+h)-(i+j)
```

Immediates

- **Immediates** are numerical constants.
- They appear often in code, so there are special instructions for them.
- Add Immediate:

  `addi $s0,$s1,10` (in MIPS)

  `f = g + 10` (in C)

  where MIPS registers `$s0,$s1` are associated with C variables `f, g`
- Syntax similar to `add` instruction, except that last operand is a number instead of a register.

Register Zero

- One particular immediate:
  - The number zero (0), appears very often in code.
- So we define register zero (`$0` or `$zero`) to always have the value 0

  `add $s0,$s1,$zero` (in MIPS)

  `f = g` (in C)

  where MIPS registers `$s0,$s1` are associated with C variables `f, g`
- defined in hardware, so an instruction

  `add $zero,$zero,$s0`

  will not do anything!

Immediates

- There is no Subtract Immediate in MIPS: Why?
- Limit types of operations that can be done to absolute minimum
  - if an operation can be decomposed into a simpler operation, don't include it
  - `addi` …, `-X` = `subi` …, `X` => so no `subi`
- `addi $s0,$s1,-10` (in MIPS)

  `f = g - 10` (in C)

  where MIPS registers `$s0,$s1` are associated with C variables `f, g`

Quiz

1) Since there are only 8 local (`$s`) and 8 temp (`$t`) variables, we can't write MIPS for C exprs that contain > 16 vars.

2) If `p` (stored in `$s0`) is a pointer to an array of `int`s, then `p++;` would be `addi $s0 $s0 1`

```
    12
a)  FF
b)  FT
c)  TF
d)  TT
e) dunno
```

Quiz

1) Since there are only 8 local (`$s`) and 8 temp (`$t`) variables, we can't write MIPS for C exprs that contain > 16 vars.

2) If `p` (stored in `$s0`) is a pointer to an array of `int`s, then `p++;` would be `addi $s0 $s0 1`

	12
a)	**FF**
b)	FT
c)	TF
d)	TT
e)	dunno

Summary

- In MIPS Assembly Language:
  - Registers replace variables
  - One Instruction (simple operation) per line
  - Simpler is Better, Smaller is Faster
- New Instructions:

  `add, addi, sub`
- New Registers:

  C Variables: `$s0 - $s7`

  Temporary Variables: `$t0 - $t7`

  Zero: `$zero`