

### Lecture #3 - Dynamic Storage

2009-06-29



### Review

- Pointers and arrays are virtually same
- C knows how to increment pointers
- C is an efficient language, with little protection
  - Array bounds not checked
  - Variables not automatically initialized
- (Beware) The cost of efficiency is more overhead for the programmer.
  - "C gives you a lot of extra rope but be careful not to hang yourself with it!"



### Dynamic Memory Allocation (1/4)

- C has operator `sizeof()` which gives size in bytes (of type or variable)
- Assume size of objects can be misleading and is bad style, so use `sizeof(type)`
  - Many years ago an int was 16 bits, and programs were written with this assumption.
  - What is the size of integers now?
- "sizeof" knows the size of arrays:
 

```
int ar[3]; // Or: int ar[] = {54, 47, 99};
sizeof(ar) => 12
```

  - ...as well for arrays whose size is determined at run-time:
 

```
int n = 3;
int ar[n]; // Or: int ar[fun_that_returns_3()];
sizeof(ar) => 12
```



### Dynamic Memory Allocation (2/4)

- To allocate room for something new to point to, use `malloc()` (with the help of a typecast and `sizeof`):

```
ptr = (int *) malloc (sizeof(int));
```

- Now, `ptr` points to a space somewhere in memory of size `(sizeof(int))` in bytes.
- `(int *)` simply tells the compiler what will go into that space (called a typecast).

- `malloc` is almost never used for 1 var

```
ptr = (int *) malloc (n*sizeof(int));
```

- This allocates an array of `n` integers.



### Dynamic Memory Allocation (3/4)

- Once `malloc()` is called, the memory location contains garbage, so don't use it until you've set its value.

- After dynamically allocating space, we must dynamically free it:

```
free(ptr);
```

- Use this command to clean up.

- Even though the program frees all memory on `exit()` (or when `main` returns), don't be lazy!

- You never know when your `main` will get transformed into a subroutine!



### Dynamic Memory Allocation (4/4)

- The following two things will cause your program to crash or behave strangely later on, and cause VERY VERY hard to figure out bugs:

- `free()` ing the same piece of memory twice
- calling `free()` on something you didn't get back from `malloc()`

- The runtime does not check for these mistakes
  - Memory allocation is so performance-critical that there just isn't time to do this
  - The usual result is that you corrupt the memory allocator's internal structure
  - You won't find out until much later on, in a totally unrelated part of your code!

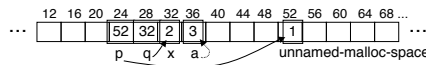


### Arrays not implemented as you'd think

```
void foo() {
  int *p, *q, x, a[1]; // a[1] = {3} also works here
  p = (int *) malloc (sizeof(int));
  q = &x;

  *p = 1; // p[0] would also work here
  *q = 2; // q[0] would also work here
  *a = 3; // a[0] would also work here

  printf("*p:%u, p:%u, &p:%u\n", *p, p, &p);
  printf("*q:%u, q:%u, &q:%u\n", *q, q, &q);
  printf("*a:%u, a:%u, &a:%u\n", *a, a, &a);
}
```



```
*p:1, p:52, &p:24
*q:2, q:32, &q:28
*a:3, a:36, &a:36
```



### Don't forget the globals!

- Remember:
  - Structure declaration does not allocate memory
  - Variable declaration does allocate memory
- So far we have talked about several different ways to allocate memory for data:
  1. Declaration of a local variable
 

```
int i; struct Node list; char *string; int ar[n];
```
  2. "Dynamic" allocation at runtime by calling allocation function (`alloc`).
 

```
ptr = (struct Node *) malloc(sizeof(struct Node)*n);
```
  3. Data declared outside of any procedure (i.e., before `main`).
- One more possibility exists...
  - Similar to #1 above, but has "global" scope.

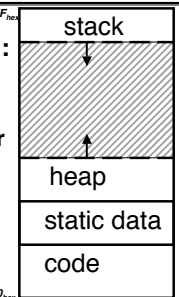
```
int myGlobal;
main() {
}
```



### C Memory Management

- A program's address space contains 4 regions:

- stack: local variables, grows downward
- heap: space requested for pointers via `malloc()`; resizes dynamically, grows upward
- static data: variables declared outside `main`, does not grow or shrink



- code: loaded when program starts, does not change

For now, OS somehow prevents accesses between stack and heap (gray hash lines). Wait for virtual memory



## Where are variables allocated?

- If declared **outside** a procedure, allocated in “static” storage
- If declared **inside** procedure, allocated on the “stack” and freed when procedure returns.
  - NB: main() is a procedure

```
int myGlobal;
main() {
    int myTemp;
}
```

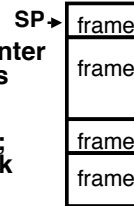


CS61CL L02 Dynamic Storage (10)

Huddleston, Summer 2009 © UCB

## The Stack

- Stack frame includes:
  - Return “instruction” address
  - Parameters
  - Space for other local variables
- Stack frames contiguous blocks of memory; stack pointer tells where top stack frame is
- When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames

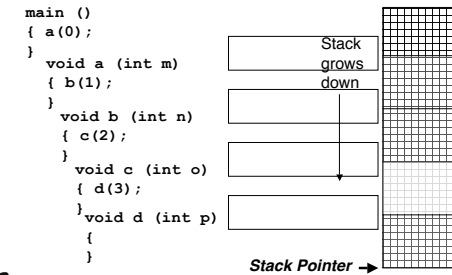


CS61CL L02 Dynamic Storage (11)

Huddleston, Summer 2009 © UCB

## Stack

- Last In, First Out (LIFO) data structure



CS61CL L02 Dynamic Storage (12)

Huddleston, Summer 2009 © UCB

## Who cares about stack management?

- Pointers in C allow access to deallocated memory, leading to hard-to-find bugs!

```
int *ptr() {
    int y;
    y = 3;
    return &y;
};
main() {
    int *stackAddr, content;
    stackAddr = ptr();
    content = *stackAddr;
    printf("%d", content); /* 3 */
    content = *stackAddr;
    printf("%d", content); /* 13451514 */
}
```



CS61CL L02 Dynamic Storage (13)

Huddleston, Summer 2009 © UCB

## The Heap (Dynamic memory)

- Large pool of memory, **not** allocated in contiguous order
  - back-to-back requests for heap memory could result blocks very far apart
  - where Java new command allocates memory
- In C, specify number of **bytes** of memory explicitly to allocate item
 

```
int *ptr;
ptr = (int *) malloc(sizeof(int));
/* malloc returns type (void *),
so need to cast to right type */
```

  - malloc(): Allocates raw, uninitialized memory from heap



CS61CL L02 Dynamic Storage (14)

Huddleston, Summer 2009 © UCB

## Memory Management

- How do we manage memory?
- Code, Static storage are easy: they never grow or shrink
- Stack space is also easy: stack frames are created and destroyed in last-in, first-out (LIFO) order
- Managing the heap is tricky: memory can be allocated / deallocated at any time



CS61CL L02 Dynamic Storage (15)

Huddleston, Summer 2009 © UCB

## Heap Management Requirements

- Want malloc() and free() to run quickly.
- Want minimal memory overhead
- Want to avoid **fragmentation\*** – when most of our free memory is in many small chunks
  - In this case, we might have many free bytes but not be able to satisfy a large request since the free bytes are not contiguous in memory.

\* This is technically called *external fragmentation*

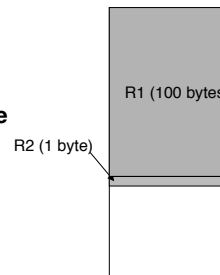


CS61CL L02 Dynamic Storage (16)

Huddleston, Summer 2009 © UCB

## Heap Management

- An example
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed
  - Request R3 for 50 bytes

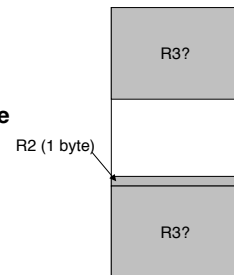


CS61CL L02 Dynamic Storage (17)

Huddleston, Summer 2009 © UCB

## Heap Management

- An example
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed
  - Request R3 for 50 bytes



CS61CL L02 Dynamic Storage (18)

Huddleston, Summer 2009 © UCB

## K&R Malloc/Free Implementation

- From Section 8.7 of K&R
  - Code in the book uses some C language features we haven't discussed and is written in a very terse style, don't worry if you can't decipher the code
- Each block of memory is preceded by a header that has two fields: size of the block and a pointer to the next block
- All free blocks are kept in a circular linked list, the pointer field is unused in an allocated block



CS61CL L02 Dynamic Storage (19)

Huddleston, Summer 2009 © UCB

## K&R Implementation

- `malloc()` searches the free list for a block that is big enough. If none is found, more memory is requested from the operating system. If what it gets can't satisfy the request, it fails.
- `free()` checks if the blocks adjacent to the freed block are also free
  - If so, adjacent free blocks are merged (coalesced) into a single, larger free block
  - Otherwise, the freed block is just added to the free list



CS61CL L02 Dynamic Storage (20)

Huddleston, Summer 2009 © UCB

## Choosing a block in `malloc()`

- If there are multiple free blocks of memory that are big enough for some request, how do we choose which one to use?
  - best-fit: choose the smallest block that is big enough for the request
  - first-fit: choose the first block we see that is big enough
  - next-fit: like first-fit but remember where we finished searching and resume searching from there



CS61CL L02 Dynamic Storage (21)

Huddleston, Summer 2009 © UCB

## Slab Allocator

- A different approach to memory management (used in GNU `libc`)
- Divide blocks in to “large” and “small” by picking an arbitrary threshold size. Blocks larger than this threshold are managed with a freelist (as before).
- For small blocks, allocate blocks in sizes that are powers of 2
  - e.g., if program wants to allocate 20 bytes, actually give it 32 bytes



CS61CL L02 Dynamic Storage (22)

Huddleston, Summer 2009 © UCB

## Slab Allocator

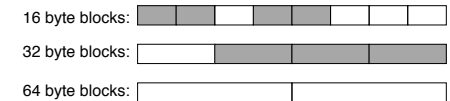
- Bookkeeping for small blocks is relatively easy: just use a *bitmap* for each range of blocks of the same size
- Allocating is easy and fast: compute the size of the block to allocate and find a free bit in the corresponding bitmap.
- Freeing is also easy and fast: figure out which slab the address belongs to and clear the corresponding bit.



CS61CL L02 Dynamic Storage (23)

Huddleston, Summer 2009 © UCB

## Slab Allocator



16 byte block bitmap: 11011000

32 byte block bitmap: 0111

64 byte block bitmap: 00



CS61CL L02 Dynamic Storage (24)

Huddleston, Summer 2009 © UCB

## Slab Allocator Tradeoffs

- Extremely fast for small blocks.
- Slower for large blocks
  - But presumably the program will take more time to do something with a large block so the overhead is not as critical.
- Minimal space overhead
- No fragmentation (as we defined it before) for small blocks, but still have wasted space!



CS61CL L02 Dynamic Storage (25)

Huddleston, Summer 2009 © UCB

## Internal vs. External Fragmentation

- With the slab allocator, difference between requested size and next power of 2 is wasted
  - e.g., if program wants to allocate 20 bytes and we give it a 32 byte block, 12 bytes are unused.
- We also refer to this as fragmentation, but call it *internal* fragmentation since the wasted space is actually within an allocated block.
- External fragmentation: wasted space between allocated blocks.



CS61CL L02 Dynamic Storage (26)

Huddleston, Summer 2009 © UCB

## Buddy System

- Yet another memory management technique (used in Linux kernel)
- Like GNU's “slab allocator”, but only allocate blocks in sizes that are powers of 2 (internal fragmentation is possible)
- Keep separate free lists for each size
  - e.g., separate free lists for 16 byte, 32 byte, 64 byte blocks, etc.



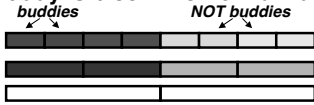
CS61CL L02 Dynamic Storage (27)

Huddleston, Summer 2009 © UCB

## Buddy System

- If no free block of size  $n$  is available, find a block of size  $2n$  and split it in to two blocks of size  $n$
- When a block of size  $n$  is freed, if its neighbor of size  $n$  is also free, combine the blocks in to a single block of size  $2n$

- Buddy is block in other half larger block



- Same speed advantages as slab allocator



CS61CL L02 Dynamic Storage (28)

Huddleston, Summer 2009 © UCB

## Allocation Schemes

- So which memory management scheme (K&R, slab, buddy) is best?
  - There is no single best approach for every application.
  - Different applications have different allocation / deallocation patterns.
  - A scheme that works well for one application may work poorly for another application.



CS61CL L02 Dynamic Storage (29)

Huddleston, Summer 2009 © UCB

## 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?



CS61CL L02 Dynamic Storage (30)

Huddleston, Summer 2009 © UCB

## 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?



CS61CL L02 Dynamic Storage (31)

Huddleston, Summer 2009 © UCB

## 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 (61A's Scheme not "simple")
- 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



CS61CL L02 Dynamic Storage (32)

Huddleston, Summer 2009 © UCB

## 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.
- Simple assignment statements can result in a lot of work, since may update reference counts of many items



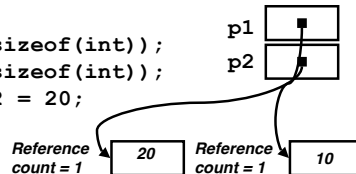
CS61CL L02 Dynamic Storage (33)

Huddleston, Summer 2009 © UCB

## 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;
```



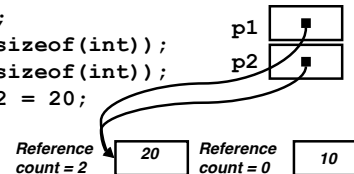
CS61CL L02 Dynamic Storage (34)

Huddleston, Summer 2009 © UCB

## 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;
```



CS61CL L02 Dynamic Storage (35)

Huddleston, Summer 2009 © UCB

## 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.

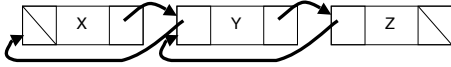


CS61CL L02 Dynamic Storage (36)

Huddleston, Summer 2009 © UCB

## 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:



CS61CL L02 Dynamic Storage (37)

Huddleston, Summer 2009 © UCB

## Scheme 2: Mark and Sweep Garbage Col.

- Keep allocating new memory until memory is exhausted, then try to find unused memory.
- Consider objects in heap 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



CS61CL L02 Dynamic Storage (38)

Huddleston, Summer 2009 © UCB

## 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)



CS61CL L02 Dynamic Storage (39)

Huddleston, Summer 2009 © UCB

## 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

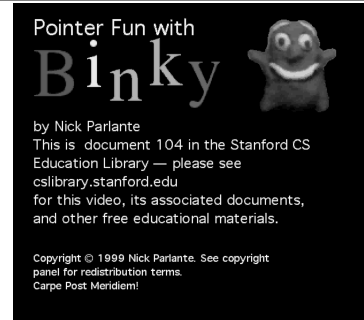
# BONUS



CS61CL L02 Dynamic Storage (40)

Huddleston, Summer 2009 © UCB

## Binky Pointer Video (thanks to NP @ SU)



Check out this video on the class website (click the link for this lecture)



CS61CL L02 Dynamic Storage (41)

Huddleston, Summer 2009 © UCB

## Kilo, Mega, Giga, Tera, Peta, Exa, Zetta, Yotta

1. Kid meets giant Texas people exercising zen-like yoga. – Rolf O
2. Kind men give ten percent extra, zestfully, youthfully. – Hava E
3. Kissing Mentors Gives Testy Persistent Extremists Zealous Youthfulness. – Gary M
4. Kindness means giving, teaching, permeating excess zeal yourself. – Hava E
5. Killing messengers gives terrible people exactly zero, yo
6. Kindergarten means giving teachers perfect examples (of) zeal (&) youth
7. Kissing mediocre girls/guys teaches people (to) expect zero (from) you
8. Kinky Mean Girls Teach Penis-Extending Zen Yoga
9. Kissing Mel Gibson, Teddy Pendergrass exclaimed: “Zesty, yo!” – Dan G
10. Kissing me gives ten percent extra zeal & youth! – Dan G (borrowing parts)



CS61CL L02 Dynamic Storage (42)

Huddleston, Summer 2009 © UCB

## C structures : Overview

- A struct is a data structure composed from simpler data types.
  - Like a class in Java/C++ but without methods or inheritance.

```
struct point { /* type definition */
    int x;
    int y;
};

void PrintPoint(struct point p)
{ As always in C, the argument is passed by “value” – a copy is made.
  printf(“(%d,%d)”, p.x, p.y);
}

struct point p1 = {0,10}; /* x=0, y=10 */
PrintPoint(p1);
```



CS61CL L02 Dynamic Storage (43)

Huddleston, Summer 2009 © UCB

## C structures: Pointers to them

- Usually, more efficient to pass a pointer to the struct.
- The C arrow operator (->) dereferences and extracts a structure field with a single operator.
- The following are equivalent:

```
struct point *p;
/* code to assign to pointer */
printf(“x is %d\n”, (*p).x);
printf(“x is %d\n”, p->x);
```



CS61CL L02 Dynamic Storage (44)

Huddleston, Summer 2009 © UCB

## How big are structs?

- Recall C operator sizeof() which gives size in bytes (of type or variable)
- How big is sizeof(p) ?

```
struct p {
    char x;
    int y;
};
```

- 5 bytes? 8 bytes?
- Compiler may word align integer y



CS61CL L02 Dynamic Storage (45)

Huddleston, Summer 2009 © UCB

## Linked List Example

- Let's look at an example of using structures, pointers, malloc(), and free() to implement a linked list of strings.

```
/* node structure for linked list */
struct Node {
    char *value;
    struct Node *next;
};
```

Recursive definition!



CS61CL L02 Dynamic Storage (46)

Huddleston, Summer 2009 © UCB

## typedef simplifies the code

```
struct Node {
    char *value;
    struct Node *next;
};

/* "typedef" means define a new type */
typedef struct Node NodeStruct;
... OR ...
typedef struct Node {
    char *value;
    struct Node *next;
} NodeStruct;

... THEN

typedef NodeStruct *List;
typedef char *String;
```

String value;

```
/* Note similarity! */
/* To define 2 nodes */

struct Node {
    char *value;
    struct Node *next;
} node1, node2;
```



CS61CL L02 Dynamic Storage (47)

Huddleston, Summer 2009 © UCB

## Linked List Example

```
/* Add a string to an existing list */
List cons(String s, List list)
{
    List node = (List) malloc(sizeof(NodeStruct));

    node->value = (String) malloc (strlen(s) + 1);
    strcpy(node->value, s);
    node->next = list;
    return node;
}

String s1 = "abc", s2 = "cde";
List theList = NULL;
theList = cons(s2, theList);
theList = cons(s1, theList);
/* or, just like (cons s1 (cons s2 nil)) */
theList = cons(s1, cons(s2, NULL));
```



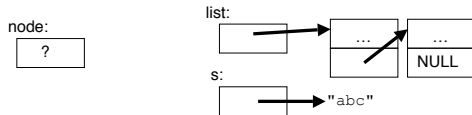
CS61CL L02 Dynamic Storage (48)

Huddleston, Summer 2009 © UCB

## Linked List Example

```
/* Add a string to an existing list, 2nd call */
List cons(String s, List list)
{
    List node = (List) malloc(sizeof(NodeStruct));

    node->value = (String) malloc (strlen(s) + 1);
    strcpy(node->value, s);
    node->next = list;
    return node;
}
```



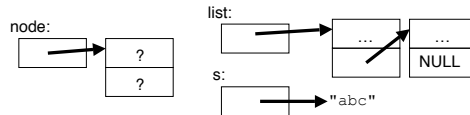
CS61CL L02 Dynamic Storage (49)

Huddleston, Summer 2009 © UCB

## Linked List Example

```
/* Add a string to an existing list, 2nd call */
List cons(String s, List list)
{
    List node = (List) malloc(sizeof(NodeStruct));

    node->value = (String) malloc (strlen(s) + 1);
    strcpy(node->value, s);
    node->next = list;
    return node;
}
```



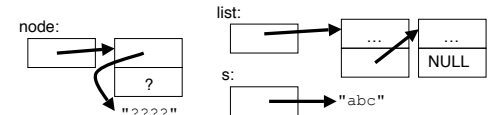
CS61CL L02 Dynamic Storage (50)

Huddleston, Summer 2009 © UCB

## Linked List Example

```
/* Add a string to an existing list, 2nd call */
List cons(String s, List list)
{
    List node = (List) malloc(sizeof(NodeStruct));

    node->value = (String) malloc (strlen(s) + 1);
    strcpy(node->value, s);
    node->next = list;
    return node;
}
```



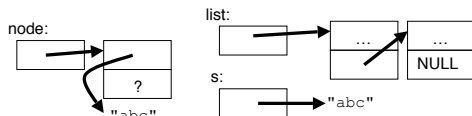
CS61CL L02 Dynamic Storage (51)

Huddleston, Summer 2009 © UCB

## Linked List Example

```
/* Add a string to an existing list, 2nd call */
List cons(String s, List list)
{
    List node = (List) malloc(sizeof(NodeStruct));

    node->value = (String) malloc (strlen(s) + 1);
    strcpy(node->value, s);
    node->next = list;
    return node;
}
```



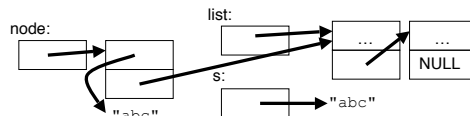
CS61CL L02 Dynamic Storage (52)

Huddleston, Summer 2009 © UCB

## Linked List Example

```
/* Add a string to an existing list, 2nd call */
List cons(String s, List list)
{
    List node = (List) malloc(sizeof(NodeStruct));

    node->value = (String) malloc (strlen(s) + 1);
    strcpy(node->value, s);
    node->next = list;
    return node;
}
```



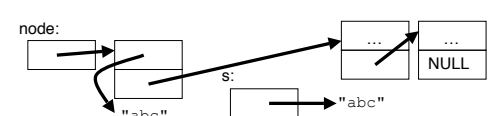
CS61CL L02 Dynamic Storage (53)

Huddleston, Summer 2009 © UCB

## Linked List Example

```
/* Add a string to an existing list, 2nd call */
List cons(String s, List list)
{
    List node = (List) malloc(sizeof(NodeStruct));

    node->value = (String) malloc (strlen(s) + 1);
    strcpy(node->value, s);
    node->next = list;
    return node;
}
```



CS61CL L02 Dynamic Storage (54)

Huddleston, Summer 2009 © UCB

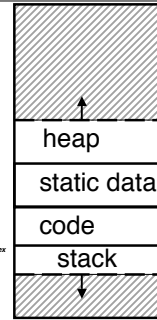
## C Memory Management

- C has 3 primary pools of memory
  - **Static storage:** global variable storage, basically permanent, entire program run
  - **The Stack:** local variable storage, parameters, return address (location of “activation records” in Java or “stack frame” in C)
  - **The Heap** (dynamic malloc storage): data lives until deallocated by programmer
- C requires knowing where objects are in memory, otherwise things don't work as expected
- Java hides location of objects



## Intel 80x86 C Memory Management

- A C program's 80x86 address space :
  - heap: space requested for pointers via `malloc()`; resizes dynamically, grows upward
  - static data: variables declared outside main, does not grow or shrink
  - code: loaded when program starts, does not change
  - stack: local variables, grows downward



## Tradeoffs of allocation policies

- **Best-fit:** Tries to limit fragmentation but at the cost of time (must examine all free blocks for each malloc). Leaves lots of small blocks (why?)
- **First-fit:** Quicker than best-fit (why?) but potentially more fragmentation. Tends to concentrate small blocks at the beginning of the free list (why?)
- **Next-fit:** Does not concentrate small blocks at front like first-fit, should be faster as a result.

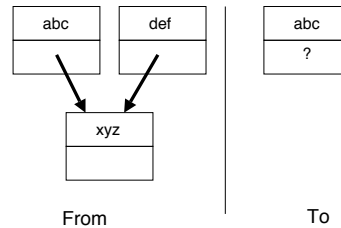


## Scheme 3: Copying Garbage Collection

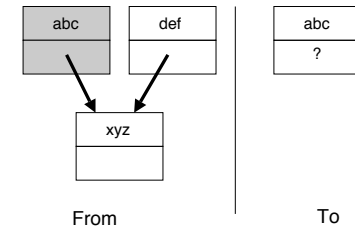
- Divide memory into two spaces, only one in use at any time.
- When active space is exhausted, traverse the active space, copying all objects to the other space, then make the new space active and continue.
  - Only reachable objects are copied!
- Use “forwarding pointers” to keep consistency
  - Simple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



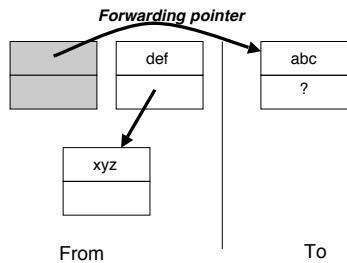
## Forwarding Pointers: 1st copy “abc”



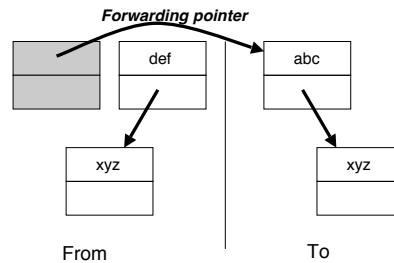
## Forwarding Pointers: leave ptr to new abc



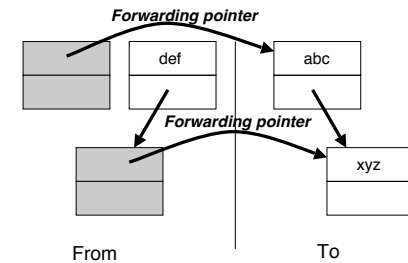
## Forwarding Pointers : now copy “xyz”



## Forwarding Pointers: leave ptr to new xyz



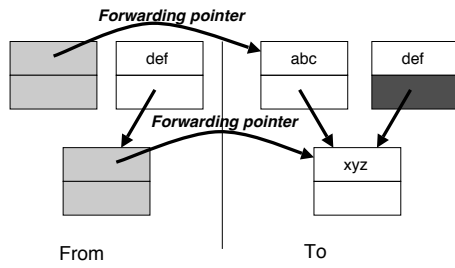
## Forwarding Pointers: now copy “def”



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*

