

Lecture 5 – Introduction to C (pt 3)  
C Memory Management



2010-01-29

Lecturer SOE Dan Garcia

www.cs.berkeley.edu/~ddgarcia

Apple's iPad, day 2 ⇒  
After the dust has settled,  
what do we have? Name causes  
chuckles & lawsuits (Fujitsu). "Haters"  
say nothing new, closed system.



apple.com/ipad



Pointers (1/4)

...review...

- Sometimes you want to have a procedure increment a variable?
- What gets printed?

```
void AddOne(int x)           y = 5
{   x = x + 1;   }

int y = 5;
AddOne(y);
printf("y = %d\n", y);
```



Pointers (2/4)

...review...

- Solved by passing in a pointer to our subroutine.
- Now what gets printed?

```
void AddOne(int *p)         y = 6
{   *p = *p + 1;   }

int y = 5;
AddOne(&y);
printf("y = %d\n", y);
```



Pointers (3/4)

- But what if what you want changed is a pointer?
- What gets printed?

```
void IncrementPtr(int *p)   *q = 50
{   p = p + 1;   }

int A[3] = {50, 60, 70};
int *q = A;
IncrementPtr(q);
printf("*q = %d\n", *q);
```



Pointers (4/4)

- Solution! Pass a pointer to a pointer, declared as \*\*h
- Now what gets printed?

```
void IncrementPtr(int **h)  *q = 60
{   *h = *h + 1;   }

int A[3] = {50, 60, 70};
int *q = A;
IncrementPtr(&q);
printf("*q = %d\n", *q);
```



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.



CS81C L05 Introduction to C (pt 3) (8)

Garcia, Spring 2010 © UCB

## 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 **free**s all memory on **exit** (or when **main** returns), don't be lazy!

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



CS81C L05 Introduction to C (pt 3) (9)

Garcia, Spring 2010 © UCB

## 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!



CS81C L05 Introduction to C (pt 3) (10)

Garcia, Spring 2010 © UCB

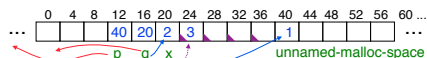
## Arrays not implemented as you'd think

```
void foo() {
    int *p, *q, x;
    int a[4];
    p = (int *) malloc (sizeof(int));
    q = &x;

    *p = 1; // p[0] would also work here
    printf("*p:%u, p:%u, &p:%u\n", *p, p, &p);

    *q = 2; // q[0] would also work here
    printf("*q:%u, q:%u, &q:%u\n", *q, q, &q);

    *a = 3; // a[0] would also work here
    printf("*a:%u, a:%u, &a:%u\n", *a, a, &a);
}
```



CS81C L05 Introd

K&R: "An array name is not a variable"

Garcia, Spring 2010 © 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, Tom Petty exclaimed: "Zesty, yo!" – Dan G
10. Kissing me gives ten percent extra zeal & youth! – Dan G (borrowing parts)



CS81C L05 Introduction to C (pt 3) (13)

Garcia, Spring 2010 © UCB

## "And in Conclusion..."

- Use handles to change pointers
- Create abstractions with structures
- Dynamically allocated heap memory must be manually deallocated in C.
  - Use `malloc()` and `free()` to allocate and deallocate memory from heap.



CS81C L05 Introduction to C (pt 3) (15)

Garcia, Spring 2010 © 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);
```



## 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);
```



## 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
```



## 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;
};
```



## 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;
```



## 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);
}

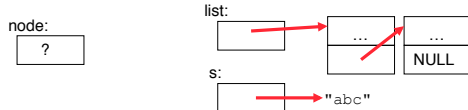
/* or, just like (cons s1 (cons s2 nil)) */
theList = cons(s1, cons(s2, NULL));
```



## 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;
}
```



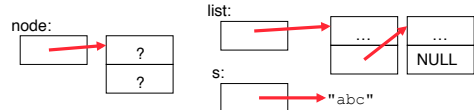
CS61C L05 Introduction to C (pt 3) (23)

Garcia, Spring 2010 © 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;
}
```



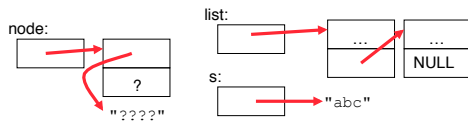
CS61C L05 Introduction to C (pt 3) (24)

Garcia, Spring 2010 © 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;
}
```



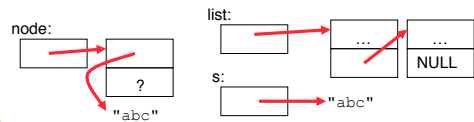
CS61C L05 Introduction to C (pt 3) (25)

Garcia, Spring 2010 © 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;
}
```



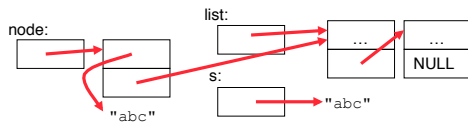
CS61C L05 Introduction to C (pt 3) (26)

Garcia, Spring 2010 © 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;
}
```



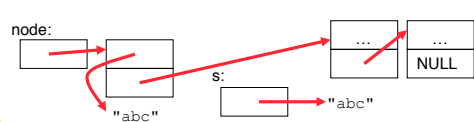
CS61C L05 Introduction to C (pt 3) (27)

Garcia, Spring 2010 © 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;
}
```



CS61C L05 Introduction to C (pt 3) (28)

Garcia, Spring 2010 © UCB