

CS61C – Machine Structures

Lecture 5 – C Structs & Memory Mangement

1/27/2006

John Wawrzynek

(www.cs.berkeley.edu/~johnw)

www-inst.eecs.berkeley.edu/~cs61c/

CS 61C L05 C Structs (1)

Wawrzynek Spring 2006 © UCB

C String Standard Functions

```
int strlen(char *string);
```

- compute the length of `string`

```
int strcmp(char *str1, char *str2);
```

- return 0 if `str1` and `str2` are identical (how is this different from `str1 == str2`?)

```
char *strcpy(char *dst, char *src);
```

- copy the contents of string `src` to the memory at `dst`. The caller must ensure that `dst` has enough memory to hold the data to be copied.

CS 61C L05 C Structs (2)

Wawrzynek Spring 2006 © UCB

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);
```

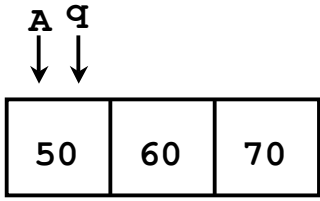
Pointers (3/4)

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

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

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

***q = 50**



The diagram shows a horizontal array with three cells containing the values 50, 60, and 70. Above the first cell, the label 'A' has a downward arrow pointing to the cell. Above the second cell, the label 'q' has a downward arrow pointing to the cell. This indicates that pointer 'q' points to the first element of array 'A', which contains the value 50.

CS 61C L05 C Structs (5)

Wawrzynek Spring 2006 © UCB

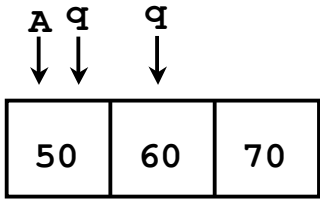
Pointers (4/4)

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

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

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

***q = 60**



The diagram shows a horizontal array with three cells containing the values 50, 60, and 70. Above the first cell, the label 'A' has a downward arrow pointing to the cell. Above the second cell, the label 'q' has a downward arrow pointing to the cell. Above the third cell, the label 'q' has a downward arrow pointing to the cell. This indicates that pointer 'q' now points to the second element of array 'A', which contains the value 60.

CS 61C L05 C Structs (6)

Wawrzynek Spring 2006 © UCB

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” is not a real function - it does its work at compile time. So:
 - `char foo[3*sizeof(int)]` is valid,
 - But neither of these is valid in C:
`char foo[3*myfunction(int)]`
`char foo[3*myfunction(7)]`

CS 61C L05 C Structs (7)

Wawrzynek Spring 2006 © UCB

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.

CS 61C L05 C Structs (8)

Wawrzynek Spring 2006 © 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.

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), and the usual result is that you corrupt the memory allocator's internal structures -- but you won't find out until much later on, in some totally unrelated part of your code.

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};

PrintPoint(p1);
```

CS 61C L05 C Structs (11)

Wawrzynek Spring 2006 © 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);
```

CS 61C L05 C Structs (12)

Wawrzynek Spring 2006 © 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

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;  
};
```

Linked List Example

```
/* add a string to an existing list */
Node * list_add(Node *list, char *string)
{
    struct Node *node =
        (struct Node *) malloc(sizeof(struct Node));

    node->value =
        (char *) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}

{
    char *s1 = "abc", *s2 = "cde";
    Node *theList = NULL;

    theList = list_add(theList, s1);
    theList = list_add(theList, s2);
}
```

CS 61C L05 C Structs (15)

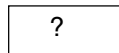
Wawrzynek Spring 2006 © UCB

Linked List Example

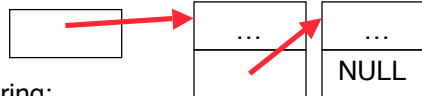
```
/* add a string to an existing list, 2nd call */
Node * list_add(Node *list, char *string)
{
    struct Node *node =
        (struct Node *) malloc(sizeof(struct Node));

    node->value =
        (char *) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}
```

node:



list:



string:



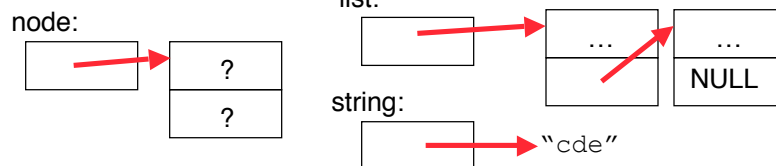
CS 61C L05 C Structs (16)

Wawrzynek Spring 2006 © UCB

Linked List Example

```
/* add a string to an existing list */
Node * list_add(Node *list, char *string)
{
    struct Node *node =
        (struct Node *) malloc(sizeof(struct Node));

    node->value =
        (char *) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}
```



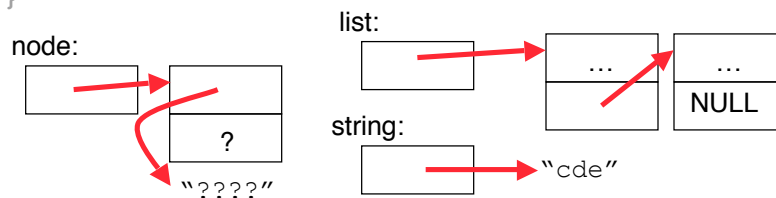
CS 61C L05 C Structs (17)

Wawrzynek Spring 2006 © UCB

Linked List Example

```
/* add a string to an existing list */
Node * list_add(Node *list, char *string)
{
    struct Node *node =
        (struct Node *) malloc(sizeof(struct Node));

    node->value =
        (char *) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}
```



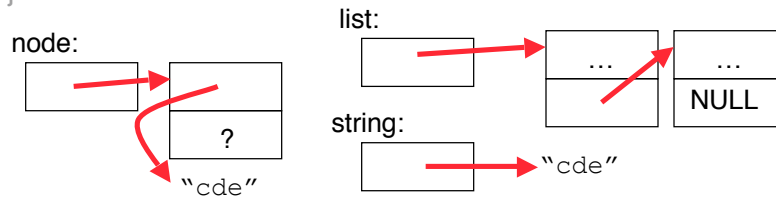
CS 61C L05 C Structs (18)

Wawrzynek Spring 2006 © UCB

Linked List Example

```
/* add a string to an existing list */
Node * list_add(Node *list, char *string)
{
    struct Node *node =
        (struct Node *) malloc(sizeof(struct Node));

    node->value =
        (char *) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}
```



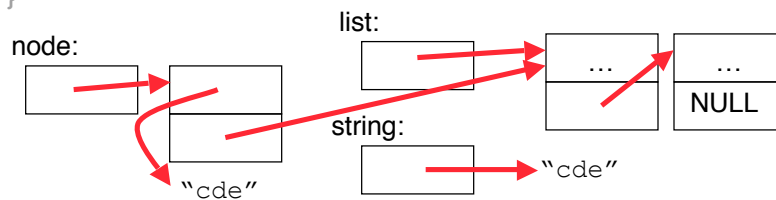
CS 61C L05 C Structs (19)

Wawrzynek Spring 2006 © UCB

Linked List Example

```
/* add a string to an existing list */
Node * list_add(Node *list, char *string)
{
    struct Node *node =
        (struct Node *) malloc(sizeof(struct Node));

    node->value =
        (char *) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}
```



CS 61C L05 C Structs (20)

Wawrzynek Spring 2006 © UCB

Linked List Example

```
/* add a string to an existing list */
Node * list_add(Node *list, char *string)
{
    struct Node *node =
        (struct Node *) malloc(sizeof(struct Node));

    node->value =
        (char *) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}
```



Errata on struct Example

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

struct Node * list_add(struct Node *list, char *string)
{
    /* body of function */
    return node;
}
```

“Typedef” Simplifies the code

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

```
Typedef struct Node NodeStruct; /* a new type */
```

OR

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

THEN

```
NodeStruct * list_add(NodeStruct *list, char *string)
{
    /* body of function */
    return node;
}
```

CS 61C L05 C Structs (23)

Wawrzynek Spring 2006 © UCB

“And in Conclusion...”

- Can use pointers 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.

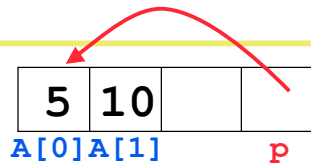
CS 61C L05 C Structs (24)

Wawrzynek Spring 2006 © UCB

Question

```
int main(void){
  int A[] = {5,10};
  int *p = A;
```

```
  printf("%u %d %d %d\n", p, *p, A[0], A[1]);
  p = p + 1;
  printf("%u %d %d %d\n", p, *p, A[0], A[1]);
  *p = *p + 1;
  printf("%u %d %d %d\n", p, *p, A[0], A[1]);
}
```



If the first `printf` outputs `100 5 5 10`, what will the other two `printf` output?

- 1: 101 10 5 10 then 101 11 5 11
- 2: 104 10 5 10 then 104 11 5 11
- 3: 101 <other> 5 10 then 101 <3-others>
- 4: 104 <other> 5 10 then 104 <3-others>
- 5: One of the two `printfs` causes an ERROR
- 6: I surrender!