CS61C : Machine Structures

Lecture #4: Strings & Structs

2005-06-23
Andy Carle

Review: Arrays

- **Arrays are (almost) identical to pointers**
  - **`char *string` and `char string[]` are nearly identical declarations**
    - They differ in subtle ways: incrementing, declaration of filled arrays
    - **Key Difference**: an array variable is a **CONSTANT** pointer to the first element.

  - **ar[i] ⬅➡ *(ar+i)**

Review: Arrays and Pointers

- **Array size `n`; want to access from `0` to `n-1`:**

Array Indexing Versions:

```
#define ARSIZE 10
int ar[ARSIZE];
int i=0, sum = 0;

...
while (i < ARSIZE)
        sum += ar[i++];

    or

while (i < ARSIZE)
        sum += *(ar + i++);
```

Pointer Indexing Version:

```
#define ARSIZE 10
int ar[ARSIZE];
int *p = ar, *q = &ar[10]*;
int sum = 0;

...
while (p < q)
        sum += *p++;
```

* C allows 1 past end of array!

Review: Common C Errors

- **There is a difference between assignment and equality**

  - $a = b$ is assignment

  - $a == b$ is an equality test

- **This is one of the most common errors for beginning C programmers!**

- **Precedence Rules**

  - int **a = {{1, 2}, {3, 4}}

  - *a[1]++;   ([ ] > *)

Topic Outline

- **Strings**

- **Handles**

- **Structs**

- **Heap Allocation Intro**

- **Linked List Example**

C Strings (1/3)

- A **string** in C is just an array of characters.

```
char string[] = "abc";
```

- How do you tell how long a string is?
  - Last character is followed by a 0 byte (null terminator)

```
int strlen(char s[])
{
    int n = 0;
    while (s[n] != 0) n++; /* '\0' */
    return n;
}
```

C Strings Headaches (2/3)

- **One common mistake is to forget to allocate an extra byte for the null terminator.**

- **More generally, C requires the programmer to manage memory manually (unlike Java or C++).**
  - **When creating a long string by concatenating several smaller strings, the programmer must insure there is enough space to store the full string!**
  - **What if you don't know ahead of time how big your string will be?**

- **String constants are immutable:**
  - **char *f = "abc";   f[0]++;   /* illegal */**
    - **Because section of mem where "abc" lives is immutable.**
  - **char f [ ] = "abc"; f[0]++;   /* Works! */**
    - **Because, in declaration, c copies abc into space allocated for f.**

C String Standard Functions (3/3)

- `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` and return `dst`. The caller must ensure that `dst` has enough memory to hold the data to be copied.

- **Sometimes you want to have a procedure increment a variable?**

- **What gets printed?**

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

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

$y = 5$

Pointers to 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);
```
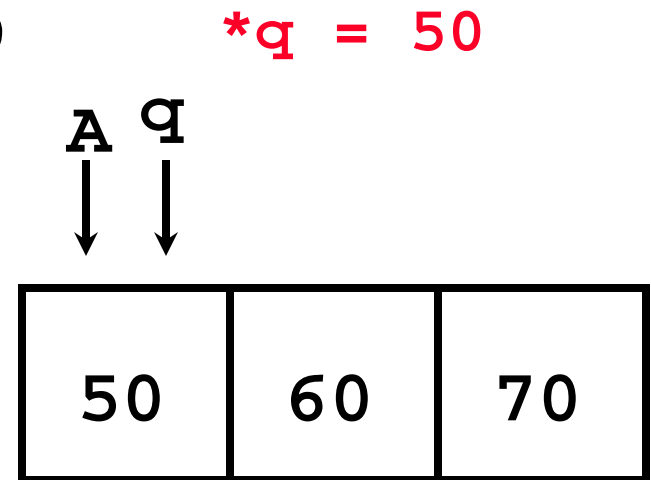
Pointers to pointers (3/4)

- **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);
```
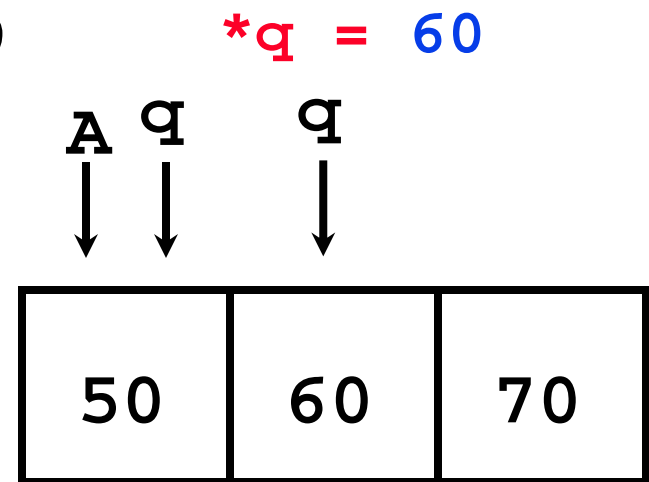
*q = 50

A q

50	60	70

Pointers to pointers (4/4)

- **Solution! Pass a pointer to a pointer, called a handle, 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);
```

A q   q

50	60	70

C structures : Overview (1/3)

- A `struct` is a data structure composed of simpler data types.
  - Like a class in Java/C++ but without methods or inheritance. Don't get hung up on this comparison.

```
struct point {
    int x;
    int y;
};
void PrintPoint(struct point p)
{
    printf("(%d,%d)", p.x, p.y);
}
```

C structures: Pointers to them (2/3)

- **The C arrow operator (->) dereferences and extracts a structure field with a single operator.**

- **The following are equivalent:**

```
struct point *p;

printf("x is %d\n", (*p).x);
printf("x is %d\n", p->x);
```

How big are structs? (3/3)

- **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

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 & is bad style, so use `sizeof(type)`

  - Many years ago an `int` was 16 bits, and programs assumed it was 2 bytes

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 might contain anything, 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.**

  - **OS keeps track of size to free.**

Dynamic Memory Allocation (4/4)

- **Malloc does <u>not</u> always succeed.**
  - **System could be out of memory**
  - **An error occurred during the memory request**
  - **Operating system just doesn't like you today…**

- **<u>Always</u> check the pointer you get back to make sure it is not NULL.**
  - ```
    int *p;
    if ((p = (int*) malloc(10 * sizeof(int))) == NULL) {
        /*do something to recover */
    }
    ```

Binky Pointer Video (thanks to NP @ SU)

Linked List Example

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

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


/* Create a new (empty) list */
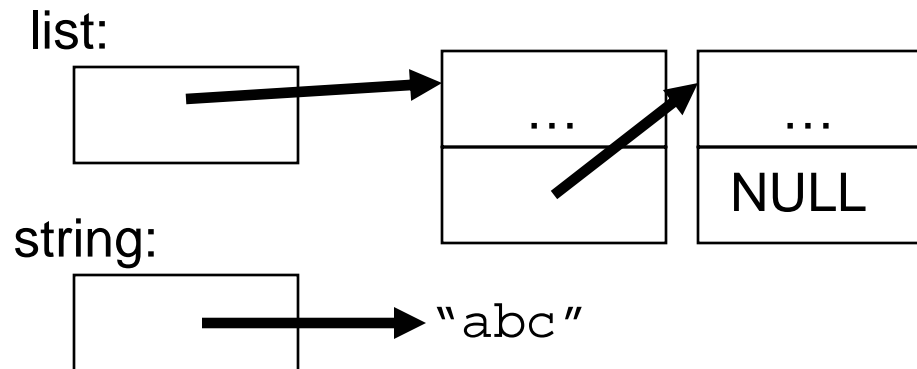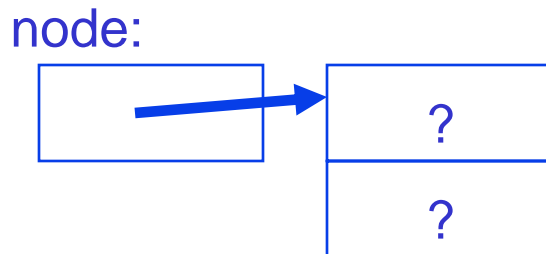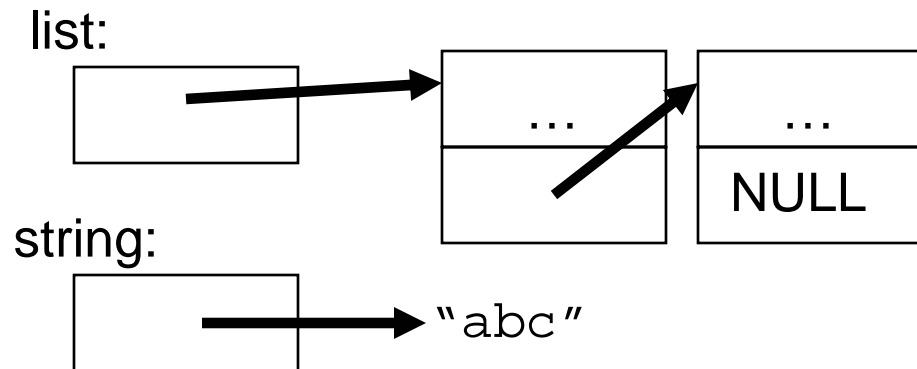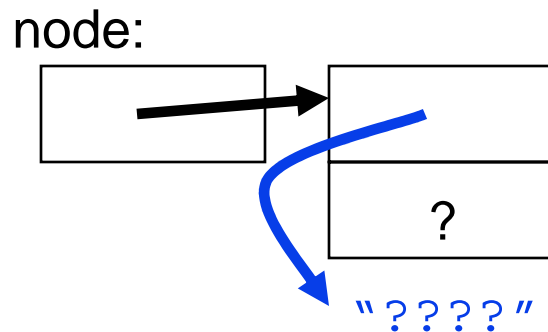List ListNew(void)
{ return NULL; }
```

Linked List Example

```
/* add a string to an existing list */
List list_add(List 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;
}
```

list:

...

... NULL

string:

"abc"

Linked List Example

```
/* add a string to an existing list */
List list_add(List 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:

...

...
NULL

string:

"abc"

Linked List Example

```c
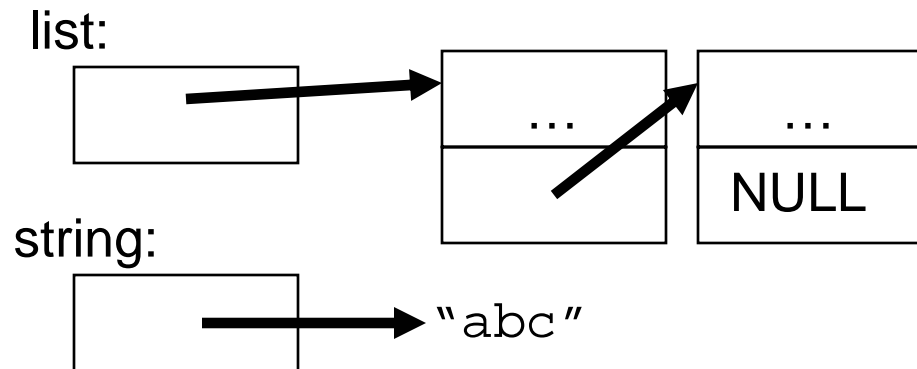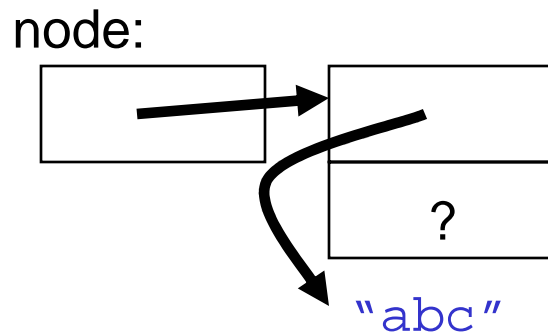/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```

node:

list:

?

"????"

string:

"abc"

NULL

Linked List Example

```
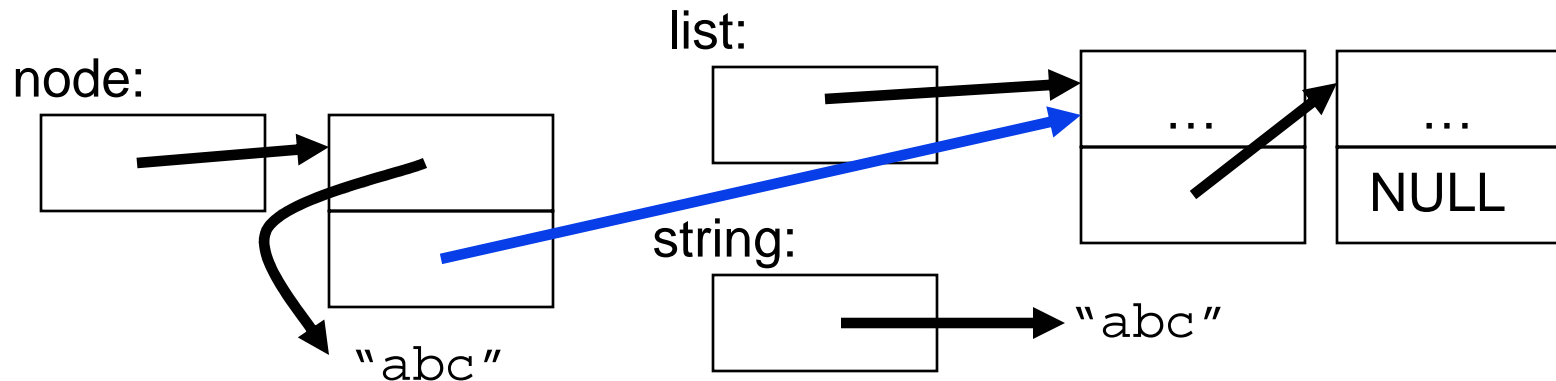/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```

node:

"abc"

list:

...

...

NULL

string:

"abc"

Linked List Example

```
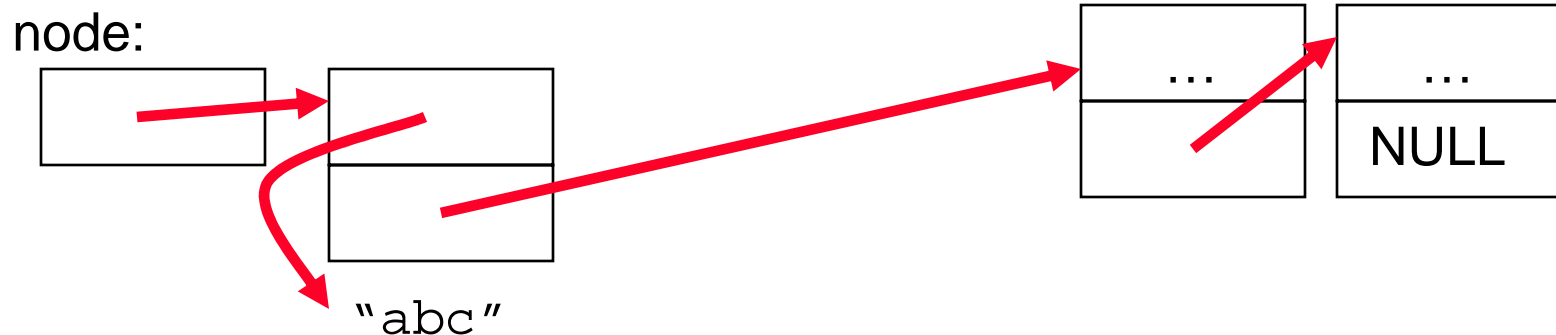/* add a string to an existing list */
List list_add(List list, char *string)
{
  struct Node *node =
    (struct Node*) malloc(sizeof(struct Node));
  node->value =
    (char*) malloc(strlen(string) + 1);
  strcpy(node->value, string);
  node->next = list;
  return node;
}
```


node:

list:

string:

"abc"

"abc"

...

...

NULL

Linked List Example

```
/* add a string to an existing list */
List list_add(List 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:

"abc"

... ... NULL

"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.**