

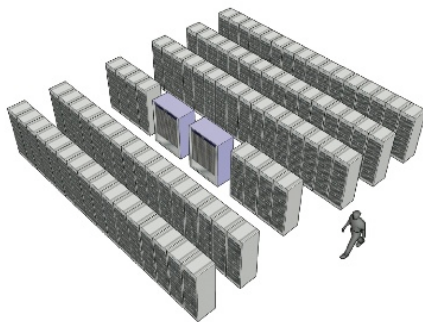
## Lecture #3 – C Strings, Arrays, & Malloc

2007-06-27

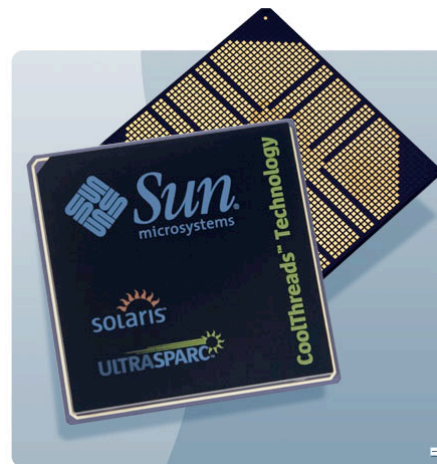


**Scott Beamer, Instructor**

**Sun announces new supercomputer:  
Sun Constellation**



TACC Ranger Cluster



# Review

---

- All declarations go at the beginning of each function.
- Only 0 and NULL evaluate to FALSE.
- All data is in memory. Each memory location has an address to use to refer to it and a value stored in it.
- A **pointer** is a C version of the address.
  - \* “follows” a pointer to its value
  - & gets the address of a value



# Has there been an update to ANSI C?

---

- **Yes! It's called the "C99" or "C9x" std**
  - Thanks to Jason Spence for the tip

- **References**

[http://en.wikipedia.org/wiki/Standard\\_C\\_library](http://en.wikipedia.org/wiki/Standard_C_library)

[http://home.tiscalinet.ch/t\\_wolf/tw/c/c9x\\_changes.html](http://home.tiscalinet.ch/t_wolf/tw/c/c9x_changes.html)

- **Highlights**

- `<inttypes.h>`: convert integer types (#38)
- `<stdbool.h>` for boolean logic def's (#35)
- `restrict` keyword for optimizations (#30)
- Named initializers (#17) for aggregate objs



# Pointers & Allocation (1/2)

---

- After declaring a pointer:

```
int *ptr;
```

`ptr` doesn't actually point to anything yet. We can either:

- make it point to something that already exists, or
- allocate room in memory for something new that it will point to... (later)



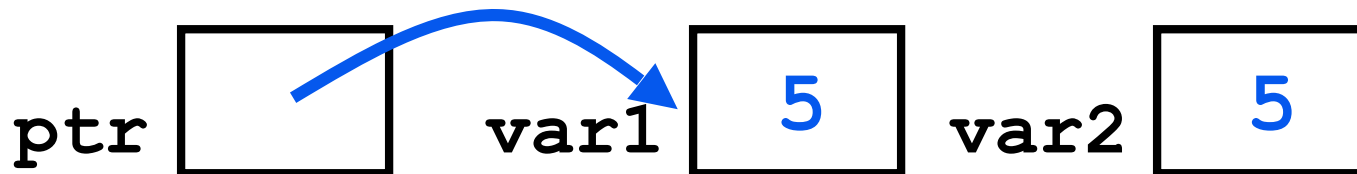
# Pointers & Allocation (2/2)

---

- **Pointing to something that already exists:**

```
int *ptr, var1, var2;  
var1 = 5;  
ptr = &var1;  
var2 = *ptr;
```

- **var1 and var2 have room implicitly allocated for them.**



# More C Pointer Dangers

---

- Declaring a pointer just allocates space to hold the pointer – it does not allocate something to be pointed to!
- **Local variables in C are not initialized**, they may contain anything.
- What does the following code do?

```
void f()  
{  
    int *ptr;  
    *ptr = 5;  
}
```



# Arrays (1/6)

---

- **Declaration:**

```
int ar[2];
```

declares a 2-element integer array.

```
int ar[] = {795, 635};
```

declares and fills a 2-elt integer array.

- **Accessing elements:**

```
ar[num];
```

returns the num<sup>th</sup> element.



## Arrays (2/6)

---

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





# Arrays (3/6)

---

- **Consequences:**
  - `ar` is an array variable but looks like a pointer in many respects (though not all)
  - `ar[0]` is the same as `*ar`
  - `ar[2]` is the same as `*(ar+2)`
  - We can use pointer arithmetic to access arrays more conveniently.
- **Declared arrays are only allocated while the scope is valid**

```
char *foo() {  
    char string[32]; ...;  
    return string;  
} is incorrect
```



## Arrays (4/6)

---

- Array size  $n$ ; want to access from 0 to  $n-1$ , but test for exit by comparing to address one element past the array

```
int ar[10], *p, *q, sum = 0;
...
p = &ar[0]; q = &ar[10];
while (p != q)
    /* sum = sum + *p; p = p + 1; */
    sum += *p++;
```

- Is this legal?
- C defines that one element past end of array **must be a valid address**, i.e., not cause an bus error or address error



# Arrays (5/6)

---

- Array size  $n$ ; want to access from 0 to  $n-1$ , so you should use counter AND utilize a constant for declaration & incr

- Wrong

```
int i, ar[10];  
for(i = 0; i < 10; i++){ ... }
```

- Right

```
#define ARRAY_SIZE 10  
int i, a[ARRAY_SIZE];  
for(i = 0; i < ARRAY_SIZE; i++){ ... }
```

- Why? **SINGLE SOURCE OF TRUTH**

- You're utilizing **indirection** and avoiding maintaining two copies of the number 10



# Arrays (6/6)

---

- **Pitfall: An array in C does not know its own length, & bounds not checked!**
  - **Consequence: We can accidentally access off the end of an array.**
  - **Consequence: We must pass the array and its size to a procedure which is going to traverse it.**
- **Segmentation faults and bus errors:**
  - **These are VERY difficult to find; be careful! (You'll learn how to debug these in lab...)**



# Pointer Arithmetic (1/4)

---

- Since a pointer is just a mem address, we can add to it to traverse an array.
- $p+1$  returns a ptr to the next array elt.
- $*p++$  vs  $(*p)++$  ?
  - $x = *p++ \Rightarrow x = *p ; p = p + 1 ;$
  - $x = (*p)++ \Rightarrow x = *p ; *p = *p + 1 ;$
- What if we have an array of large structs (objects)?
  - C takes care of it: In reality,  $p+1$  doesn't add 1 to the memory address, it adds the size of the array element.



## Pointer Arithmetic (2/4)

---

- **So what's valid pointer arithmetic?**
  - **Add an integer to a pointer.**
  - **Subtract 2 pointers (in the same array).**
  - **Compare pointers (<, <=, ==, !=, >, >=)**
  - **Compare pointer to NULL (indicates that the pointer points to nothing).**
- **Everything else is illegal since it makes no sense:**
  - **adding two pointers**
  - **multiplying pointers**
  - **subtract pointer from integer**



## Pointer Arithmetic (3/4)

---

- **C knows the size of the thing a pointer points to – every addition or subtraction moves that many bytes.**
  - 1 byte for a char, 4 bytes for an int, etc.
- **So the following are equivalent:**

```
int get(int array[], int n)
{
    return (array[n]);
    /* OR */
    return *(array + n);
}
```



## Pointer Arithmetic (4/4)

---

- We can use pointer arithmetic to “walk” through memory:

```
void copy(int *from, int *to, int n) {  
    int i;  
    for (i=0; i<n; i++) {  
        *to++ = *from++;  
    }  
}
```





# Pointers in C

---

- **Why use pointers?**
  - If we want to pass a huge struct or array, it's easier to pass a pointer than the whole thing.
  - In general, pointers allow cleaner, more compact code.
- **So what are the drawbacks?**
  - Pointers are probably the single largest source of bugs in software, so be careful anytime you deal with them.
  - **Dangling reference** (premature free)
  - **Memory leaks** (tardy free)



# C Pointer Dangers

---

- Unlike Java, C lets you **cast** a value of any type to any other type without performing any checking.

```
int x = 1000;
```

```
int *p = x;          /* invalid */
```

```
int *q = (int *) x; /* valid */
```

- The first pointer declaration is invalid since the types do not match.
- The second declaration is valid C but is almost certainly wrong



• Is it ever correct?

# Segmentation Fault vs Bus Error?

---

- <http://www.hyperdictionary.com/>
- **Bus Error**
  - A fatal failure in the execution of a machine language instruction resulting from the processor detecting an anomalous condition on its bus. Such conditions include invalid address alignment (accessing a multi-byte number at an odd address), accessing a physical address that does not correspond to any device, or some other device-specific hardware error. A bus error triggers a processor-level exception which Unix translates into a “SIGBUS” signal which, if not caught, will terminate the current process.
- **Segmentation Fault**
  - An error in which a running Unix program attempts to access memory not allocated to it and terminates with a segmentation violation error and usually a core dump.



# Administrivia

---

- **Homework expectations**
  - Readers don't have time to fix your programs which have to run on lab machines.
  - **Code that doesn't compile or fails all of the autograder tests  $\Rightarrow$  0**
- **Labs due in lab or by first 10 minutes of next lab**
- **Worried about getting into the class?**  
**stick around...**



# Administrivia

---

- **Slip days**

- You get 2 “slip days” per year to use for any assignment (except the last one of the term)
- They are used at 1-day increments. Thus 1 *minute* late = 1 slip day used.
- They’re recorded automatically (by checking submission time) so you don’t need to tell us when you’re using them
- Once you’ve used all of your slip days, when a project/hw is late, it’s ... **0** points.
- If you submit twice, we **ALWAYS** grade the latter, and deduct slip days appropriately
- You no longer need to tell anyone how your dog ate your computer.
- You should really save for a rainy day ... we all get sick and/or have family emergencies!



# C Strings

---

- 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++;  
    return n;
```



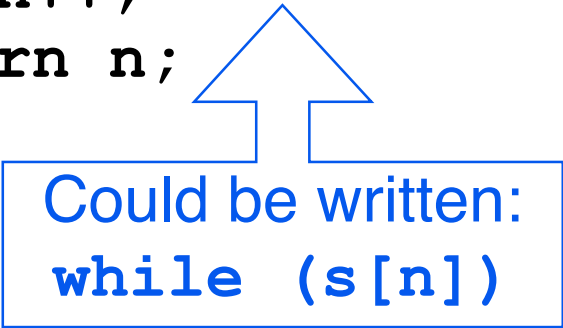
# Arrays vs. Pointers

---

- An array name is a read-only pointer to the 0<sup>th</sup> element of the array.
- An array parameter can be declared as an array or a pointer; an array argument can be passed as a pointer.

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

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



Could be written:  
while (s[n])



# C Strings Headaches

---

- **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?**
  - **Buffer overrun security holes!**





# 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!



# Pointer Arithmetic Peer Instruction Q

---

How many of the following are **invalid**?

- I. pointer + integer
- II. integer + pointer
- III. pointer + pointer
- IV. pointer – integer
- V. integer – pointer
- VI. pointer – pointer
- VII. compare pointer to pointer
- VIII. compare pointer to integer
- IX. compare pointer to 0
- X. compare pointer to NULL

<u>#invalid</u>
1
2
3
4
5
6
7
8
9
(1) 0



# Pointer Arithmetic Peer Instruction Ans

- How many of the following are **invalid**?
  - I. pointer + integer ptr + 1
  - II. integer + pointer 1 + ptr
  - III. pointer + pointer **ptr + ptr**
  - IV. pointer – integer ptr - 1
  - V. integer – pointer **1 - ptr**
  - VI. pointer – pointer ptr - ptr
  - VII. compare pointer to pointer ptr1 == ptr2
  - VIII. compare pointer to integer **ptr == 1**
  - IX. compare pointer to 0 ptr == NULL
  - X. compare pointer to NULL ptr == NULL

<u>#invalid</u>
1
2
<b>3</b>
4
5
6
7
8
9
(1) 0



# Pointer Arithmetic Summary

---

- $x = *(p+1) ?$

$\Rightarrow x = *(p+1) ;$

- $x = *p+1 ?$

$\Rightarrow x = (*p) + 1 ;$

- $x = (*p)++ ?$

$\Rightarrow x = *p ; *p = *p + 1 ;$

- $x = *p++ ? (*p++) ? *(p)++ ? * (p++) ?$

$\Rightarrow x = *p ; p = p + 1 ;$

- $x = *++p ?$

$\Rightarrow p = p + 1 ; x = *p ;$

- Lesson?



• Using anything but the standard  $*p++$ ,  $(*p)++$  causes more problems than it solves!

# 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.



# Pointers to pointers (1/4) ...review...

---

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

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

y = 5

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



## Pointers to pointers (2/4) ...review...

---

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

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

y = 6

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



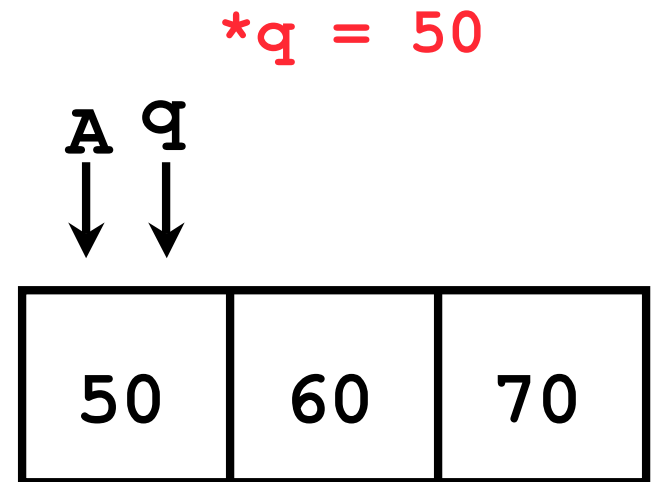
# Pointers to pointers (3/4)

---

- 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 );
```





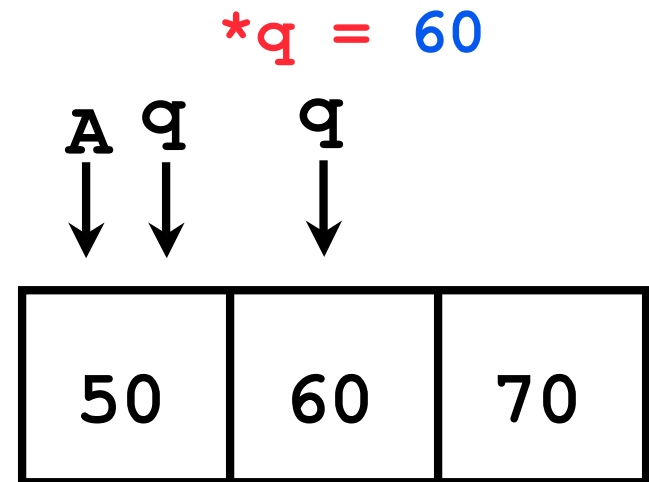
# Pointers to pointers (4/4)

---

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

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

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



## Dynamic Memory Allocation (1/3)

---

- 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 (2/3)

---

- 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 (3/3)

---

- 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.



# Binky Pointer Video (thanks to NP @ SU)

---

Pointer Fun with

**B** **i** **n** **k** **y**



by Nick Parlante

This is document 104 in the Stanford CS Education Library — please see [cslibrary.stanford.edu](http://cslibrary.stanford.edu) for this video, its associated documents, and other free educational materials.

Copyright © 1999 Nick Parlante. See copyright panel for redistribution terms.  
Carpe Post Meridiem!



## “And in Conclusion...”

---

- C99 is the update to the language
- 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!”



• Use handles to change pointers