

inst.eecs.berkeley.edu/~cs61c
CS61C : Machine Structures

Lecture 4 – Introduction to C (pt 2)

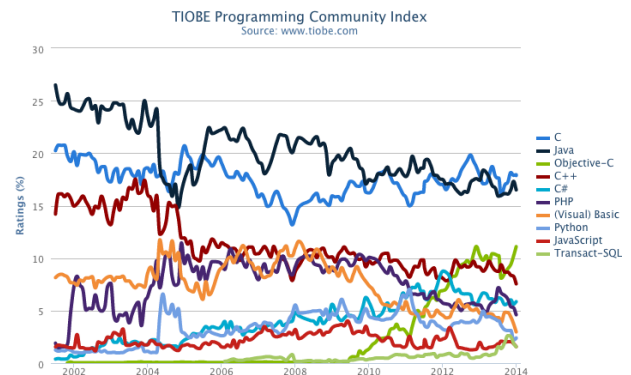


2014-01-29

Senior Lecturer SOE Dan Garcia

www.cs.berkeley.edu/~ddgarcia

C most popular! ⇒
TIOBE programming
has been tracking programming
language popularity for the past
decade, and C (in blue) is now on top!



www.tiobe.com/index.php/content/paperinfo/tpci/

Enrollment update...

Remember, all labs will use pair programming!

(both partners must know stuff, tho!)



Review

- All declarations go at the beginning of each function **except if you use C99.**
- 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
- Only 0 (i.e., NULL) evaluate to FALSE.



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 (1/5)

- **Declaration:**

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

declares a 2-element integer array. *An array is really just a block of memory.*

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

declares and fills a 2-elt integer array.

- **Accessing elements:**

```
ar[num]
```



returns the numth element.

Arrays (2/5)

- **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 (3/5)

- **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 (4/5)

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

- Wrong

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

- Right

```
int 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 (5/5)

- **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...)**



- 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 (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);
```

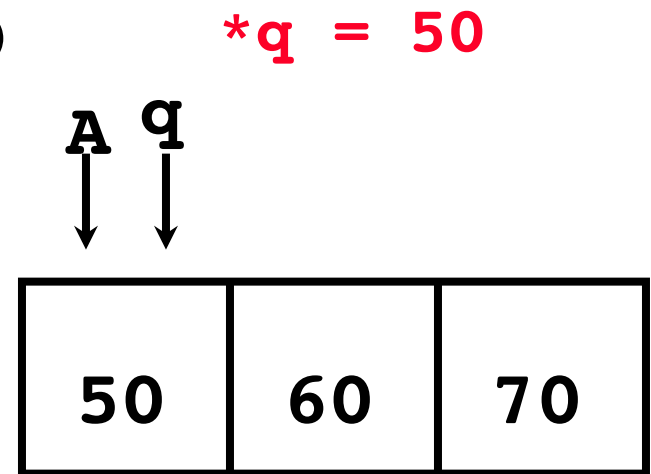


Pointers (3/4)

- 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);
```

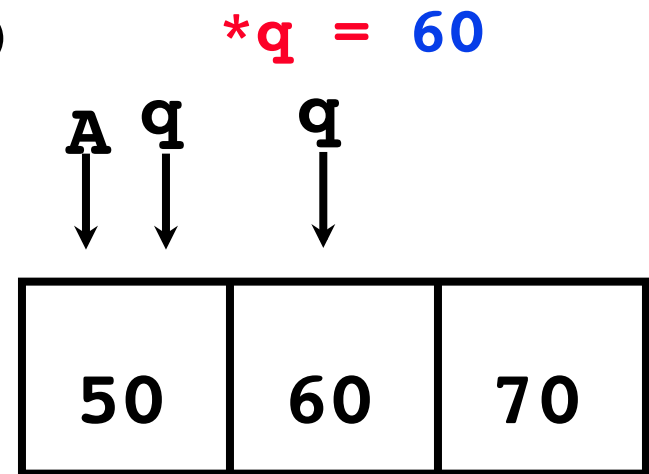


Pointers (4/4)

- **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);
```



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 **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!



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!



Pointers in C

- **Why use pointers?**
 - If we want to pass a huge struct or array, it's easier / faster / etc 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** (use ptr before malloc)
 - **Memory leaks** (tardy free, lose the ptr)



Arrays not implemented as you'd think

```

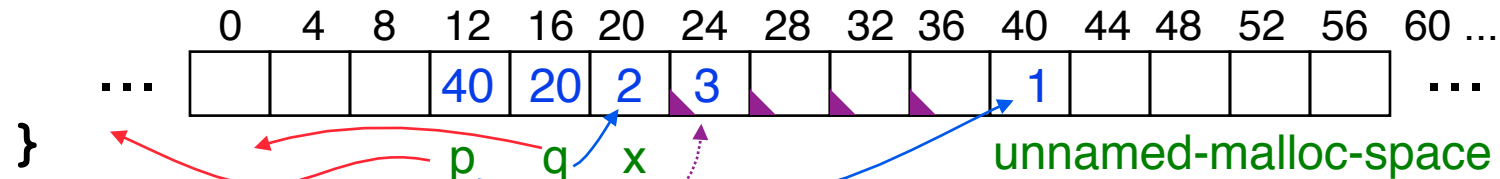
void foo() {
    int *p, *q, x;
    int a[4];
    p = (int *) malloc (sizeof(int));
    q = &x;

    *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);
}

```



?
 24
 a

*p:1, p:40, &p:12
 *q:2, q:20, &q:16
 *a:3, a:24, **&a:24**



Peer Instruction



Which are guaranteed to print out 5?

```
I: main() {
    int *a_ptr = (int *)malloc(int);
    *a_ptr = 5;
    printf("%d", *a_ptr);
}
```

```
II: main() {
    int *p, a = 5;
    p = &a; ...
    /* code; a, p NEVER on LEFT of = */
    printf("%d", a);
}
```

	<u>I</u>	<u>II</u>
a)	-	-
b)	-	YES
c)	YES	-
d)	YES	YES
e)	No idea	



Binky Pointer Video (thanks to NP @ SU)

Pointer Fun with

Binky



by Nick Parlante

This is document 104 in the Stanford CS Education Library — please see 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...”

- 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**
- **Use handles to change pointers**
- **Dynamically allocated heap memory must be manually deallocated in C.**
 - **Use `malloc()` and `free()` to allocate and deallocate memory from heap.**
- **(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!”**

