Review

- All declarations go at the beginning of each function except if you use C99.
- 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 variable

Common C Error

- There is a difference between assignment and equality
  - `a = b` is assignment
  - `a == b` is an equality test
  - If `(x = 5) {...} // This builds, and costs sleep`
- This is one of the most common errors for beginning C programmers!
  - One solution (when comparing with constant) is to put the var on the right! If you happen to use `=`, it won’t compile.
  - `if (3 == a) { ...`
Arrays (1/6)

• Declaration:
  
  ```
  int arr[2];
  ```

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

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

  declares and fills a 2-elt integer array.

• Accessing elements:
  
  ```
  arr[num]
  ```

  returns the num\textsuperscript{th} element.

Arrays (2/6)

• Arrays are closely tied with pointers
  
  • They differ in very subtle ways:
    incrementing, declaration of filled arrays, value
  
  • Pointers are frequently used to access arrays
  
  • Some pointer math can be used on arrays

• Key Concept: An array variable is a “pointer” to the first element.

Arrays (3/6)

• Arrays vs pointers
  
  ```
  int arr[] = {1,2,3};
  int *ptr = arr;
  ```

  Location (address)

  ```
  Value of & ptr:  156
  Value of & arr:  116
  Value of *ptr:  1
  Value of *arr:  1
  Legal: ptr = ptr + 1  Illegal: arr = arr + 1
  Legal: Asgn to ptr  Illegal: Asgn to arr
  ```

Arrays (4/6)

• Consequences:
  
  • arr is an array variable but looks like a pointer in many respects (though not all)
  
  • arr[0] is the same as *arr
  
  • arr[2] is the same as *(arr+2)
  
  • We can use pointer arithmetic to access arrays more conveniently.

• Declared arrays are only allocated while the scope is valid
  
  ```
  int *foo() {
    int arr[32]; ...
    return arr;
  } is incorrect
  ```

Arrays (5/6)

• 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, arr[10];
    for(i = 0; i < 10; i++){ ... }
    ```
  
  • Right
    ```
    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...)
Administrivia

- Read K&R 6 by the next lecture
- Lab problems
  - Labs should be fixed by Thursday.
  - You can get Tuesdays lab checked off Thursday. Everyone gets 1 bonus point (for first hour checkoff).
- Homework expectations
  - Readers don’t have time to fix your programs. They must run on lab machines.
  - Code that doesn’t compile or fails all of the autograder tests ⇒ 0
- C Help session tonight! 7pm, 306 Soda
- HW1 due Friday!

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.

Pointer Arithmetic (1/2)

- 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 (2/2)

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

C Strings

- A string in C is just an array of characters.
  ```c
  char str[] = "abc";
  ```
- How do you tell how long a string is?
  - Last character is followed by null terminator character, denote ‘\0’
  ```c
  int strlen(char s[])
  { int n = 0;
    while (s[n] != ‘\0’) n++;
    return n;
  }
  ```
Peer Instruction:

```c
int main(void){
    int *p = &A[0];
    printf("%u %d %d %d %d\n", A[0], A[1], *p, *p, A[0]);
    printf("%u %d %d %d %d\n", A[0], A[1], *p, *p, A[0]);
    printf("%u %d %d %d %d\n", A[0], A[1], *p, *p, A[0]);
    printf("%u %d %d %d %d\n", A[0], A[1], *p, *p, A[0]);
    if (the first printf outputs 100 5 5 10, what will the
    other two printf output?)
        a) 101 11 11
        b) 104 11 11
        c) 101 <3-others>
        d) 104 <3-others>
        e) One of the two prints causes an ERROR
}
```

C string.h Standard Functions

- `int strlen(char *string);`
  - compute the length of string.
- `int strcmp(char *s1, char *s2);`
  - return 0 if s1 and s2 are identical (how is this different from `s1 == s2`?)
- `char *strcpy(char *dst, const *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.
- But these functions aren't very safe:
  - What if no null terminator?

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 ensure there is enough space to store the full string!
  - What if you don’t know ahead of time how big your string will be? Tomorrow!
  - Buffer overrun security holes!

C string.h SAFE(R) Functions

- `int strlen(char *string, int max_len);`
  - compute the length of string. Not standard.
- `int strcmp(char *s1, char *s2, int max_len);`
  - return 0 if s1 and s2 are identical (how is this different from `s1 == s2`?). Standard
- `char *strncpy(char *dst, char *src, int max_len);`
  - 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. Standard
- These functions are much safer, and recommended! See K&R Pg 249 for details

C gives you a lot of extra rope but be careful not to hang yourself with it!

“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
- (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!”
Reference slides

You ARE responsible for the material on these slides (they’re just taken from the reading anyway); we’ve moved them to the end and off-stage to give more breathing room to lecture!

Arrays (one elt past array must be valid)

• 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

Pointers & Allocation (1/2)

• After declaring a pointer:

```
int *ptr;
```

• ptr doesn’t actually point to anything yet (it actually points somewhere - but don’t know where!). We can either:

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

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.

```
```

Pointer Arithmetic

• 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 to Copy memory

• 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++;
    }
}
```

• Note we had to pass size (n) to copy
### Pointer Arithmetic Summary

- **x = *(p+1)?**  
  \[ \Rightarrow x = *_{(p+1)}; \]

- **x = *p+1?**  
  \[ \Rightarrow x = (*p) + 1; \]

- **x = (*p)++?**  
  \[ \Rightarrow x = *_{p} ; \quad p = *_{p} + 1; \]

- **x = *p++? \quad (+p++) \quad (*p)++? \quad (p++)?**
  \[ \Rightarrow x = *_{p} ; \quad p = p + 1; \]

- **x = ++p?**  
  \[ \Rightarrow p = p + 1 ; \quad x = *_{p} ; \]

**Lesson**

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

---

### Arrays vs. Pointers

- **An array name is a read-only pointer to the 0th 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.**

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

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

Could be written:
```
while (*s)
```