Address vs. Value

- Consider memory to be a single huge array
  - Each cell of the array has an address associated with it
  - Each cell also stores some value
  - Do you think they use signed or unsigned numbers? Negative address?!
- Don’t confuse the address referring to a memory location with the value stored there

Pointers

- An address refers to a particular memory location; e.g., it points to a memory location
- Pointer: A variable that contains the address of a variable

Creating and Using Pointers

- How to create a pointer:
  - & operator: get address of a variable
  - * operator: get value at address in \( x \) to \( z \)
    - \& p \( \text{ declaration is to indicate that } p \text{ is going to be a pointer, and in the } \) printf to get the value pointed to by \( p \).

```c
int *p, x;

x = 3;
p = &x;

*p = 5;

printf("\%d\n", *p);
```
Using Pointer for Writes

- How to change a variable pointed to?
  - Use the dereference operator * on left of
    assignment operator

  ![Pointer for Writes](image)

  \[ *p = 5; \]

Pointers and Parameter Passing

- Java and C pass parameters “by value”
  - Procedure/function/method gets a copy of the
    parameter, so changing the copy cannot change the
    original

  ```
  void add_one (int x) {
    x = x + 1;
  }
  int y = 3;
  add_one(y);
  y remains equal to 3
  ```

- How can we get a function to change the value held in a variable?

  ```
  void add_one (int *p) {
    *p = *p + 1;
  }
  int y = 3;
  add_one(&y);
  y is now equal to 4
  ```

Pointers and Structures

- Types of Pointers
  - Pointers are used to point to any kind of data
    (int, char, a struct, a pointer, etc.)
  - Normally a pointer only points to one type
    (int, char, a struct, etc.).
    - void * is a type that can point to anything
      (generic pointer)
    - Use void * sparingly to help avoid program bugs,
      and security issues, and other bad things!

- Pointers and Parameter Passing
  - Declaring a pointer just allocates space to hold the
    pointer – does not allocate thing being pointed to!
  - Local variables in C are not initialized, they may
    contain anything (aka “garbage”)
  - What does the following code do?

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

  Note, C structure assignment is not a “deep copy”. All members are copied, but not
  things pointed to by members.

More C Pointer Dangers

- Pointers and Structures
  - typedef struct {
    /* dot notation */
    int x;
    int y;
  } Point;
  
  / * arrow notation */
  Point p1;
  int h = paddr->x;
  Point p2;
  int h = (*paddr).x;
  Point *paddr;
  /
  *structure assignment*/
  p2 = p1;

  typedef struct {
    /* dot notation */
    int x;
    int y;
  } Point;
  
  / * arrow notation */
  Point p1;
  int h = paddr->x;
  Point p2;
  int h = (*paddr).x;
  Point *paddr;
  /
  *structure assignment*/
  p2 = p1;

  Note, C structure assignment is not a “deep copy”. All members are copied, but not
  things pointed to by members.
Pointers in C

• Why use pointers?
  – If we want to pass a large struct or array, it's easier / faster / etc. to pass a pointer than the whole thing
  – Want to modify an object, not just pass its value
  – In general, pointers allow cleaner, more compact code

• So what are the drawbacks?
  – Pointers are probably the single largest source of bugs in C, so be careful anytime you deal with them
  • Most problematic with dynamic memory management—coming up next lecture
  • Dangling references and memory leaks

Why Pointers in C?

• At time C was invented (early 1970s), compilers often didn’t produce efficient code
  – Computers 25,000 times faster today, compilers better
• C designed to let programmer say what they want code to do without compiler getting in way
  – Even give compiler hints which registers to use!
• Today, many applications attain acceptable performance using higher-level languages without pointers
• Low-level system code still needs low-level access via pointers, hence continued popularity of C

Clickers/Peer Instruction Time

```c
void foo(int *x, int *y)
{
    int t;
    if ( *x > *y ) { t = *y; *y = *x; *x = t; }
    int a=3, b=2, c=1;
    foo(&a, &b);
    foo(&b, &c);
    printf("a=%d b=%d c=%d\n", a, b, c);
}
```

A: a=3 b=2 c=1
B: a=1 b=2 c=3
Result is: C: a=1 b=3 c=2
D: a=3 b=3 c=3
E: a=1 b=1 c=1

C Arrays

• Declaration:
  ```c
  int ar[2];
  ```
declares a 2-element integer array: just a block of memory

  ```c
  int ar[] = {795, 635};
  ```
declares and initializes a 2-element integer array returns the numth element

C Strings

• String in C is just an array of characters
  ```c
  char string[] = "abc";
  ```
• How do you tell how long a string is?
  – Last character is followed by a 0 byte (aka "null terminator")
  ```c
  int strlen(char s[])
  {
      int n = 0;
      while (s[n] != 0) n++;
      return n;
  }
  ```

Administrivia

• We can accommodate all those on the wait list, but you have to enroll in a lab section with space!
  – Lab section is important, but you can attend different discussion section
  – Enroll into lab with space, and try to swap with someone later
• HW0 due 11:59:59pm Sunday 2/1
  – Right after the Superbowl...
• Midterm-II now Thursday April 9 in class
Array Name / Pointer Duality

- **Key Concept**: Array variable is a “pointer” to the first (0th) element
- So, array variables almost identical to pointers
  - `char *string` and `char string[]` are nearly identical declarations
  - Differ in subtle ways: incrementing, declaration of filled arrays
- **Consequences**:
  - `ar` is an array variable, but looks like a pointer
  - `ar[0]` is the same as `*ar`
  - Can use pointer arithmetic to conveniently access arrays

Pointers to a Pointer

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

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

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

C Arrays are Very Primitive

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

Use Defined Constants

- Array size `n`; want to access from 0 to `n-1`, so you should use counter AND utilize a variable for declaration & incrementation
  - Bad pattern
    ```
    int i, ar[10];
    for(i = 0; i < 10; i++) { ... }
    ```
  - Better pattern
    ```
    const int ARRAY_SIZE = 10;
    int i, a[ARRAY_SIZE];
    for(i = 0; i < ARRAY_SIZE; i++) { ... }
    ```
- Accessing elements:
  - `ar[num]`
  - **SINGLE SOURCE OF TRUTH**
    - You’re utilizing indirect and avoiding maintaining two copies of the number 10
    - DRY: “Don’t Repeat Yourself”

Pointing to Different Size Objects

- Modern machines are "byte-addressable"
  - Hardware’s memory composed of 8-bit storage cells, each has a unique address
- A C pointer is just abstracted memory address
- Type declaration tells compiler how many bytes to fetch on each access through pointer
  - E.g., 32-bit integer stored in 4 consecutive 8-bit bytes

```
short *y
int *x
char *s
```
sizeof() operator

- sizeof(type) returns number of bytes in object
  - But number of bits in a byte is not standardized
    - In olden times, when dragons roamed the earth, bytes could be 5, 6, 7, 9 bits long
- By definition, sizeof(char)==1
- Can take sizeof(arr), or sizeof(structtype)
- We'll see more of sizeof when we look at dynamic memory management

Arrays and Pointers

- Array = pointer to the initial (0th) array element
  
  \[ a[i] = *(a+i) \]

- An array is passed to a function as a pointer
  - The array size is lost!

- Usually bad style to interchange arrays and pointers
  - Avoid pointer arithmetic!

Arrays and Pointers

Passing arrays:

- Really int *array
- Must explicitly pass the size

Arrays and Pointers

int i;
int array[10];
for (i = 0; i < 10; i+1)
{ array[i] = …; }

int *p;
int array[10];
for (p = array; p < array[10]; p++)
{ *p = …; }

These code sequences have the same effect!

Clickers/Peer Instruction Time

int x[5] = { 2, 4, 6, 8, 10 }; int *p = x;
int **pp = &p;
(**pp)++;
printf("%d\n", *p);

Result is:
A: 2
B: 3
C: 4
D: 5
E: None of the above
In the News (1/23/2015):
Google Exposing Apple Security Bugs

- Google security published details of three bugs in Apple OS X (90 days after privately notifying Apple)
  - One network stack problem fixed in Yosemite, all in next beta
  - One is dereferencing a null pointer!
  - One is zeroing wrong part of memory!
- Separately, Google announces it won't patch WebKit vulnerability affecting Android 4.3 and below (only about 930 million active users)

---

Concise `strlen()`

```c
int strlen(char *s)
{
    char *p = s;
    while (*p++)
        /* Null body of while */
    return (p - s - 1);
}
```

What happens if there is no zero character at end of string?

---

Point past end of array?

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

---

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 illegal since makes no sense:
- adding two pointers
- multiplying pointers
- subtract pointer from integer

---

Arguments in `main()`

- To get arguments to the main function, use:
  ```
  int main(int argc, char *argv[])
  ```
- What does this mean?
  ```
  argc contains the number of strings on the command line (the executable counts as one, plus one for each argument). Here argc is 2:
  ```
  ```
  unix% sort myfile
  ```
  ```
  argv is a pointer to an array containing the arguments as strings
```

---

Example

- ```
  foo hello 87
  ```
- ```
  argc = 3 /* number arguments */
  ```
  ```
  argv[0] = "foo",
  ```
  ```
  argv[1] = "hello",
  ```
  ```
  argv[2] = "87"
  ```
  ```
  - Array of pointers to strings
```
And In Conclusion, ...

- Pointers are abstraction of machine memory addresses
- Pointer variables are held in memory, and pointer values are just numbers that can be manipulated by software
- In C, close relationship between array names and pointers
- Pointers know the type of the object they point to (except void*)
- Pointers are powerful but potentially dangerous