Arrays and Memory Management
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
- But we actually want “word alignment”
  - Some processors will not allow you to address 32b values without being on 4 byte boundaries
  - Others will just be very slow if you try to access “unaligned” memory.

![Memory Diagram]

- 16-bit short stored in two bytes
- 32-bit integer stored in four bytes
- 8-bit character stored in one byte
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 Standard C99 definition, **sizeof(char)==1**
- C does not play well with Unicode (unlike Python), so no `char c = '💩'`
- Can take **sizeof(arg)**, or **sizeof(structtype)**
- We’ll see more of sizeof when we look at dynamic memory management
  - Remember, C **does not specify sizeof(int)**, so you need to ask it whenever you need to find this out!
## Pointer Arithmetic

### pointer + number  \[\text{pointer} - \text{number}\]

**e.g., pointer + 1** adds 1 something to a pointer

<table>
<thead>
<tr>
<th>char   *p;</th>
<th>int   *p;</th>
</tr>
</thead>
<tbody>
<tr>
<td>char    a;</td>
<td>int    a;</td>
</tr>
<tr>
<td>char    b;</td>
<td>int    b;</td>
</tr>
</tbody>
</table>

| p = &a; | p = &a; |
| p += 1; | p += 1; |

In each, \(p\) now points to \(b\)

(Assuming compiler doesn’t reorder variables in memory.

*Never code like this!!!!*

Adds 1\(\times\)\text{sizeof(char)} \>

\[\text{to the memory address}\]

| Adds 1\(\times\)\text{sizeof(int)} \>

\[\text{to the memory address}\]

**Pointer arithmetic should be used cautiously**
Valid Pointer Arithmetic

- Add an integer to a pointer.
- Subtracting an integer from a pointer
- Subtract 2 pointers (in the same array & of the same type)
- 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
C Arrays

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

  ```
  int ar[] = {795, 635};
  ```
declares and initializes a 2-element integer array
Array Name / Pointer Duality

• **Key Concept:** Array variable is simply 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 works like a pointer
  • `ar[0]` is the same as `*ar`
  • `ar[2]` is the same as `*(ar+2)`
  • Can use pointer arithmetic to access arrays
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)
  - But also “fun” to exploit:
    - “Stack overflow exploit”, maliciously write off the end of an array on the stack
    - “Heap overflow exploit”, maliciously write off the end of an array on the heap
C Strings

• 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
    (aka “null terminator”):
    written as 0 (the number) or '\0'
    as a character

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

- **SINGLE SOURCE OF TRUTH**
  - You’re utilizing indirection and avoiding maintaining two copies of the number 10
  - DRY: “Don’t Repeat Yourself”
  - And don’t forget the < rather than <=:
    - When Nick took 60c, he lost a day to a “segfault in a malloc called by printf on large inputs”:
      - Had a <= rather than a < in a single array initialization!
Changing a Pointer Argument?

- What if want function to change a pointer?
- What gets printed?

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

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

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

```c
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);
```

```plaintext
void inc_ptr(int **h)          *q = 60
{                             
    *h = *h + 1;              
}                             

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

```
50 60 70
```
When Arrays Go Bad: Heartbleed

- In TLS encryption, messages have a length…
  - And get copied into memory before being processed
- One message was “Echo Me back the following data, its this long…”
  - But the (different) echo length wasn’t checked to make sure it wasn’t too big…

```plaintext
POST / HTTP/1.0
Host: www.mydomain.com
Cookie: login=1
User-Agent: Mozilla...
```

- So you send a small request that says “read back a lot of data”
  - And thus get web requests with auth cookies and other bits of data from random bits of memory…
Arrays and Pointers

- Array $\approx$ pointer to the initial element
  - $a[i] \equiv *(a+i)$
- 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!
    - Especially avoid things like $ar++$;

Passing arrays:

```c
int foo(int array[],
        unsigned int size)
{
    ... array[size - 1] ...
}

int main(void)
{
    int a[10], b[5];
    ... foo(a, 10)... foo(b, 5) ...
}
```

Really int *array

Must explicitly pass the size
Arrays and Pointers

```c
int foo(int array[], unsigned int size) {
    ...
    printf("%d
", sizeof(array));
}

int main(void) {
    int a[10], b[5];
    ...
    foo(a, 10) ... foo(b, 5) ...
    printf("%d
", sizeof(a));
}
```

What does this print? 4

... because `array` is really a pointer (and a pointer is architecture dependent, but likely to be 4 or 8 on modern 32-64 bit machines!)

What does this print? 40
Arrays and Pointers

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

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

These code sequences have the same effect!

But the former is much more readable:
Especially don't want to see code like \texttt{ar++}
Clickers/Peer Instruction Time

```c
int x[] = { 2, 4, 6, 8, 10 };
int *p = x;
int **pp = &p;
(*pp)++;
(*(*pp))++;
printf("%d\n", *p);  // Result is:
A: 2
B: 3
C: 4
D: 5
E: None of the above
```
Clickers/Peer Instruction Time

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

Result is:

A: 2
B: 3
C: 4
D: 5
E: None of the above

P points to the start of X (2)
PP points to P
Increments P point to 2\textsuperscript{nd} element (4)
Increments 2\textsuperscript{nd} element by 1 (5)
Administrivia

- hw0 - edX is due on 2/13
- hw0 mini bio is due in second section
- Project 1 and the first lab will be out RSN
  - "Real Soon Now"
- MT1 will be Friday, 2/24 from 7 PM - 9 PM
- MT2 will be Thursday, 4/06 from 7 PM - 9 PM
  - Email the head Rebecca and Stephen for conflicts
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?
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
    • Since it is an array of pointers to character arrays
    • Sometimes written as char **argv
Example

- `foo hello 87 "bar baz"
- `argc = 4 /* number arguments */`
- `argv[0] = "foo",
  argv[1] = "hello",
  argv[2] = "87",
  argv[3] = "bar baz",
- Array of pointers to strings
C Memory Management

• How does the C compiler determine where to put all the variables in machine’s memory?
• How to create dynamically sized objects?
• To simplify discussion, we assume one program runs at a time, with access to all of memory.
• Later, we’ll discuss virtual memory, which lets multiple programs all run at same time, each thinking they own all of memory.
C Memory Management

- Program’s address space contains 4 regions:
  - **stack**: local variables inside functions, grows downward
  - **heap**: space requested for dynamic data via `malloc()` resizes dynamically, grows upward
  - **static data**: variables declared outside functions, does not grow or shrink. Loaded when program starts, can be modified.
  - **code**: loaded when program starts, does not change
Where are Variables Allocated?

- If declared outside a function, allocated in “static” storage
- If declared inside function, allocated on the “stack” and freed when function returns
  - `main()` is treated like a function
- For both of these types of memory, the management is automatic:
  - You don’t need to worry about deallocating when you are no longer using them

```c
int myGlobal;
main() {
    int myTemp;
}
```
The Stack

- Every time a function is called, a new frame is allocated on the stack
- Stack frame includes:
  - Return address (who called me?)
  - Arguments
  - Space for local variables
- Stack frames use contiguous blocks of memory; stack pointer indicates start of stack frame
- When function ends, stack pointer moves up; frees memory for future stack frames
- We’ll cover details later for MIPS processor

```plaintext
fooA() { fooB(); }
fooB() { fooC(); }
fooC() { fooD(); }
```
Stack Animation

• Last In, First Out (LIFO) data structure

```c
main ()
{ a(0);
 }
void a (int m)
{ b(1);
}
void b (int n)
{ c(2);
}
void c (int o)
{ d(3);
}
```
Managing the Heap

C supports functions for heap management:

- `malloc()` allocate a block of uninitialized memory
- `calloc()` allocate a block of zeroed memory
- `free()` free previously allocated block of memory
- `realloc()` change size of previously allocated block
  - `careful – it might move!`
  - And it *will not update other pointers pointing to the same block of memory*
Malloc()

- `void *malloc(size_t n)`:
  - Allocate a block of uninitialized memory
  - NOTE: Subsequent calls probably will not yield adjacent blocks
  - `n` is an integer, indicating size of requested memory block in bytes
  - `size_t` is an unsigned integer type big enough to "count" memory bytes
  - Returns `void*` pointer to block; `NULL` return indicates no more memory (check for it!)
  - Additional control information (including size) stored in the heap for each allocated block.

- Examples:
  - `int *ip;`  
    `ip = (int *) malloc(sizeof(int));`
  - `typedef struct { … } TreeNode;`  
    `TreeNode *tp = (TreeNode *) malloc(sizeof(TreeNode));`
  - `sizeof` returns size of given type in bytes, **necessary if you want portable code!**
And then free()

- **void free(void *p):**
  - p is a pointer containing the address originally returned by malloc()

- **Examples:**
  - int *ip;
    ip = (int *) malloc(sizeof(int));
    ...
    free((void*) ip); /* Can you free(ip) after ip++ ? */
  - typedef struct {… } TreeNode;
    TreeNode *tp = (TreeNode *) malloc(sizeof(TreeNode));
    ...
    free((void *) tp);

- When you free memory, you must be sure that you pass the original address returned from malloc() to free(); Otherwise, crash (or worse)!
Using Dynamic Memory

typedef struct node {
    int key;
    struct node *left; struct node *right;
} Node;

Node *root = NULL;

Node *create_node(int key, Node *left, Node *right){
    Node *np;
    if(!(np = (Node*) malloc(sizeof(Node)))){
        printf("Memory exhausted!\n");
        exit(1);
    } else{
        np->key = key;
        np->left = left;
        np->right = right;
        return np;
    }
}

void insert(int key, Node **tree){
    if (((*tree) == NULL){
        (*tree) = create_node(key, NULL, NULL);
    } else if (key <= (*tree)->key){
        insert(key, &((*tree)->left));
    } else{
        insert(key, &((*tree)->right));
    }
}

int main(){
    insert(10, &root);
    insert(16, &root);
    insert(5, &root);
    insert(11, &root);
    return 0;
}
Observations

- Code, Static storage are easy: they never grow or shrink
- Stack space is relatively easy: stack frames are created and destroyed in last-in, first-out (LIFO) order
- Managing the heap is tricky: memory can be allocated / deallocated at any time
  - If you forget to deallocate memory: “Memory Leak”
    - Your program will eventually run out of memory
  - If you call free twice on the same memory: “Double Free”
    - Possible crash or exploitable vulnerability
  - If you use data after calling free: “Use after free”
    - Possible crash or exploitable vulnerability
And In Conclusion, ...

- C has three main memory segments in which to allocate data:
  - Static Data: Variables outside functions
  - Stack: Variables local to function
  - Heap: Objects explicitly malloc-ed/free-d.
- Heap data is biggest source of bugs in C code