

# POINTERS

**CS 580U Fall 2017**

# ARRAY ADDRESSES

- The memory address of the first element of an array is given the name of the array.
  - for example, given: `int list[] = {5,6,7,8}`  
*list* is the memory address of '5'
- When memory is allocated for the array cells, the starting address is fixed
  - To ensure that this address is not changed array names may not be used as variables on the left of an assignment statement

# CLASSWORK

## Printing Addresses

```
#include <stdio.h>

int main(){
    char str[1];
    printf("%p\n", str);
    printf("%p\n", str+1);
    int num[1];
    printf("%p\n", num);
    printf("%p\n", num+1);
    double db[1];
    printf("%p\n", db);
    printf("%p\n", db+1);
}
```

---

# POINTERS

- A variable that has the memory address of another variable
  - Sometimes called references or alias
- One of the most powerful features of C
  - Allows you to write very efficient code
- Pointers do not hold data
  - Pointers hold a number that corresponds to an address in memory of some data

# WHY POINTERS?

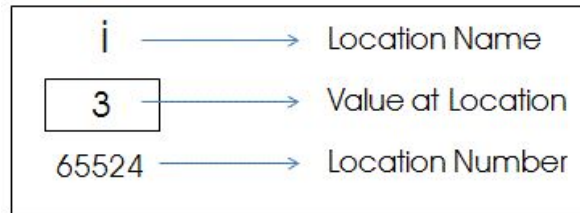
- They allow you to refer to large data structures in a compact way
- They make memory usage and management easier
- They make it possible to get new memory dynamically as your program is running

# MANAGING MEMORY

- Recall, C is pass by value
  - What if we want to change a value in different scope?
- Example
  - We write a function to keep track of various info since we have been castaway on a deserted island
    - We have an `int num_days = 0, food_left = 50;`
  - We want to increment the `num_days` in the function.
    - We could return the new number of days, but then `food_left` does not get decremented
  - We need to access the original value, not a copy

# DEFINING A POINTER

- Uses the `*` operator
  - Definition: `<type> * <var>;`
  - Example: `int * num_ptr`
- What does a pointer contain?
  - just a number
  - You can treat it just like an integer



# A POINTER IN MEMORY

- Computer memory is just a very large array
  - Memory is addressed in bytes
  - byte 1 address = 0, byte 2 address = 1...
- Every byte has an address
  - Byte addressable

Ox0	Ox1	Ox2	Ox3	Ox4	Ox5	Ox6	Ox7	Ox8
-----	-----	-----	-----	-----	-----	-----	-----	-----



# POINTER'S FLEXIBLE SYNTAX

- The ‘\*’ that designates the pointer type may have a space between the data type (int, char, etc) and the variable name
  - It does not have to
- Example: `char * x == char* x == char *x;`
- **IMPORTANT:** The pointer is a type
  - `char != char *` ←these are different types

# BEST PRACTICE

- What does the following declare?
  - `int * x, y;`
- For clarity, best practice is to keep the pointer symbol next to the variable
  - `int *x, *y;`


# NULL

- NULL is essentially 0
- NULL is the universal value for ‘this variable has not been initialized yet’
  - Pointer variables should ALWAYS be initialized to NULL if not initialized to a value
    - Why? What do they contain if you don't?
- Don't confuse a void pointer and a NULL point
  - Void pointer doesn't have a type

# CLASSWORK: MYSIZEOF

# DEREFERENCING

Address	Ox0	Ox1	Ox2	Ox3	Ox4	Ox5	Ox6	Ox7	Ox8
Value				Ox7				'A'	



- “Any problem in computer science can be solved by more indirection”
- A pointer is a memory address with a restricted type, also known as a **reference**

# DEREFERENCE POINTERS

- **Dereference** the pointer with `*`
  - `int x = *ptr`
  - Same symbol as pointer declaration, not the same operation
    - When you follow GPS directions to an address, you are dereferencing the address
- Dereferencing jumps from the address in the pointer to the data stored at that address

# DEREFERENCE

```
char some_string[] = "string";  
char *letter_ptr = some_string;  
//to access some_string[0]  
char letter_1 = *letter_ptr;  
//to access some_string[1]  
char letter_2 = letter_ptr[1];
```

- We can treat pointers just like arrays
  - But they aren't the same thing (more on this later...)

# ORDER OF OPERATIONS

- Dereference example: `int num = *num_ptr;`
- Used in an expression
  - `int num = *num_ptr+2; //this may not give the expected results`
- Dereference has the higher priority
  - `int num = *(num_ptr + 2); //must use parenthesis`



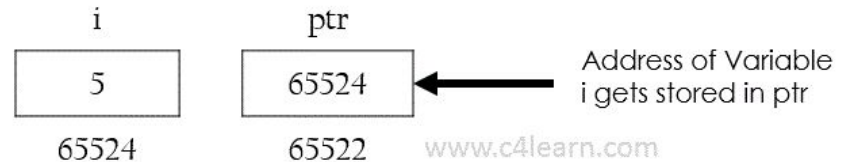
# REFERENCE OPERATOR

- Get the memory address of data with the reference operator: &
- Example:
  - `char c = 'c';`  
`char * c_ptr = &c;`
- Type should match
  - You can assign the address of an int to a char pointer, but what happens?



# & - MEMORY ADDRESS

- Grab the memory address of any variable with the reference operator - &
- The value must be stored in memory, not a literal
  - &var gives me the memory address of var
  - &10 not possible, 10 is not 'stored', it is a literal



# POINTER PARAMETERS

- Declare pointer parameters like other types

- Example:

```
int foo(char * string, int * list);
```

- functionally the same as:

```
int foo(char string[], int list[]);
```

- *All arrays are passed as pointers*

# PASS BY VALUE

- C only has pass by value
  - But you can pass the value of a memory address  
\*tricky\*
- We say all types passed by value except pointers and arrays
  - even though they are too
  - Pointer passes by value, but values pointed to are always referenced

# PASS BY REFERENCE

- Pass the memory address of the data
- Pass by reference using & and \*

- Example:

```
int x = 1;
passByReference(&x);
/.../
int passByReference(int * x) {
    (*x) = (*x) + 2;
}
```

# CLASSWORK

## Pointers

4, 4

3, 4

```
void swap(int *p1, int *p2){  
    int temp = *p1;  
    *p1 = * p2;  
    *p2 = temp;  
}
```

# POINTER ARITHMETIC

- Because pointer's are basically just numbers, you can do basic arithmetic with them
  - add and subtract only
- Math works different with pointers
  - `int * ptr = 2;`  
`int * new_ptr = ptr + 2;`
  - *What do you think new\_ptr would equal?*

# MATH WITH POINTERS

- Pointer math always takes into account the type
- **Formula:**  $\text{<address>} + (\text{<val>} * \text{<type size>})$ 
  - type size is implicit/automatic
- Example
  - `int * ptr = 2;`  
`new_ptr = ptr + (2 * sizeof(int))`



# UNARY OPERATOR

- Preoperations

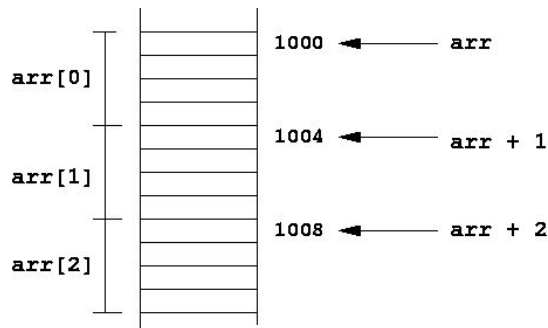
- ++i or --i
- (Inc/Dec)rements first, then performs operation

- Postoperations

- i++ or i--
- performs operation, then (Inc/Dec)rements

# POINTER INCREMENT

- Remember, pointers have types to support pointer arithmetic
- Pointers add according to their types
- Example:
  - `++int_ptr`: adds by `sizeof(int)`
  - `char_ptr++`: adds by `sizeof(char)`



# CLASSWORK

## Pointer Arithmetic

```
#include <stdio.h>

void prints(char * str){
    char * c;
    for(c = str; *(c+1) != '\0'; c++){
        printf("%c -", *c);
        printf("%c", *c);
    }
}

void reverse(int * arr, int size){
    int * end;
    for(end = arr+size-1; arr < end; arr++, end--){
        int temp = *arr;
        *arr = *end;
        *end = temp;
    }
}

int main(){
    int nums[] = {1,2,3,4,5,6,7};
    char string[] = "Hello";
    prints(string);
    printf("\n");
    reverse(nums, sizeof(nums)/sizeof(int));
    int * arr = nums;
    for(; arr < nums+sizeof(nums)/sizeof(int); arr++){
        printf("%d", *arr);
    }
    printf("\n");
    return 0;
}
```

# POINTERS TO POINTERS

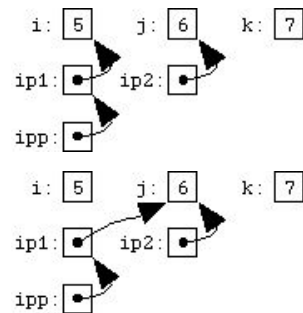
- Since we can have pointers to int, and in fact pointers to any type in C, it shouldn't come as too much of a surprise that we can have pointers to other pointers.
  - We'll now have to distinguish between the pointer, what it points to, and what the pointer that it points to points to.
    - And, of course, we might also end up with pointers to pointers to pointers, or pointers to pointers to pointers to pointers...

# DOUBLE POINTERS

- So, what if I want to keep any array of 5 strings?
  - Since strings are just pointers to an array of characters
    - `char * strings[5]`
- But arrays are just pointers to a starting memory address
  - `char ** strings;`
    - where the two asterisks indicate that two levels of pointers are involved.

# DOUBLE POINTERS

- Suppose we have:
  - `int ** ipp;`
  - `int i = 5, j = 6; k = 7;`
  - `int *ip1 = &i, *ip2 = &j;`
- Now we can set:
  - `ipp = &ip1;`
- If we say:
  - `*ipp = ip2;`



# USING POINTERS TO MANAGE MEMORY

# SEGMENTATION

- When your executable runs, the Operating System reserves a segment of memory available for the program's execution
  - When your program tries to access memory outside of this allotted segment, you create a 'segmentation fault'
  - All resources used by this segment are released when the segment is released

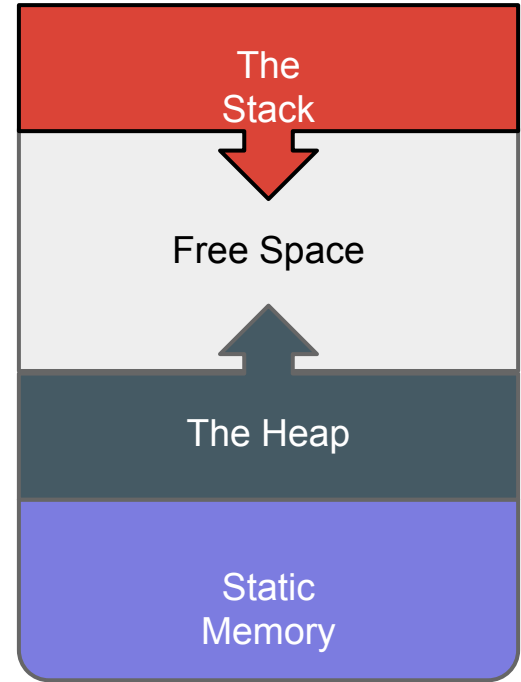


# HOW TO BUILD A SEGMENT

- Your segment is further divided into different kinds of memory
  - Why? Because some information is known at compile time and some won't be known until runtime
- The memory-segment is divided into sub-segments
  - We need space for our program to run
  - We also need free memory to allocate and deallocate

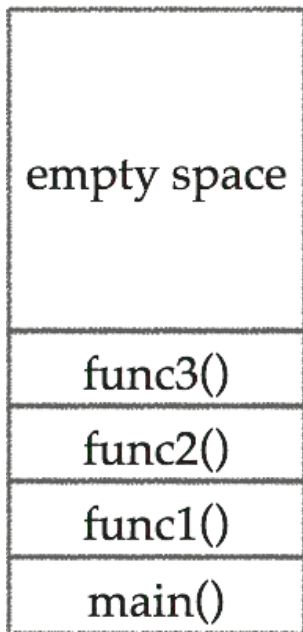
# MEMORY ORGANIZATION

- 3 Primary Memory Sub-Segments
  - Stack - running program
  - Heap - free space
  - Static - compile time memory
- Stack and Heap are Dynamic Memory
  - They are allocated during runtime



# THE STACK

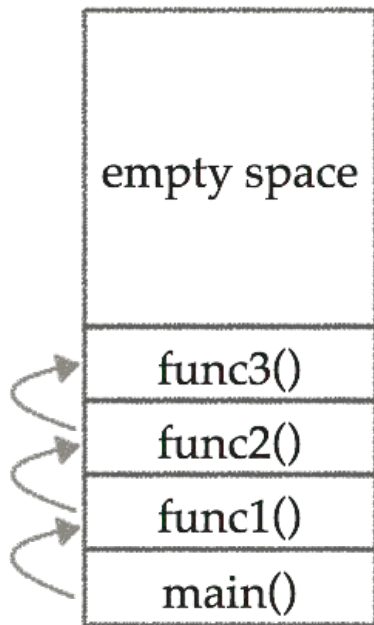
Function calls



- The stack grows as functions allocate local variables
- Automatically allocated and freed
- The stack has size limits
  - OS Dependant
- Local variables only exist while the function that created them is running
  - scope

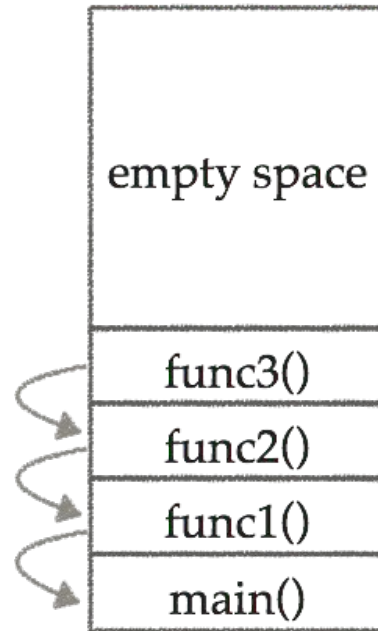
# STACK MOVEMENT

Function calls



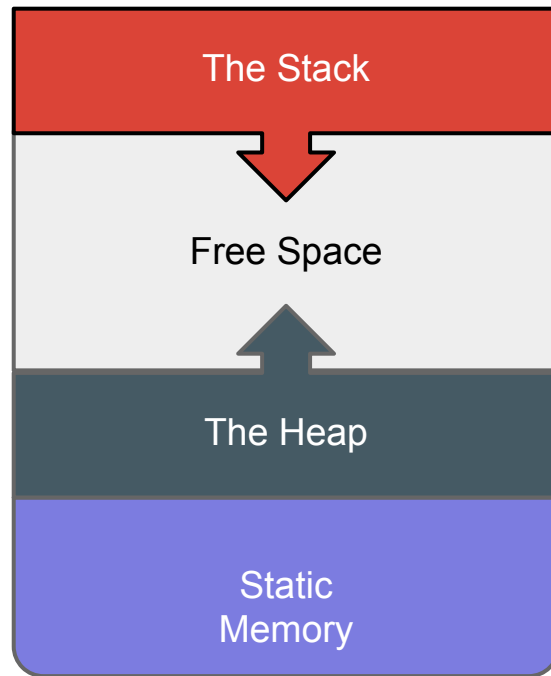
- Function calls add a frame to the stack
- Function returns remove frames from the stack
- Once a frame is unwound, the data is lost
  - *or is it?*

Function Returns



# THE HEAP

- Writable memory can be allocated and deallocated by the programmer (dynamically)
- Memory does not automatically free from the heap for the life of the program
  - Pointers are essential



# STACK VS HEAP VS GLOBAL

	Stack	Heap	Static
Memory Type	Runtime	Runtime	Compile Time
How is it created?	Automatically Allocated	Explicitly Allocated	Statically Allocated
How is it freed?	Automatically Freed	Explicitly Freed	Life of the Program

# VOID \*

- typeless or universal pointer
- cannot be dereferenced
  - must be cast into a type before dereferencing
    - why?
- cannot be used in pointer arithmetic
  - why?
- How to know what to cast it?
  - This is the programmer's responsibility

# VOID \* EXAMPLES

- As variables
  - `void * v_ptr = &foo;`
    - remember, pointer addresses are just integers
- On most compilers, `void *` automatically become the pointer type assign
  - Example: `int * x = returnVoidPtr();`
    - This automatically becomes an `int *`

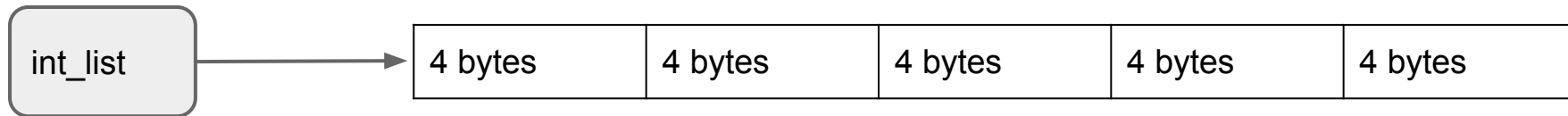


# MALLOC()

- malloc allows the programmer to dynamically allocate memory that lives outside the life of stack
- `void* malloc (size_t size);`
  - `size_t` is a library typedef for an unsigned int
  - `size_t` is always in bytes
- Examples
  - `int * int_ptr = malloc(4);`
  - `int * int_ptr = malloc(sizeof(int));`
  - `int * my_ptr = malloc(sizeof(MyStruct));`

# ALLOCATING SPACE FOR ARRAYS

- Allocate space for arrays by multiplying the number of items by the size of the type
- Example
  - `int * int_list = malloc(sizeof(int) * 5);`
    - `sizeof(int) = 4`
    - `4*5 = 20`
  - `MyStruct * mystruct_list = malloc(sizeof(MyStruct) * 20);`



# WHERE IS THIS MEMORY?

- Dynamically allocated memory is put on the heap
  - The heap grows as more memory is allocated
- The amount of room for heap memory is finite
  - Though you'll probably never run into this limit unless you have a bad memory leak

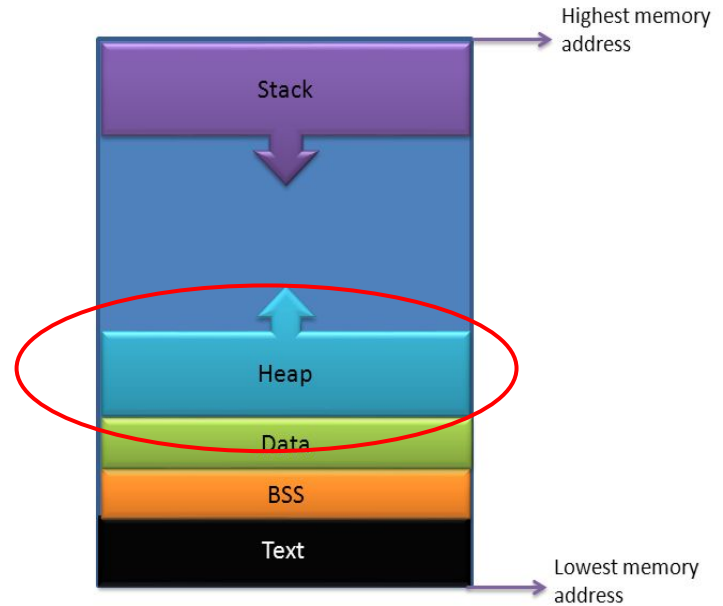


Figure : Process memory organization

# MALLOC() ONLY ALLOCATES

- malloc allocates and marks a chunk of memory in the heap as 'in use'
- malloc does not
  - clear or zero out memory (some OS's do)
    - whatever was there previously is still there (junk)
  - do **any** bounds checking to make sure you are only using that memory
    - You can write and read past the end of your allocated memory
  - guarantee the memory is secure
    - another pointer can point to the same memory and overwrite it

# MALLOC() ISN'T THE BOSS OF YOU...

What's wrong with this code?

```
int * ptr = malloc(sizeof (int));  
*ptr = 8;  
*(ptr + 1) = 5;  
printf("\n%d\n", ptr[1]);
```

...BUT MALLOC() WILL TAKE YOUR STUFF

```
int * ptr = malloc(sizeof (int));  
*ptr = 8;  
*(ptr + 1) = 5;  
printf("\n%d\n", ptr[1]);
```

Allocated by malloc()

Not allocated, but (probably) still accessible

000000000	000000000	000000000	00001000	000000000	000000000	000000000	00000101
-----------	-----------	-----------	----------	-----------	-----------	-----------	----------

# BUFFER OVERRUN

- malloc finds and allocates free space,
  - it doesn't limit what you can access within your segment
- As long as the memory is in your dynamic segment, you can access it with a pointer
- No guarantee that the OS won't overwrite memory not allocated by malloc

# VERIFY SUCCESSFUL ALLOCATION

- malloc returns NULL if there is not enough memory to allocate
- You can check malloc's return value for NULL
- Example:
  - ```
if((ptr = malloc(num_bytes)) == NULL){  
    exit(-1);  
}
```
  - Is this necessary?



# ADDITIONAL ALLOCATION CALLS

- `calloc()`
  - `void* calloc (size_t num, size_t size);`
  - Allocates a block of memory for an array of `num` elements, each of them `size` bytes long, and initializes all its bits to zero.
- `realloc()`
  - `void* realloc (void* ptr, size_t size);`
  - Changes the size of the memory block pointed to by `ptr`. The function may move the memory block to a new location (whose address is returned by the function).

# FREE()

- free releases memory marked as allocated by malloc
  - requires the address of memory allocated by malloc
  - memory is reclaimed for later use
- Syntax: `void free (void* ptr);`
- Example:
  - `char * str = malloc(sizeof(char)*6);`
  - ...
  - `free(str);`

# ALWAYS NULL FREED POINTERS



- Free does not change the pointer
  - Free does not 'clear' memory
  - The freed pointer still has the memory address of the freed memory
    - Dangling Pointer
- Always NULL freed pointers
  - `free(ptr);`  
`ptr = NULL;`
  - Calling `free()` on NULL is a no-op, not an error

CLASSWORK: SPACE