



# C Minicourse Day 2

Memory Management  
Functions



# Outline

- more on pointers
- malloc/free
- dynamic arrays
- functions
- main arguments
- pass by: value / pointer
- function pointers



# Pointers and Arrays

```
int main() {  
    int array[10];  
    int* pnter = array;  
    for(int i=0; i < 10; i++) {  
        printf("%d\n", pnter[i]);  
    }  
    return 0;  
}
```

- We can get a pointer to the beginning of an array using the name of the array variable without any brackets.
- From then on, we can index into the array using our pointer.



# Pointer Arithmetic

```
int main() {
    int array[10];
    int* pnter = NULL; //set pointer to NULL
    for(pnter = array; pnter < array + 10; pnter++) {
        printf("%d\n", *pnter); //dereference the pnter
    }
    return 0;
}
```

- We can “increment” a pointer, which has the effect of making it point to the next variable in a array.
- Instead of having an integer counter, we iterate through the array by moving the pointer itself.
- The pointer is initialized in the for loop to the start of the array. Terminate when we get the tenth index.



# void\*

- A pointer to nothing??
- NO!! A pointer to *anything*

```
int main() {  
    char c;  
    int i;  
    float f;  
    void* ptnr;  
    ptnr = &c; //OK  
    ptnr = &i; //OK  
    ptnr = &f; //OK  
    return 0;  
}
```

- Remember, all pointers are the same size (typically 32 or 64 bits) because they all store the same kind of memory address.



# when to pass by pointer

- When declaring a function, you can either pass by value or by reference.
- Factors to consider:
  - How much data do you have?
  - Do you “trust” the other functions that will be calling your function.
  - Can you handle the memory management complexity?



# Stack vs. Heap

- Review from cs31: Stack vs. Heap
  - Both are sources from which memory is allocated
  - Stack is automatic
    - Created when memory is “in scope”
    - Destroyed when memory is “out of scope”
  - Heap is manual
    - Created upon request
    - Destroyed upon request



# Two ways to get an `int`:

## ○ On the Stack:

```
int main() {
    int myInt; //declare an int on the stack
    myInt = 5; //set the memory to five
    return 0;
}
```

## ○ On the Heap:

```
int main() {
    int *myInt = malloc(sizeof(int));
                                //allocate mem. from heap
    *myInt = 5;                  //set the memory to five
    return 0;
}
```



# A closer look at `malloc`

```
int main() {  
    int *myInt = malloc(sizeof(int));  
    *myInt = 5;  
    return 0;  
}
```

- `malloc` short for “memory allocation”
  - Takes as a parameter the number of bytes to take from the heap
  - `sizeof(int)` conveniently tells us how many bytes are in an `int` (usually 4)
  - Returns a *pointer* to the memory that was just allocated



# ...but we forgot something!

- We requested memory but never released it!

```
int main() {
    int *myInt = malloc(sizeof(int));
    *myInt = 5;
    free(myInt); //use free() to release memory
    myInt = NULL;
    return 0;
}
```

- `free()` releases memory we aren't using anymore
  - Takes a *pointer* to the memory to be released
  - Must have been allocated with `malloc`
  - Should set pointer to `NULL` when done



# Don't do these things!

- `free()` memory on the stack

```
int main() {
    int myInt;
    free(&myInt); //very bad
    return 0;
}
```

- Lose track of `malloc()`'d memory

```
int main() {
    int *myInt = malloc(sizeof(int));
    myInt = 0; //how can you free it now?
    return 0;
}
```

- `free()` the same memory twice

```
int main() {
    int *A = malloc(sizeof(int));
    int *B = A;
    free(A);
    free(B); //very bad
    return 0;
}
```



# Dynamic arrays (1)

- Review: static arrays

```
int main() {  
    int array[5];           //static array of size 5  
    for(int i = 0; i < 5; i++) { //initialize the array to 0  
        array[i] = 0;  
    }  
    return 0;  
}
```

- **Problem:** Size determined at compile time. We must explicitly declare the exact size.



# Dynamic arrays (2)

- **Solution:** use `malloc()` to allocate enough space for the array at run-time

```
int main() {
    int n = rand() % 100;           //random number between 0 and 99
    int *array = malloc(n * sizeof(int)); //allocate array of size 'n'
    using malloc
    for(int i = 0; i < n; i++) {    //we can count up to (n-1) in our array
        array[i] = 0;
    }
    free(array);                   //make sure to free the array!
    array = NULL;
    return 0;
}
```



# Using malloc for “strings”

- Since “strings” in C are just arrays of chars, malloc can be used to create variable length “strings”

```
int main() {
    int n = rand() % 100;          //random number between 0 and 99
    char *string = malloc(n);
    //a char is one byte, so we don't have to use sizeof
    for(int i = 0; i < n-1; i++) {
        string[i] = 'A';
    }
    string[n-1] = '\0';          //a string must be null terminated!
    free(string);                //make sure to free the string!
    string = NULL;
    return 0;
}
```



# mallocing structs

- We can also use `malloc` to allocate space on the heap for a struct

```
typedef struct foo {
    int    value;
    int[10] array;
} foo_t;

int main() {
    //sizeof(foo_t) gives us the size of the struct
    foo_t *fooStruct = (foo_t*) malloc(sizeof(foo_t));
    fooStruct->value = 5;
    for(int i = 0; i < 10; i++) {
        fooStruct->array[i] = 0;
    }

    //free the struct. DON'T need to also free the array
    free(fooStruct);
    fooStruct = NULL;
    return 0;
}
```



# Functions

- So far, we've done everything in main without the use of functions
- But we have *used* some functions
  - `printf` in “hello world”
  - `malloc` for allocating memory
- C functions similar to methods in Java
  - In Java, a method is tied to a particular class  
In C, all functions are global
  - In Java, you can have overloaded methods, but not in C



# Our first function

```
int foo(int a, int b) {  
    return a + b;  
}
```

```
int main() {  
    int sum = foo(1, 2);  
    return 0;  
}
```

- First we define the function “foo”
  - foo will return an int
  - foo takes two ints as arguments
- In main, we call foo and give it the values: 1 and 2
  - foo is invoked and returns the value 3



# Order matters

- In Java, the order that methods are defined doesn't matter
- Not true in C. Look what happens when we flip the order around:

```
int main() {  
    int sum = foo(1, 2);  
    return 0;  
}
```

```
int foo(int a, int b) {  
    return a + b;  
}
```

- Compiling the above code yields the following errors:

```
test.c: In function `int main()':
```

```
test.c:2: error: `foo' undeclared (first use this function)
```

```
...
```

```
test.c: In function `int foo(int, int)':
```

```
test.c:6: error: `int foo(int, int)' used prior to declaration
```



# Definition vs. declaration

- In C, we can *declare* functions using a “function prototype” without *defining* what happens when the function is called

```
int foo(int, int);           //function prototype
```

```
int main() {  
    foo(1, 2);  
    return 0;  
}
```

```
int foo(int a, int b) {  
    return a + b;  
}
```

- Now the program will compile



# Declarations

```
int foo(int, int);
```

- In a function declaration, all that is important is the *signature*
  - function name
  - number and type of arguments
  - return type
- We do not need to give names to the arguments, although we can (this is often done)
- We can also declare `structs` and `enums` without defining them, but this is less common



# void\* (\*function) (point, ers);

- You already know about pointers to data. Remember, a pointer just stores an address in memory.
- Functions have an address in memory just like everything else, so a “function pointer” stores that address

```
int foo(int a, int b) {  
    return a + b;  
}
```

```
int main() {  
    int (*funcPtr)(int, int) = &foo;  
    int sum = funcPtr(1, 2);  
    return 0;  
}
```



# Let's break it down

```
int (*funcPtr)(int, int) = &foo;
```

## ○ Starting from the left...

- `int`: the return type
- `(*funcPtr)`: the `*` denotes that this is a function *pointer*, not a regular function declaration. `funcPtr` is the name of the pointer we are declaring
- `(int, int)`: the argument list
- `= &foo`: we are assigning the address of `foo` to the pointer we just declared



# ...if you thought that was ugly

- What is being declared here?

```
void* (*func)(void* (*)(void*), void*);
```

- First person with the correct answer wins a delicious pack of Skittles®



# The Answer

```
void* (*func)(void* (*)(void*), void*);
```

- We are declaring a function pointer called `func`. `func` is a pointer to a function which returns a `void*` (un-typed pointer), and takes two arguments. The first argument is a function pointer to a function that returns a `void*` and also takes a `void*` as its only argument. The second argument of function pointed to by `func` is a `void*`.
- Inconceivably contrived way to set a pointer to null:

```
void* foo(void* pnt) {  
    return pnt;  
}  
void* bar(void* (*func)(void*), void* pntr) {  
    return func(pntr);  
}  
int main() {  
    void* (*func)(void* (*)(void*), void*) = &bar; //here's the declaration  
    void* pntr = func(&foo, NULL); //pntr gets set to NULL  
    return 0;  
}
```



# main arguments

- Remember those funky arguments that were passed to main in the hello world program?

```
#include <stdio.h>
```

```
int main(int argc, char **argv) {  
    printf("Hello, world!\n");  
    return 0;  
}
```

- Main arguments are a simple way to pass information into the program (filenames, options, flags, etc.)



## argc and argv ?

- `argc` stands for “argument count”
  - An `int`, representing the number of arguments passed to the program
- `argv` stands for “argument vector”
  - A `char**`, meaning an array (size `argc`) of null terminated strings.
  - The first (0<sup>th</sup>) null terminated string is always the name of the executable



# Argument example program

```
#include <stdio.h>

int main(int argc, char **argv) {
    for(int i = 0; i < argc; i++) {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

- Enters the loop once for every argument
- Prints out each argument, adds a newline
- Program output:

```
> gcc hello.c -o hello
> hello how are you today?
hello
how
are
you
today?
```