

UMassAmherst



# CS197c

## Intro to C/C++

Nicolas Scarrci

University of Massachusetts  
Amherst



# Before we start: C99

- The edlab machines do not conform to C99 by default, so technically the following does not work:
  - `for(int i=0;i<10;i++){ ... }`
- Solution 1
  - `int i;`  
`for(i=0;i<10;i++){...}`
- Solution 2
  - `> gcc -std=c99 filename.c`  
`> ./a.out`



# Important functions

- `FILE * fopen(char * file, char * mode);`
  - `char * file` is the name of the file.
  - `char * mode` determines what io is allowed
    - “r” means read only
    - “w” means write only
  - The `FILE *` returned is needed for file io.



# Important functions

- `int fscanf(FILE * file, char * format, &args)`
  - `FILE * file` is the `FILE *` returned by `fopen`.
    - NOTE: you must open in read mode
  - `char * format` is a string containing format specifiers
  - `&args` is the list of addresses to read values into



# Important functions

- `int fprintf(FILE * file, char * format, args)`
  - `FILE * file` is the `FILE *` returned by `fopen`.
    - NOTE: you must open in write mode
  - `char * format` is a string containing format specifiers
  - `args` is the list of variables to print



# Important functions

- `void fclose(char * file);`
  - `char * file` is the name of the file.
  
- It is important to call `fclose` on all files that you open with `fopen`.
  - C is reserving a lot of memory behind the scenes.



# File IO basics

- `#include<stdio.h>`  
`int main(int argc, char * argv[]){`  
    `FILE * f = fopen("filename.c", "r");`  
    `int a;`  
    `fscanf(f, "%i\n", &a);`  
    `fclose(f);`  
    `f= fopen("copy.c", "w");`  
    `fprintf(f, "%i\n", a);`  
    `fclose(f);`  
    `return 0;`  
`}`



# Your Assignment

- Write a parser2.h which lists function prototypes.
- Write a parser2.c which implements these function prototypes.
- Write a structs.h which defines all structs used.
- Write a gamelogic.c which utilizes parser2.h functions to play the game.
- Write a makefile that creates an executable called 'JCAdventures'.



# Pointer basics

- `&variable`
  - This is the address of the variable in memory.
- `*ptr`
  - This is the value stored in the memory address held by `ptr`.
  - This sort of access is called dereferencing.
- When declaring a pointer use:
  - `type * ptrname;`



# Pointer arithmetic

- When we add to pointers it considers the type!
- ```
int myint;  
int * intptr = &myint;  
intptr=intptr+4;
```
- If  $\&\text{myint} = 0\text{xBEF08763}$  then  
     $\text{intptr} = 0\text{xBEF08773}$   
    (plus 16, or  $4 * \text{sizeof}(\text{int})$ )



# Pointer arithmetic

- `char mychar;`  
`char * charptr=&mychar;`  
`charptr=charptr+4;`
- If `&mychar = 0xBEF08763` then  
`charptr = 0xBEF08767`  
(plus 4, or `4*sizeof(char)`)



# Important functions

- `int sizeof(typename)`
  - The int returned is the size of the type in bytes.
  - `typename` is the name of a type
    - `int`
    - `char`
    - User-defined structs
    - etc.



# Important functions

- `void * malloc(int size)`
  - The `void *` returned is a pointer to the area reserved in memory.
  - `int size` is how much memory is being reserved.
    - Usually you call `malloc(sizeof(type))`



# Important functions

- `void free(void * ptr)`
  - `void * ptr` refers to the address in memory to be freed.
    - NOTE: `free` will free the whole chunk.



# C strings

- Pointer to the beginning of a string.
  - `char * string="hello";`
- Character array where `array[0]` begins the string.
  - `char string[10];`
- Ended in memory by a `'\0'` or `NULL`.
  - For this reason `char string[10]` only holds 9 characters.



# Arrays in C

- `int myints[10];`  
`int * ptr = &myints[0];`
- Is the same as
- `int myints[10];`  
`int * ptr = myints;`
- `myints` is a pointer to the beginning of the array.
- `myints` is a **constant** pointer!



# Arrays and pointers

- `int myints[k];`  
`int * ptr = myints;`
- `myints[i];`
- Is the same as
  - `*(ptr+i);`



# Multi- arrays

- `int myints[m][n];`  
`int * ptr = myints;`
- `myints[i][j]`
- Is the same as
  - `*(ptr+(i*n)+j);`
- Is the same as
  - `*(*(ptr+i)+j);`



# Array oddity

- `int myints[m][n];`  
`int * ptr = myints[2];`
- Is the same as
  - `int * ptr = &myints[2][0];`
- Is the same as
  - `int * ptr = &myints[0][0]+2*n;`
- Is the same as
  - `int * ptr = &myints[0]+2;`



# *Fancy pointers*

- `struct hero jackieChan;`  
`hero * chanptr=&jackieChan;`
- We can access the fields “directly”.
  
- `chanptr->name;`
- Is the same as
- `jackieChan.name;`



# Segmentation

- C separates memory into four sections
  - Code segment (stores your code)
  - Data segment (global variables)
  - Stack segment (parameters/local variables)
    - Stack data is deleted when a function exits
  - Heap segment (used by malloc and free)



# Segmentation Faults

- Occurs when you write to read-only memory
- Occurs when you attempt to access memory that you do not have permission to access
  - ex. `*(ptr+1000000);`
- Occurs when you write/read uninitialized memory
  - ex. dereferencing a null pointer



# Returning data

- Variables declared in a function are local
  - So they die when the function exits
- There are two ways to beat this:
  - Copy the data returned (Ex, using strcpy)
  - Pass in a pointer and modify it's contents.
- There are shadier ways:
  - Create static variables
  - Use global variables



# Function prototypes

- If gcc encounters a function call before that function's definition it assumes the function returns an int.
- This can be solved with a function prototype
- `returnType name(arglist);`
  - Just like an abstract method in Java.



# *Without prototypes*

- Without them
  - ```
int main(int argc, char * argv[]){  
    short fivefunc();  
}  
short fivefunc(){  
    return 5;  
}
```
- > gcc filename.c



# Output

- filename.c:5:7: error: conflicting types for `fivefunc` [enabled by default]
- filename.c:3:15: note: previous implicit declaration of `fivefunc` was here



# With prototypes

- With them
  - ```
short fivefunc();  
int main(int argc, char * argv[]){  
    short fivefunc();  
}  
short fivefunc(){  
    return 5;  
}
```
- > gcc filename.c
- Runs happily



# *Cleaning up our files*

- filename.c

- #include “fivefunc.h”

- int main(int argc, char \* argv[]){  
    short fivefunc();  
}

- fivefunc.c

- #include “fivefunc.h”

- short fivefunc(){  
    return 5;  
}



# *Cleaning up our files*

- fivefunc.h
  - short fivefunc();
- Compiling together
- gcc filename.c fivefunc.c
- Note: a header file is a good place to put struct definitions



# *A bit more with make*

- filename: filename.c fivefunc.o  
gcc filename.c fivefunc.o
- fivefunc.o: fivefunc.c  
gcc -c fivefunc.c
  
- > make  
> ./a.out



# Compiling files

- Create header files with function prototypes/struct definitions
- Create a main.c file which includes the headers
- Create file.c for each file.h which includes file.h and fleshes out each function prototype



# Your Assignment

- Recommended layout
  - Define a struct for the hero
  - Define a struct for the villains
    - Or one for each villain and one for the team.
  - Write a prototype for loading saves
  - Write a prototype for saving games
  - Write a prototype for printing the map (to string)
  - Write a main function in gamelogic.c for handling input and game logic.



# Your Assignment

- I will provide a function for:
  - Returning input (as an array of strings)
  - Printing maps (taken in as a string)
  - Winning the game
  - Losing the game
  - Taking the villain's turn (will return a matrix of new villain positions)
- In addition there will be:
  - `registerSave(char * savename);`

