Tar

Tar.gz

- Often called a tarball
- Tar is a Linux command to create an archive of a set of files
  - The gz part compresses the files
- Has a number of commands
  - You can use the man pages to see all the commands
  - We will only need a few of them

Creating a Tarball

- Syntax:
  - Tar <flags> <name of the tar.gz> <list or path to the files>
  - Order matters - you screw this up you will have a corrupt file

Creating a tarball continued

- Flags to tar
  - c: Create an archive
  - z: Compress the archive with gzip
  - v: Display progress in the terminal, also known as “verbose” mode - optional but helpful
  - f: Allows you to specify the filename of the archive
- Let’s do an example with files: **tarTest1.txt, tarTest2.txt, tarTest3.txt**

Opening a Tarball

- We can open a tarred file in the same manner as creating with a couple changes
- Syntax:
  - Tar <flags> <name of tarred file to be opened>

Flags for Opening a Tarred file

- All the same except change “c” to “x” for extract
- -xzvf
- You can have the files untag to a specific location by using the -C flag
  - Tar -xzvf archive.tar.gz -C tempFolder (specify the path of the folder you want the files to go)

Header Guards

Header Guards

- They are conditional pre-processing directives
  - Helpful in “C” due to cyclical dependencies
  - More important for “C++” because of redefinition issues
  - Standerd for C and C++

Header Guards

- Syntax

```
#ifndef UNIQUE_NAME
#define UNIQUE_NAME
CODE
#endif
```

- #pragma once is often used rather than header gaurds - however #pragma is not standard for c and c++
- Some compilers do not recognize it
- Not always portable
- DO NOT USE PRAGMA IN THIS CLASS

File Handling

File Handling in C

- There are 4 basic operations you can do with files
  - Create a new file
  - Open an existing file
  - Close a file
  - Read from and write information to a file

Types of Files

- Text based files (.txt - plain text, characters, etc.)
  - Readable
  - Editable
- Binary Files (.bin - basically1’s and 0’s)
  - Can hold higher amount of data
  - Not easily readable
  - More secure

Working with files

- Declare a point of type FILE*
  - FILE* ptr;
- Then open the pointer
  - ptr = fopen(“filename”, “mode”)
- What are the modes?

File Pointer Modes

Mode	Meaning of Mode	Action if file not available
r	Open for reading	If the file does not exist fopen returns null
rb	Open for reading in binary mode	If the file does not exist fopen returns null
w	Open for writing	If the files exists, the content is overwritten If the file does not exist, it will be created
wb	Open for writing in binary mode	If the files exists, the content is overwritten If the file does not exist, it will be created
a	Open for append. Data is added to the end of the file.	If the file does not exist, it will be created
ab	Open for append in binary mode. Data is added to the end of the file	If the file does not exist, it will be created
r+	Open for both reading and writing	If the file does not exist, fopen will return NULL
rb+	Open for both reading and writing in binary mode	If the fees does not exist, fopen will return NULL
w+	Open for both reading and writing	If the file exists, the content is overwritten If the file does not exist, it will be created
wb+	Open for both reading and writing in binary mode	If the file exists, the content is overwritten If the file does not exist, it will be created
a+	Open for both reading and appending.	If the file does not exist, it will be created
ab+	Open for both reading and appending in binary mode	If the file does not exists, it will be created

fread()

- size_t fread(void * ptr, size_t size, size_t n, FILE* fp);
- void* ptr: the starting address of the memory block where data will be stored after reading from the file.
- size_t n: the number of items to be read
- size_t size: the size of **each** "**n**" item read in
- On success, fread reads **n** items from the file and returns **n**
- On error or End of the file, it will return less than n.

assert in C

- Assertions are statement used to test assumptions made by programmers. Evaluates to True or False.

- FILE* input = fopen(“file.txt”, r)

  assert(input != NULL);

  This checks if the file opened successfully. If the expression evaluates to false (0), then the expression, source code filename, and line number are sent to standard error, and the abort() function is called. This will abort the program.

rewind

- We have seen an example of rewind
  - It simply moves the file pointer to the begining of the “file” or buffer
  - Advantage: easy to call
  - Disadvantage: cannot check if successful - clears the error indicator

fseek

- Similar to rewind
- Syntax: int fseek(FILE* str, long int offset, int pos)
  - str - the opened file pointer
  - offset - the number of bytes or characters - determines where the position indicator needs to be placed to define the new position of the file pointer
  - pos - defines the point where the file offset needs to be added - basically defines the position where the file pointer needs to be moved.

fseek

- Three constants mostly used
  - SEEK_CUR: indicates the current position of the pointer of the file
  - SEEK_END: moves the pointer of the file to the end of the file
  - SEEK_SET: moves the file pointer to the beginning of the file
- ftell - returns the current position of the file pointer.
- CODE: FileSeek.c

Structures

Structures

- Structures allow us to group various data types under one name. Basically create our own data type.
  - Must use the key word **struct**
  - Must be grouped together using { } with a ;
    struct structName
    {
        Datatype one;
        Datatype two;
        Datatype* three;
    };

    We now have a new data type called structName. To create an instance of this new data type:
    struct structName varName; (The keyword struct is necessary)

Structures

- Suppose I don't want to use that pesky keyword struct every time I create an instance of the struct. I can use the keyword "typedef"
- You should already know this so we will move on to sample code

Structure Padding

- Structure padding is a concept in C that adds one or more empty bytes between the memory addresses to align the data in memory
- padding.c

Other info about bit fields

- Why use them?
  - They are often used in embedded systems because of the scarcity of memory
  - Saves memory, reduces the size of the struct
- Can not have an array of bit fields
- Only integral types can use bit fields: bool, char, short, int, long
- Cannot print the address of a bit field member

Self-Referential Structures

- A structure that contains a member that has the same type as the structure.

```
struct SELF_REF{
        int a;
        struct SELF_REF b;
        int c;
};
```

This has a problem. Compiler needs to know the size in order to seaside memory. Since it is referring to itself it does not know the size.

We can fix this?

Self-Referential Structures

- To fix the problems the self-referential variable must be a pointer. We know that a pointer no mater the type is always 8 bytes. (Assuming 64-bit architecture.

```
struct SELF_REF{
        int a;
        struct SELF_REF *b;
        int c;
};
```

Self-Referential Structures

- Another trap easy to fall in.

```
typedef struct{
        int a;
        struct SELF_REF *b;
        int c;
}SELF_REF;
```

What is the problem here?

Pointer Review

Pointer Review

- By this point you should be familiar with pointers. So we will quickly go through the basics, then more advanced pointer information
- What is a pointer:
  - A pointer is something that points to something else
  - In programming a pointer is a variable, whose value is the address of another variable that is stored somewhere in the computer memory
- Obtaining the value stored at that location is known as indirection/ dereferencing

Why Use Pointers

- Gives the ability to work with particular memory locations
- Necessary for programming microcontrollers
- Necessary for arrays and strings(C++)
- Allows us to effectively represent complex data structures (linked list, stacks, queues, trees, etc)
- Change values passed as arguments to functions
- Optimization of programs -pass pointer to a function does not have to duplicate the data
- So many more

Double Pointers

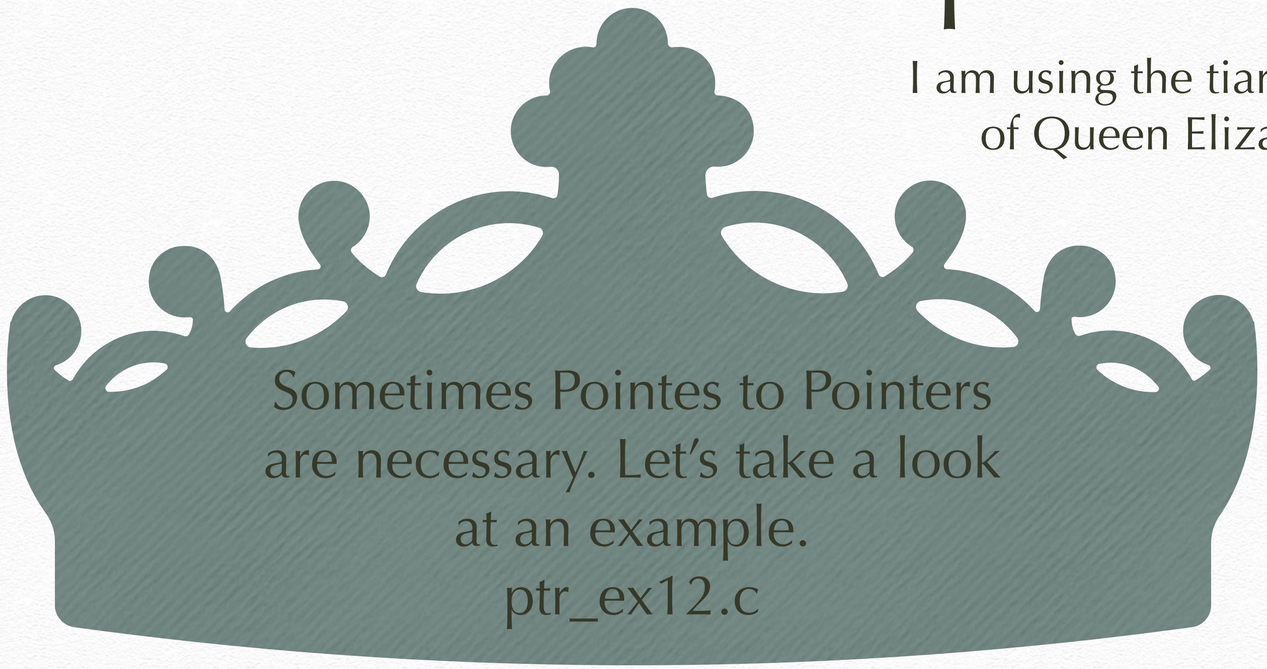
- We know we can have pointers to ints, chars, floats, etc
- It should be no surprise that we can have pointers to pointers as well
- You have already been using double pointers when you use command line arguments
- Two reasons to use double pointers
  - Used to handle multi-dimensional arrays
  - When passing a pointer by reference

int main(int arc, char* argv[])
char*argv[] is a double pointer
argv is basically an array of strings

Consider the following:
./a.out first.txt second.txt third.txt
argv[0] is a pointer pointing to the string ./a.out
argv[1] is a pointer pointing to the string first.txt
Etc.

Double Pointer Example

- int = 5; int j = 6; int k = 7;
- int **Ptr2Ptr;
- int *intPtr1 = &i;
- int *intPtr2 = &j;
- Ptr2Ptr = &intPtr1;



- *Ptr2Ptr = intPtr2 - This says you go to where Ptr2Ptr is pointing and set the value equal to the value of intPtr2 and intPtr2 holds the address of j so intPtr1 will now point to j
- Now if we do the following *Ptr2Ptr = &k - this says go to where Ptr2Ptr is pointing and set it to the address of k - Ptr2Ptr points to intPtr1, so intPtr1 is now pointing to k

Dynamically Allocation

Dynamic Memory

- When we declare - int arr[10] = {1,2,3,4,5,6,7,8,9,10}; or int a = 10; In C space is allocated on the stack
- However, often we may not know at compile time how many integers we will need for ‘arr’ we will not know this until run time
- C gives us a way to handle this by dynamically allocating the memory using Malloc or Calloc

Malloc

- Prototype: void *malloc(size_t size)
  - Returns a pointer to the beginning of the allocated memory
    - int *ptr = (int*) malloc(num*sizeof(int)); num represents a value provided prior to calling malloc
  - Must #include <stdlib.h>
  - When using malloc must also free the memory after you are finished with the memory
  - Void free(void *ptr);
  - So, what is the difference in malloc and calloc

Calloc

- Calloc - void *calloc(size_t num, size_t size);
  - Returns a pointer to the beginning of the allocated memory
    - Int *ptr = (int*) calloc(num, sizeof(int)); num represents a value provided prior to calling malloc
- Must #include <stdlib.h>
- When using calloc must also free the memory after you are finished with the memory
- Calloc also initializes the data, where as malloc does not.
- void free(void *ptr);

1D Malloc Example

- This is an example of mallocing for a a one-dimensional array
  - int *arr = (int*)malloc(row*col*sizeof(int)); ‘arr’ points to this place in memory

  - There is a neat trick that allows us to think of this as a 2D array but use 1D notation.
    - arr[(current_row*(total width)) + current_col]
    - Malloc1.c and Malloc_Ex1.c

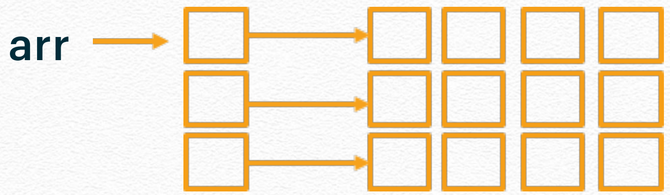
Malloc/Free 2D

int **arr = (int**)malloc(row*sizeof(int*));

for(I=0;i<row;i++) {arr[i] = (int*)malloc(col*sizeof(int));}

I can now access arr using 2D notation: arr[2][1];

Example below is a 3X4 matrix.



PRINTF
AND
SCANSET CONVERSION

PRINTF FORMAT SPECIFIERS

- Prototype for format specifier
  - %[flags][width][.precision][length]specifier

specifier	argument
d or i	Signed INT
u	Unsigned INT
o	Unsigned Octal
x or X	Unsigned Hexadecimal (lower/upper case)
e or E	Floating Point
g or G	Shortest Representation
a or A	Hexadecimal Floating Point (lower/upper case)
c	Character
s	String of characters
p	Pointer address

flag	description
-	Left-justify (default: right)
+	Force print + sign with positives
(space)	Add space, if no sign before value
#	Precede o, x, or X with 0
0	Left pad number with zeros

.precision	description
.number	specifies number of digits to write
.*	not specified in cstring, additional INT value given

width	description
(number)	Minimum number of characters to print
.	Not specified in cstring, additional INT value given

print.c
multiLinePrint.c

https://www.educative.io/answers/how-to-use-printf-in-c

Scanset Conversion

- **We already know you can use (f)scanf(), (f)getc(), (f)gets() to read data**
- **We also can read string data using scanset conversion**
- **Scanset is basically a specifier supported by scanf family functions. Allows us to process a particular set of characters, numbers, etc.**
- **scanset.c, testScanf.c goodTest.txt test.txt**
- **https://www.knowprogram.com/c-programming/read-and-display-the-string-in-c-programming/**

MACROS

Macros

- Often used to define something that is used repeatedly in a program
- We have already seen something that is considered a macro
  - #define MAX_SIZE 10
- You can also undefined an object
  - #undef MAX_SIZE
  - Code: macroEx1.c

Predefined Macros

- _DATE_ give the date you compiled
  - FORMAT: MmmddYYYY
- _FILE_ gives the file name of the current source file
- _LINE_ gives the line number within the current course file
- _TIME_ gives the time you compiled
  - FORMAT: hh:mm:ss
- _func_ prints the name of the current function - this is a predefined identifier rather than a macro
- Code: predefinedMacros.c

FUNCTION MACROS

- We can create macros that are basic functions
- To define a function like macro you use the same #define directive. However, you must put () after the name. In earlier classes you probably learned that spaces do not usually matter, i.e. int* a; is the same as int *a; Space is often ignored in C. However, not the case when creating a function like macro.
- To avoid trouble with operator precedence, it is wise to use () in your definition
- CODE: macroEx2.c, macroEx4.c

WRAPPING UP MACROS

- As we have discussed macros are basically a text substitution that are handled during the pre-processing step of the compilation process
- Down fall of macro handling is during the substitution process, the compiler does not do type checking
- Macros are risker when called multiple times. Such as, recursively or when the same variable is used multiple times in the definition of the macro
- The use of macros with some C functions can be problematic
- CODE: MacroSideEffect.c