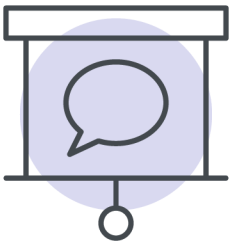
UTSG

CSC209H1
Final EXAM
STUDY GUIDE



Lecture Notes

CSC209 – C Basics

Input, Output and Compiling

- Printing Values
  - **printf()** → is a function to print to the screen
    - For C to find it you must include: **#include <stdio.h>**
    - Library name is standard io (input and output)
    - Text MUST be based inside of double quotes
    - **\n** is interpreted as a single character → end of line
    - To print a variable you need a format specifier
      - **printf**("text [identifier]\n", variable name)
      - **%d** → expect an integer (can also use %i)
      - **%f** → floating point
  - If you want a specific number of digits after the decimal… **%.**[number of digits to print after the decimal]**f**
  - A function is a well-defined piece of code that carries out a specific task
  - The command line has the notion of a current working directory
- Compiling From the Command Line
  - **cd** → change directory
  - **ls** → lists file in current directory
  - **gcc** [program name aka argument] → produces an executable file
    - Will automatically saved to a file called a.out
  - **./** → runs program
  - How to check if a file is of executable type?
    - ls –F → shows asterisk
  - **ls –l** → lists files and shows permissions
  - -**Wall** → prints out additional warning messages
  - -**o** → allows to specify name of executable file in which gcc will save its output
- Reading Input
  - printf() and **scanf()** are similar
    - Recall that to use printf() you must include io lib → need same lib for scanf()
    - scanf() also needs format identifiers
    - Instead of %f scanf uses **%lf**
    - In order for scanf() to change the value of a variable to what the user types, it is user to tell scanf a location (memory) to scanf → **&**[variable] gives location
    - Can take more than 1 piece of data by prompting the user to provide input separated by a space (for numbers)

Types, Variables and Assignment Statements

- What is a Program?

- Program is…
  - Sequences of instructions to a computer
  - The power of computers lies in programs and how we control the CPU
- Complier converts the C code into much more complicated machine code for the computer to understand
- **main()** → acts as a wrapper for instructions
  - Forms the core for each program
- Text inside of printf() quotation marks (if a str) is called a String Literal
- **return 0** → tells the computer to finish executing the main function, anything after this won't run

- Introducing Variables
  - Computers store information in memory → giant grid with cells, with individual memory addresses
  - Variable declaration statement: [data type] [variable name];
    - Ex. int canada;
  - Variable names must follow some rules:
    - Contain letters, numbers and underscores
    - NOT begin with a number
    - Case-sensitive (upper and lowercase letters make a difference)
  - If you do a variable declaration if the variable has already been declared, the previously contained data is still there
  - Assignment statement: [variable name] = data;
  - Can declare empty variables of the same type on the same line, by separating variable names by commas
  - Expressions → ways to perform arithmetic on variables
- Using Variables in Expressions
  - Note: C doesn't have a built in exponentiation operator!
  - BEDMAS follows in C expressions → AKA operator precendence
  - Obvious but note: any changes to variables do not change previously established variables derived from expressions using these variables (if set at one instance)
- Double Variables
  - Storing a variable that isn't a whole number? → Doubles allow you to store floating point values
  - If a floating point value is saved to a variable of type int, it will cut/truncate off the decimals
  - Doubles save to 2 memory addresses and takes DOUBLE the amount of space as an int (go figure)
  - Type Double has limited precision and limited number of significant digits
  - **[number] / [number]** → gives the INTEGER result of this division, and outputs only whole numbers
- Programming in Style
  - Readability and Documentation: some basic rules to follow

        - Operators should always have space on both sides
        - Statements should always be on separate lines
    - White space between variables and operators does not change the function of the code, but is crucial for readability
    - Clear up ambiguity by clarifying descriptive variable names
    - Conventions for variable names in this course (called snake_case or pothole_case):
        - All lowercase
        - Words separated by underscores
    - In camelCase (another valid convention type):
        - The first letter is lowercase
        - No underscore
        - Each new word begins with uppercase
    - To improve readability, keep each line of code up to 79 characters or less, can put the rest of the statement on the next line (with an indentation on the lower line)
    - Commenting → English description of what program does
        - Benefit for collaborators and users
        - **/*** → begins a comment that continues for as many lines as you like until ***/** is specified
        - **//** → begins a single line comment
- Storing Characters
    - Char data type → for characters
        - Two ways of declaring char variables:
            - Literally: ex. **char** letter = 'a'
            - OR with ACSII number: ex. char letter = 97
            - BOTH score the letter a
        - The code for lowercase and (not both together) upper case characters are continuous (don't have other characters between, good for counting up or down the alphabet)
        - char + 1 → will move 1 up in the char codes [ex. a → b]

Functions

- Calling a Function
    - C comes with a number of built-in functions
    - **fmax()** → will return a result of the largest number in the 2 given arguments
        - Must include: **#include <math.h>**
- Using Function Calls
    - Can use variable, and expressions as function arguments
    - Functions that return a value/datta can also be used inside of other functions, or even itself, as arguments (with their own arguments specified)

- Introduction to Writing Functions
  - A function prototype is a function declaration
    - **[data type] [name of function]([parameters with types specified, CAN include names]);**
  - Function definition:
    - **[data type] [name of function]([parameters]) {**
      }
  - What happens if you forget to tell the complier how to call a function?
    - AKA removing include statement
    - Program warns, but still builds the program → at our own risk
  - If you remove function prototype…
    - Warning AND an error, as the complier has to guess how to call the function
    - Doesn't compile in this case
  - Can define the function before it is used to declare and define at the same time
- Writing a New Function
  - Type long is an int value that includes decimals with more precision than float types
  - **lround()** → rounds the value and returns a long
    - in the math library, therefore must be #include
  - **else if()** only runs if the previous if() fails
- Visualizing Function Execution
  - Stack frame → A section of memory created for the function specified, where the variables are stored
    - Last in, first out order → removing from the top of the stack
  - Variables declared inside of specified functions are only usable/accessible within the score of the function AKA local
  - Using a function return value to declare a variable → AKA pass by value

Iteration

- Introducing For Loops
  - Each iteration of a loop can not only run multiple times, but actions depend on the previous iterations
  - int [variable name] ****MUST DO THIS BEFORE THE LOOP
    for (INITIALIZATION; CONIDITION [if true, continues]; UPDATE) {
    }
- Nested Loops
  - \n must be contained in quotations when printed
  - **\t** → single tab space/character
  - Can nest loops within other loops to run instruction/code chunks multiple times
  - If using a nested loop, the initialized counting variable must be distinct for each of the different loops

        - Choose meaningful names or use i for the counter
- While Loops
    - **int i;** ****ALSO NEEDED HERE
      i = 0;
      while(CONDITION [if true, then loop body is executed again]) {
      }
    - Semantic sugar → convenient structure
        - Not as many specific expectations
        - Update must occur inside the loop body
    - Do While Loop…
      do {
      LOOP BODY
      } while (CONDITION)
    - Good for retrieving user input
    - Loop body is executed before the condition is checked
    - The condition in the loop is a continuing condition, but we mostly think of stopping conditions
    - **!=** → Is NOT condition
- Break and Continue
    - **break** → terminates loop iteration
    - Replacing the loop condition with a strange value (i.e. just a number) → will always evaluate to True
        - This is a flag and tells the user that something interesting is happening inside of the loop
        - Good way to show the user that break is contained inside the loop
    - **continue** → causes the rest of the loop body to be skipped; within the same loop, not the most other loop in the case of nested loops
    - Try not to use these statements, to improve readability

Boolean Expressions and Conditionals

- Introducing If Statements
    - Provides the functionality to pick between two actions
    - Let us control which statements in a program are executed
- Conditional Operators
    - && → is the logical AND operator
        - Connects two expressions
        - Resulting compound expression is true when the expressions on the left AND right are both true
    - || → the logical OR operator
        - Resulting compound expression is true when the expression on the left is true, or the right is true, or both are true
    - ! → the logical NOT operator
        - Negates condition
- Structuring If Statements

- Can have if and else if blocks even without the final else statement

Types and Type Conversions

- Numeric Types
  - int = double results in truncation of the decimals
    - In the case in which the double is so large it exceeds the maximum integer that an be represented on the computer
    - There is no warning but stores the integral value in the int
  - Can all integers be represented by doubles?
    - True in pure math but not true in computer representations
  - INT_MAX → the largest/max value that can be contained in an integer variable
  - A float uses the same amount of bytes as an int, but a double uses…double
  - If an integer is larger than a float can precisely represent it provides an estimate
  - What if we have an int value too large to put in a char?
    - Conversion drops the higher order bytes (# - 256???) to hold 64
  - You need to remember integral values are limited in range, floating point values are limited both in range and precision
    - Think carefully if variable types are appropriate for data transfer
- Casting
  - Even if the variable is a double, if the operands and the operator are of type int, the result will be an int, then turned into a double, therefore losing the valuable decimals
    - But if at least 1 of the operands is of type double, the result is accurate
  - Can convert int to double (Casting):
    - Ex. d = (double) i/j
    - This doesn't change the type of the operands in memory

CSC209 – Week 2: Arrays and Pointers

Arrays

- Introduction
  - Arrays can hold multiple variables of the same type within the same name
  - Specify the size of the array after the name with [NUMBER] → must be done before placing data inside
  - Index of the array starts at 0 and ends at (length-1)
- Accessing Array Elements
  - Can use {} and data separated by commas to initialize variables within the array instead of individual statements for each index
  - Arrays cannot change in size; once declared and allocated, there is no way to go back and make the array larger
  - Once we know the address where the array starts, and the size of each element, we can calculate the address of each element (based on the type of the variables) → we rely on that fact when we use indices
    - Ex. Add 4 to the address of Array at index 0, to get the memory address of Array at index 1
  - If you try to access an element of an array that is out of bound, the program will still compile, with warning → c doesn't store the array bounds, thus cant even check the bounds
    - Sometimes just replaces the invalid element with 0, or the value belonging to a different variable!!
    - This may not always be the case, the memory allocated is not within the array, so it could contain any data at all in it
  - If you try to assign a value to an out of bounds element of an array, it is troublesome because it might replace a value another variable is using → which will be very hard to find
  - Note: Memory Addresses are hexadecimal
  - Segmentation fault → the address that was accessed was not legal!
    - Actually better than incorrect program
- Iterating Over Arrays
  - If you notice repetition in your code → replace with a loop!
    - And move other actions to another statement/line
  - Can use i as the index variable within the loop (starting at 0)
  - Use a pound to find constant, to store array size if it is used more than to initialize
    - Below #includes (if any), write:
      #define [variable name ALL CAPS] [value]

Pointers

- Introduction

- Can hold the address of one variable and use it as a value of another AKA Pointer!
- **&[variable name]** → returns the address of the variable
- If a variable is a pointer, its value is a memory address
  - When declaring, you must specify the type stored at that address
  - To declare:
    [type] *[variable];
  - Ex. **int *pt;**
    pt = &I;
    - This chunk of code says that pt points to i
    - pt holds the address of i
  - Equivalent to…
    int *pt = q;
    but NOT
    int *pt;
    *pt = q;
  - To access the value pointed at by the pointer (pt here)
    - Dereferencing → ***pt**
    - This is different from the use of an asterisk in the declaration, as it is the type of the variable, not a use of dereferencing
- Can have pointers to other types, like chars

- Assigning to Dereferenced Pointers
  - Changing the memory pointed at by a pointer
  - ***[pointer name] = [data];** → changes the value contained at the memory address the pointer points to – not in the pointer itself
  - If a pointer is used to change the value in a variable, if another variable was previous initialized by this changed variable, nothing changes; only the variable pointed at has changes made
  - ***pt = *pt + 1;**
    - Value of the variable pointed to, plus 1
    - The result of this expression is assigned to the value of the stored memory address
- Pointers as Parameters to Functions
  - Void return type → won't return value to calling function
  - Function parameters are local variables in the stack frame of the corresponding function
  - Instead of returning a value to the calling function, you can use a pointer to change the variable values directly
- Passing Arrays as Parameters
  - When passing an array to a function normally, you are passing the address of the first index of the array rather than the entire array

        - Therefore initializing a parameter as ex. int A[4] isn't completely correct
        - Clearer to use ex. **int *A**
        - In this case, remember NOT to use sizeof(A) for looping, as it is a pointer to the first element of an array and not the full array, it will instead get the size of the data types in A
        - ***Provide array sizes as a parameter to avoid error*****
    - You do NOT need to dereference A (the array/elements) to change variables when passed in this way
    - **sizeof()** → returns the amount of indices in an array
- Pointer Arithmetic
    - What happens when an int is added to a pointer (int)?
        - The result is a pointer
        - If 1 is added to a pointer (which by definition contains a memory address), the result is a pointer 4/8 bytes larger based on the data types and thus byte size
        - Therefore anything added to the pointer will be multiplied by the byte size of its data type and added to the byte size of the address
    - 3 byte difference of a pointer to the char its pointing at
        - Address of the char is the location of the first byte of the word and the pointer is the exact address where the character is stored
    - Char take up 1 byte
    - A good example:

```
int A[3] = {13, 55, 20};
int *p = A;      // dereference p
printf("%d\n", *p); → prints 13
printf("%d\n", *(p + 1)); → prints 55
printf("%d\n", p[1] → prints 55
```

    - BIG TAKE AWAY: **address of A[i] = address of A + (i * size of one element of A)**
        - Also works for subtraction if the address of A is changed
        - Would be of type: **[data type] **[var name];**
        - If you dereference the pointer to the pointer (i.e. *pt_ptr), you get the value at the address stored in pt_ptr
        - To dereference twice use ex. ****pt_ptr**, but must be pointer to pointer to int
- Pointers to Pointers
    - Can store the address of a pointer in another variable
    - To get the value of the value pointed to by a pointer and pointed at by another pointer use **
- Why Use Pointers to Pointers
    - Can have arrays containing pointers
    - If the type of the elements in the array are pointers, the parameter type must be int **A (refer back to passing arrays as parameters)

- Therefore to dereference in this case, you do not need to use **A

CSC209 – Week 3 Memory

C Memory Model

- Code and Stack Segments
  - Imagine memory as a single vast area
    - A programmers, local view of memory
  - Memory is split into segments which each hold a different type of data

  - Even though it is known that arrays begin at 0, the Code Segment starts at an address that is close to but not zero
  - Once code is run it is stored near the top of the array
    - As the code runs, it goes through main() and then any other existing functions
    - Each function invocation is provided space in the stack
  - Stack space functions similar to stack data structure, most recent function call is stored at the top of the stack, and function calls are removed in last in, first out order
    - Stack is a location in memory reserved as a workspace for functions
  - To store local variables of a function, we must allocate a stack frame with enough memory in the stack to store them
    - Located at the top of the stack, above the stack frame for its called
    - When finished, it is popped off the stack frame and frees up space for other functions, and the function that originally called the function is again on the top
  - Local variables are only accessible if the function defines them as active
    - Once the function has been deallocated, the variables are no longer valid
- Heap and Global Segments
  - If we assign a variable that is not in main (or any other function for that matter) it is a global variable, then is accessible from anywhere
  - Cannot declare global variables in the stack
    - Stack only contains variables that have been defined for a certain function

        - They are stored in their own global data segment right below the code segment
    - Unlike the stack, global variables are not attached to any specific function
    - Global data holds more than global variables, also hold string literals
    - Even if a pointer is on the stack, value it holds AKA the memory address, cannot be held there
        - String literals are not always assigned to variables
        - That string/constant is stored in the global segment to be accessed from anywhere
    - Dynamic allocations
        - Malloc or memory allocations allows us to allocate memory while the program runs
            - Ex. **int *k = malloc(sizeof(int) * 500);**
        - Dynamically allocated memory should have a lifetime longer than the function in which it is located
            - Stored in the Heap
        - As you malloc more and more things into the heap, it eventually fills up
        - free() frees up space in the heap for more allocation
        - If the stack of heap is filled to its maximum, your pogram will reach an out of memory error (or e no mem)
        - The stack and heap wont ever meet at a boundary, other segment like file buffer segments may be located between them
        - The bottom holds an OS segment for the computer's own use
            - If you try to access this memory, you will get a segmentation fault
        - Segmentation faults occur when you try to access something we don't have access to or something that does not exist
            - Uninitialized variables will point to the beginning of the memory array which is not valid and will result in a segmentation fault
            - 0 or anything close to 0 is used as a buffer for failure

Dynamic Memory

- Introduction
    - When using a function to assign a variable and return a pointer to an address to the set variable, a warning occurs
        - Trying to return an address of a variable on the stack
        - If we ran this? It would appear to work
        - Memory is allocated on the stack for the variable in this function
        - Once the function is finished/returned the memory is freed!
            - And reallocated for another purpose

        - Therefore, the address we are referring to may point to a completely different variable depending on the other functions
    - Allocating memory and keeping it accessible after the function returns?
        - Use the heap!
        - **void *malloc(size_t size)** allocates memory on the heap
            - size → how many bytes of memory should be allocated
            - size_t → type defined by the standard library as the type return by sizeof(); in general is an unsigned int
            - void *→ holds the address of the memory allocated by malloc on the heap – what type does it hold? → returns generic type as it doesn't know how it will be used
        - Ex. **int *i_pt = malloc(sizeof(int));**
          ***i_pt = 5;**
    - Heap memory remains allocated until the user explicitly deallocates it
    - To calculate the address of all the elements in an array, a pointed needs to hold the element type information (due to variations in byte sizes)
- Freeing Dynamically Allocated Memory
    - Recall memory for a pointer is stored on the stack
    - If you use malloc to allocate memory inside a function, but do not store the address on the heap by returning it to the caller function, you have no way to access it
        - However it is not deallocated
        - This is called a memory leak
        - Doesn't cause a problem in the program but can be a serious underlying program
        - Each time a function with this problem is called, leaks one more size of memory that can no longer be used
    - Eventually long term memory leaks will result in **out-of-memory error: ENOMEM**
        - No more space for necessary variables
    - How to deallocate memory on the heap?
        - **void free(void *ptr);**
        - The function free takes a single argument which is a pointer to the beginning of a block of memory that had been previously returned by a call to malloc
        - free() returns that address to the system managing the memory for the program so it can be reallocated
    - If you try to access memory on the heap after free() and before the function is returned, it may appear to work
        - Free() doesn't reset the value of the variable and it doesn't change the values
        - Free() tells the memory management system that is block is now available for other purposes, therefore, until another variable

uses this memory, the previous memory still exists at the specified address!

- Dangling pointer → a pointer that points to memory that has already been freed; unsafe to use!

- Returning an Address with a Pointer
  - To continue using a variable and avoid a memory leak, need to retain a copy of the address on the heap
  - In order to change a variable from another function, pass the pointer to the variable to a parameter of the type **
    - You can have a return type of void, but allocate memory on the heap to represent the passed parameter!
  - To dereference a pointer with a variable for ease, ensure it is BELOW the malloc() statement

    Ex. ***arr_matey = malloc(sizeof(int) * 3);**
    int *arr = *arr_matey;
- Nested Data Structures
  - Need a nested array of pointers with each element that may itself point to an array
    - Use **malloc(sizeof(int *) * [# of elements])** for this
  - Instead of dereferencing an array (i.e. ***pointers[0] = 55;**) you can instead use **pointers[0][0];** to do the same thing!
    - This is in the case of an element of an array pointing to a 1 element array
  - Before freeing an array of pointers, you must free the subarrays/elements
  - When using malloc() you must consider when and where it is appropriate/safe to deallocate memory
    - Each malloc() should have a free() statement
    - Created from the top down and free in the opposite order
  - Memory left allocated on the heap at the end of a running program is automatically deallocated

Command-line Arguments + Type and Type Conversions

- Converting Strings to Integers
  - C strings are special arrays of char elements
  - We can declare and initialize a string literal variable like ex. **char *s = "hello";**
  - In the case… **char *s = "17";**
    - String with 2 characters, 1 and 7
    - Each holds 1 byte of memory and their appropriate ASCII code
  - **convert_to_an_integer()**
    - Takes a string as a parameter and returns the integer representing by the string
  - **long int strtol(const char *str, char **endptr, int base);**

        - "String to long"
        - **strtol([string we want to convert], NULL, [base of the number system, usually 10]);**
        - Returns a long which you can assign to an integer without losing information
        - Strtol() can handle leading spaces and/or +/- signs
        - The second parameter
            - If the string begins with an int and is followed by non-numeric characters?
            - Still extracts the integers correctly, and uses the middle parameter to tell the caller where the leftover piece of the string starts
            - Type pointer to pointer to char, it is the address of a pointer to char
            - Can pass NULL or the address of a pointer to char
            - After strtol() is run, it will contain the remaining piece of the string
            - Ex. **char *rest;**
              sum1 = strtol(input_line, &rest, 10);
              sum2 = strtol(rest, NULL, 10);
    - Strtol() skips over leading whitespace but trailing white space is unconverted
- Command-line Arguments
    - Can either use a main() with no arguments or two
        - **int main(int argc, char **argv){}**
        - First parameter could technically be named anything you want → holds the number of command line arguments
        - Second parameter can also be see as **char *arg[]**, argument vector → stores an array of strings
            - Each character pointer in the array refers to an array of characters → a string
            - The argument at position 0 is the name of the executable*** (ex. ./my_executable)
            - The array is as long as the amount of words + 1
    - If you need to use the command line for integers you can use strtol()
- A Worked Example
    - To read the first character of the first command-line argument: **argv[1][0];**
    - Use an extra if statement to check the amount of arguments and ensure you get the one specified
    - Use a non-zero exit code to terminate a program immediately (i.e. **return 1**)

CSC209 – Week 4: Strings and Structures

Strings

- Introducing Strings
  - Strings are the typical way we handle text
  - To store "hello" we can use a char to hold a letter in each element
    - But there's no convenient way to print this text – must use a loop
  - A C-string is a character array that has a null character immediately after the final character of text
    - Special null character → **\0**
    - Marks the end of a text/string
    - Never actually displayed as part of the string
  - Use %s when printf() the string
- Initializing Strings and String Literals
  - Instead of initializing strings by each element, you can use an array {} with characters separated by commas
    - **Don't forget the null character
  - If the null character is specified and the rest of the elements do not store a specified value, they also hold a null character (\0)
  - Can also just initialize string in double quotes – which is really just an abbreviation of the array form
    - Initialized characters hold their values, and non-initialized hold the null character
    - It is illegal for the initializer (AKA the portion in double quotes) to be longer than the char array
    - It IS legal for the initializer and the array to be the same size
      - Common source of bugs
      - There is no room for the null terminator
    - Can omit the size of the array using [] and the complier will allocate a char array with enough space for each character of the initializer, plus one for the null
      - BUT*** this does not mean the size of the array can change in the duration of the program
  - You can edit the elements of the string/char at all times since it is just an array
  - String literal → String constant that cannot be changed
    - Declared as ex. **char *text = "hello";**
    - Cannot be changed
    - Text points to the first character of the array/string
    - Trying to change the elements of a string literal will result in unexpected results at run time → Bus error

- Size and Length

- On an array sizeof() gives the number of bytes occupied by the array
  - Compile time operation
  - Thus for strings, it is not an accurate way to see length/number of characters
- If you simply try to add two strings using concatenation I.e. s1 + s2, it will not work as planned as C stores a pointer to the address of the first element/char in each variable
  - It would just add the two pointers
- strlen()
  - **size_t strlen(const char *s)**
  - Turns the number of characters in the string s up to but not including the null character
  - size_t → unsigned integer type; treated as an integer
  - **Must include **#include <string.h>**

- Copying Strings
  - When we copy a string we overwrite what was previously there
  - When we concatenate two strings, we add one string to the end of the other
  - **char *strcpy(char *s1, const char *s2);**
  - **strcpy()** → copies the characters from string s2 into the beginning of s1, s1 doesn't need to be a string with strcpy() is called
    - *s2 is required to be a string
      - Must be a string literal or a char array with a null terminator
    - strcpy() is an unsafe function → if the first parameter has a smaller size than is contained in s2, the result is unknown based on the complier
      - Can result in an error: Abort trap
      - Can also not raise an error and appear to be fine, but may have a deeper hidden bug
    - **char *strncpy(char *s1, const char *s2, int n);**
      - n parameter indicates maximum number of character that s1 can holds, that means max amount of character including null that can be copied from s2
      - This code is also unsafe because it is not guaranteed to add a null character unless it finds one in the first few characters of s2
      - To be safe: explicitly add the null terminator to the end of s1
- String Concatenation
  - Also called appending to a string
  - **char *strcat(char *s1, const char *s2);**

        - Adds the characters from s2 to the end of s1, both (s1 and s2) are required to be strings.
        - If you want strings to be added cleanly, you must add a space manually, as strcat() does not recognize and do this for you
        - Just copies each character from s2, starting from s1's first null terminator
        - Is an unsafe function → may not have enough space to store it's contents along with s2's
    - **char *strncat(char *s1, const char *s2l int n);**
        - Safe version of strcat()
        - n parameter indicates max number of characters not including the null terminator that should be copied from s2 to the end of s1
            - Always adds a null terminator to s1!
            - An example of setting n would be: **sizeof(s1) – strlen(s1) – 1**
- Searching with Strings
    - **char *strchr(const char *s, int c)**
        - s is the string to search, c is the character to search for
        - Searches from left to right
        - Returns a pointer to the character found, or null if nothing was found
    - To find an index, pointer subtraction is used**, (pointer holding farther index – pointer holding earlier/first index)**
        - Must be in the same array
    - **char *strstr(const char *s1, const char *s2)**
        - Searches left to right in s1 for the first occurance of s2
        - Both s1 and s2 are strings
        - If the character is found strstr() returns a pointer to the character of s1 that begins the match with first character of s2

Structures and Union

- Using Structs in Functions
    - Structs or Structures are used to aggregate data when the values are not necessarily all the same type

- Important differences between structs and arrays

	array	structure
data of same type	yes	not required
declaration details	type and number of elements (array [] notation)	types of members (struct keyword)
access via ...	index notation	dot notation

  - Members are specifically used in structs
  - Declaring an array requires subscript or [] notation
  - Declaring a struct requires listing the names and types of all the members
- Building a struct, example:

```
struct student {
        // Members of struct student:
        char first_name[20];
        char last_name[20];
        int year;
        float gpa;
}
struct student good_student;
```

  The word "student" is a structure tag → gives the struct a name
  Can now create variables with the type **struct student** → each with its own members
  good_student has 4 members that belong to it
- To set string variables with less than 20 characters in a struct use:

  Ex.
```
strcpy(good_student.first_name, "Jo");
strcpy(good_student.last_name, "Smith");
```
- Int variables are set as usual
- Structs aren't passed to functions in the same way arrays are (even if the struct is a global variable) → recall: arrays get passed as pointers to their first element
- Any change a function makes to a passed struct, is done on a copy
- Functions get a copy of an entire struct including arrays
- You can fix this copy problem by retuning the struct back to the caller array, but this is ugly as it is sent back and forth
  - Better to work directly on the struct
  - Pass a pointer to the struct as a parameter

- Pointers to Structs

  Ex.
```
struct student *p;
p = &s;
(*p).gpa = 3.8;
```
  - C also has -> as an alternate to dereferencing and then dot notation for structs

Ex. p->year = 1

CSC209 – Week 5: Streams, Files, I/O, and Compiling

Stream

- Introduction
  - In addition to reading from the keyboard we may want to read from the disk
    - Ex. Read from a webpage for data to store in a spreadsheet file
  - C's input and output files read from streams
  - Input stream is a source of data that provides input to the program
  - Output stream is a destination for data output for a program
  - When using scanf() we are reading from a stream called Standard Input
  - By default any program will be set to read from the keyboard
    - This is why scanf reads keyboard input
  - When using printf to write to the screen, what actually happens is that data is written to standard output
    - Standard output automatically defaults to refer to your screen
  - Two streams are automatically open and available when a program runs:
    - Standard input
    - Standard output
    - There is a third stream that is automatically open too → standard error
      - Is another output stream, like standard output
      - The default behaviour is that standard error also refers to your screen.
  - Programmers typically use standard output for normal program output and standard error for error output
    - Their default location is the same there is a way to change the location of any stream
    - You might want the normal output for your program to be saved in a file but any error messages to appear on your screen
    - Recall that printf() can be used to produce output on standard output
      - Cannot be told to write anywhere else except for standard output
    - There is a related function, fprintf, that sends its output to the stream that you specify
  - These three streams have names that can be used to refer to them
    - Using their names is sometimes required when using C input/output functions
  - Streams can also disk files, network connections, and computer devices

- Redirection
  - It is possible to change, or _redirect_, a stream when a program is executed

- Standard input refers to the keyboard by default
  - Can change this using input redirection
- Input redirection → instead of retrieving standard input from the keyboard, redirecting to a file
  - Use readint to read from file by using the less than symbol (<)
  - Example command line:
    ./program < input.txt
- We can also perform Output Redirection
  - Causes the output of a program to go into a file rather than the screen (i.e. a printf() statement)
  - Redirect to a file using the greater than symbol (>)
  - Example command line:
    ./program > output.txt
  - If you use redirection on a file that already contains information/words, the file will be overwritten!
- Input and Output redirections are features of your computer's OS
- Input and output redirection limitations
  - Only one file can be used for input or output redirection
  - It is impossible, to read from two different files by redirecting standard input
    - If you do want to do this, require features of C files

Files

- An Introduction to Using Files
  - We need a way to store values that will last between executions of our program → FILES!
  - Use the fopen() function to open a file and make it available as a stream
    - ***fopen(const char *filename, const char *mode)**
    - Takes two arguments, the file name to open and how we want to use the file
    - The second argument requires a "mode string" that indicates whether we'd like to…
      - Read from an file: **"r"**
      - Writes from the beginning of a file: **"w"**
        - If the file already contains data, empties it before writing
      - Writes from the end of a file: **"a"**
    - The file must exist if the file is being read from
    - If the file is being written to, an empty file will be created if it does not already exist
  - fopen() returns a file pointer that we will use when we want to close the file, read from the file, or write to the file; referring to a file on the computer, after we open a file, we access that file through its file pointer.
  - Ex**. fopen("top10.txt" "r");**

        - Refers to a file that is in the same directory as our executable.
        - By entering a path to a file, we can access a file anywhere on our machine
    - fopen() may fail for a variety of reasons
        - The file might not exist
        - Might not have the proper permissions to open the file
    - If fopen fails, then it returns NULL instead of a valid file pointer
        - Output an error message to standard error (**stderr**), not standard output
        - Exit the program and return a value other than 0 to show that the program terminated with an error
    - The typical pattern when dealing with files is to:
        - Open a file
        - Read from or write to the file
        - Close the file
    - To call fclose, we pass it a file pointer that was previously returned by fopen()
        - **Int fclose(FILE *stream)**
        - Returns 0 if the fclose call is successful, and a nonzero value if the fclose call failed
        - Could fail, if you call fclose with an invalid file pointer
- Reading From Files
    - When reading text or complete lines of data, fgets() is often the right choice of function
        - **char *fgets(char *s, int n, FILE *stream);**
        - Takes 3 parameters:
        - The first argument, **s**, is a pointer to memory where text can be stored
        - The second argument, **n**, is the maximum number of characters that fgets is allowed to put in s, including a null character at the end of the string
            - fgets() reads a mot n-1 characters and then ends the string with a null character
        - The third argument, **stream**, is the source of the data -- where fgets acquires its input
    - On success, fgets returns s, or signals an error by returning NULL
    - fgets() always stops reading when it reaches the end of a line of text in the stream
    - Define a maximum line length
        - With no line in the file that has max characters, so can always read one full line at a time
        - If a line in the file had more than 80 characters, then it would end up being split across multiple fgets() calls

        - To define a char array that will hold each line of the file, add 1 to the max length to account for the null terminator
        - Incidentally, fgets and other file-reading functions can also be used to read from standard input
        - fgets() is often more useful than scanf() for reading strings from the keyboard
        - Why? Because scanf stops reading at a space character whereas fgets stops reading after the newline character that ends a line
        - When using standard input for fgets(), the third parameter is: **stdin**
- The scanf Function
    - fscanf() and scanf() are very similar, only difference between them is that scanf is forced to read from standard input, whereas fscanf can read from any stream.
    - **int fscanf(FILE *stream, const char *format, …)**
        - Similar to scanf prototype
        - fscanf has a new argument giving the stream from which to read
        - Returns the number of items successfully read
            - The number of items will be equal to the number of format specifiers, unless something goes wrong when reading
        - fscanf stops reading a string at a space character
    - Can use fscanf to pick apart pieces of the line of a file and store them in separate variables
    - In this example (name with score)
        - **fscanf(scores_file, "%80s %d", name, &total)**
        - First, we must include two format specifiers because we want to read two fields
        - The first specifier tells fscanf to read a string of at most 80 characters
            - Space for 81 characters (80 + null terminator)
        - Second specifier, we have a space and second format specifier **%d**, indicating that we want to read a space and then an integer from each line
        - The second important thing is the lack of "&" in front of name
            - Recall that the name of an array is interpreted as a pointer to its first element, so name is already a char pointer
            - Using &name would be incorrect: that would be a pointer to a pointer to char
            - We DO use "&" in front of total, since total is an integer, and we need a pointer to the location where fscanf should store the integer
        - If fscanf is the stream-reading version of scanf, why don't we have a gets function related to fgets? In fact, there is.

- However, like strcpy or strcat, **gets()** is an unsafe function and should not be used in practice**
- **gets()** does not let you specify the maximum number of characters to read, so user input can cause your program to write past the end of the memory allocated for a string
    - Can lead to hard-to-find bugs in your programs

- Writing to Files
    - Whenever we intend to write into a file, then we must use 'w' or 'a' as the mode when calling fopen
        - Example: **output_file = fopen("myfile.txt", "w")**
        - If the file already exists, then its contents will be deleted as part of the fopen call
        - We could also use 'a' if we wanted to append to the end of a file rather than deleting its contents
    - printf() writes output to standard out, there is a related fprintf()that is very similar except that it additionally allows you to specify the stream where the output should go
        - Uses the same format specifiers as printf()
        - Calls look just like printf() calls, except that each call includes the output_file stream as the first argument
        - When said that fprintf caused each string to be written to the stream, we very carefully didn't say that the file was being written
            - Writing the file is the desired result, but your operating system is involved
            - When a program writes to a stream, it's actually writing to a location in memory controlled by the OS
                - That memory is periodically written to the file on disk -- when, exactly, the write-backs occur depends on how the stream is set up
                - In almost every case, the contents of the file will be what you've written
            - If your computer loses power in the middle of your program? The short answer is: we don't know
                - That's one reason why we don't recommend that you use I/O for debugging
                - Abnormal program crashes or terminations, may result in fprintf and printf statements that were executed, but may not actually be written to disk, or to the screen
        - If you absolutely need to make sure that key modifications have been made to a stream you're writing, then you can "flush" the stream
            - **int fflush(FILE *stream)**

- Requests that the operating system write any changes that are in its buffer

- Putting It All Together
  - For example, op ten scores file, and want to create a new file that contains only the names, not the scores, with one score per line
    - Can use a combination of fscanf and fprintf

```
#include <stdio.h>

int main() {
FILE *scores_file, *output_file;
int error, total;
char name[81];
scores_file = fopen("top10.txt", "r");

if (scores_file == NULL) {
        fprintf(stderr, "Error opening input file\n");
        return 1;
}
output_file = fopen("names.txt", "w");

if (output_file == NULL) {
        fprintf(stderr, "Error opening output file\n");
        return 1;
}

while (fscanf(scores_file, "%80s %d", name, &total) == 2) {
        printf("Name: %s. Score: %d.\n", name, total);
        fprintf(output_file, "%s\n", name);
}
error = fclose(scores_file);

if (error != 0) {
        fprintf(stderr, "fclose failed on input file\n");
        return 1;
}
error = fclose(output_file);

if (error != 0) {
        fprintf(stderr, "fclose failed on output file\n");
        return 1;
}
return 0;
}
```

Low-Level I/O

- Introducing Binary Files
  - Until now, we have been using text files when performing file input or output, but not all files contain text
    - All files ultimately contain binary data, but in text files, each of these bytes can be interpreted as a human-readable text
  - If you open a binary file like a compiled C program, you will notice that it is not human-readable
    - It will be displayed as junk, since it is not intended to be read
  - Why would we want to store files in this binary format, rather than a readable text format?
    - One major reason is that there is often no convenient way of storing something as text
    - Ex. image files like jpg or gif, or music files like wav or mp3, have no obvious text in them. They hold numeric representations of pictures or music - there is no reason to invent some way to display these files as text
    - We may choose to store data in a binary format when…
      - the intended consumer is a computer, rather than a human
      - if we are concerned about the size of the file, since storing data in binary format typically leads to smaller files
    - OVERALL: binary files are smaller and more versatile than text files
  - Binary files are not directly viewable or editable, so in applications where you want humans to view or edit the content, a text format is appropriate
  - To open a binary file, we include a letter b in the mode that we pass to fopen
    - Binary files come with many different extensions -- no extension at all, "dat", "jpg", "mp3", etc., which tells us how we should interpret the contents
    - Now that the binary file is open, we will want to read and write the file
    - The data in a binary file is not broken up into lines as is the case for text files****
- Writing Binary files
  - The function for writing binary data to a file is fwrite()
    - **size_t fwrite(const void *str, size_t size, size_t nmemb, FILE *stream)**
    - Four parameters…
      - First is a pointer to the data that we want to write to the file
        - Typically the starting address of an array, but can also be a pointer to an individual variable

- Second parameter is the size of each element that we're writing to the file
- Third parameter is the number of elements that we are writing to the file
  - For an individual variable, the parameter is one
  - For an array, it's the number of elements in the array
- Final parameter is the file pointer to which we will write
  - Must refer to a stream open in binary mode
- **Note that this ordering of arguments is different from the ordering with text files.

▪ fwrite() returns the number of elements successfully written to the file, or 0 on error

- Using fopen() on a binary file:
  - If the file does not exist, then fopen will create it
  - If the file does exist, then fopen will destroy its contents before continuing
  - Exactly what happens with text files
- If 3 lines/characters/elements of an array are written to a binary file, the sum of the returned values from fwrite() should be 3
- Close the binary file, just as we did for a text file
- It turns out that character data is human-readable whether the file is a text file or binary file
  - But everything else -- integers, floating-point numbers, and so on -- have different text and binary representations
- When sending an array to fwrite(), include the array name rather than the &name, which only points to the first element

- Reading Binary Files
  - fread() → a function for reading binary data from files
    - **size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);**
    - Takes four parameters:
      - First is a pointer to the memory where the data from the file will be stored
      - The second is the size of one element
      - The third is the number of elements to read
      - The fourth is the stream to read from
    - There is only one difference in the fwrite and fread prototypes:
      - Whereas fwrite takes a constant pointer as its first argument, fread takes a non-constant pointer
      - Why? fwrite is writing data from memory to the file, so the memory does not need to be modified but, fread is writing data from the file to memory, so the parameter cannot be const

        - fread() returns the number of elements successfully read from the file
            - If returns 0, then no elements were successfully read (signals an error or end of file)
    - It's important that the order in which we read values is the same order that we wrote them to the file
    - Might expect that you can use fwrite to create a binary file, and then use fread on a different computer to read that data back, BUT, this isn't guaranteed to work
    - Different computers may represent data in different ways. In this case, we're seeing the effective of **endianness**
        - Little endian
        - Big endian
    - When you are transferring binary data between machines, you need to be aware of how the computer represents data internally.

- Putting it Together: wav Files
    - Wav files are similar to mp3's but their structure is much simpler
        - A wav file has two parts:
            - The header, which is 44 bytes of data
                - Contains information about the wav file, including parameters required to properly play the file in a music program.
            - One or more 2-byte values, each value is called a sample
                - When a wav file is created, the audio signal is sampled many thousands of times per second
                - Each of these samples is stored as one of these two-byte integers
                - Each sample if of the data type "short"
        - Since wav files are binary files, viewing them in a text editor will not be productive
            - One useful utility is od, which prints out the values found in a binary file → file viewer for binary files
    - We will use od to print the contents of the example "short.wav"
        - We will need to give it several commandline options
        - **-A d**: translates od's output from the default of base 8 to a more convenient base 10
        - **-j 44**: causes od to skip the first 44 bytes of the file
            - We're skipping these bytes because the header occupies those first 44 bytes
        - **-t d2**: tells od that the file consists of two-byte values
        - Last argument is the filename

    - Results: In the leftmost column, od provides the file position for the bytes given in that line
    - For example:
      0000044 2 2 2 2 2 8 8 8
      - The leftmost column says 44, which means that we're starting at byte 44 of the file
      - The rest of that line -- 2, 2, 2, 2, 2, 8, and so on - are the two-byte integers found at bytes 44, 46, 48, 50, 52, 54, and so on.
    - The last line in the example shows us that we end at byte 84 which means that the entire file is 84 bytes.
- Increasing the Sound in wav Files
  - The first file is opened for reading; the second is opened for writing
    - It's especially important that we check the return value of these fopen calls, since we are using user input as a parameter
  - Read the header from the input file and write the header to the output file
    - Use fread and fwrite to copy the header from our source file to the new audio file
  - To increase the volume of the sound in a wav file, we must multiply each of the samples by a number greater than 1
  - We need to access each sample individually, so we use a while-loop that reads one short integer at a time from the input file
    - If fread does not return 1, then we have reached the end of the file
    - Inside the loop, we multiply our sample value by 4 to make it four times as loud as the original, and then we write the new sample to the output file using fwrite.
  - Finally, we close both files to complete the program. Now let's compile and run our program and use od to test the results
  - When running od, each sample in short_loud.wav is four times as large as the corresponding sample in short.wav
- Learn About Structs
  - There's one more popular use of fread and fwrite, however, and that is for reading and writing structures (structs)
  - Using fwrite() we are going to write some student structs to a file, so we declare a file pointer and then open a binary file called student_data for writing
  - Most of the parameters are the same as when we were writing individual values but the interesting parameter is the second one, where we use sizeof to determine the size of the struct that we are writing
    - Example: sizeof(struct student)
    - It's very important we used sizeof instead of manually computing the value of the struct

        - It seems possible for us to compute the struct size: we know that the first name is allocated 14 bytes and the last name is allocated 20 bytes. Most likely, the year will be 4 bytes, and the GPA will be 4 bytes. So perhaps the entire struct is 14+20+4+4 = 42 bytes
            - We are guessing at the size of the int and float
            - Even if we knew those sizes, we wouldn't know the size of the struct because C is allowed to insert spaces between memebers of a stuct!!!
        - Moving the members a little in memory can speed up memory accesses -- and it may even be required by the computer running the code
        - Ex. most machines require that integers start on an address that is a multiple of four, so the compiler will add two bytes of padding after the characters to make this happen
    - Notice that if our first name is allowed to be up to 13 characters (plus a null terminator), the name doesn't always take up the full space
    - For example, Betty is only 5 characters plus a null terminator
        - What happens? fwrite places all 14 bytes of the first name field into the file, even though we only use the first 6!
- Moving Around in Files
    - Sometimes, we need to jump around in a file, or perhaps we'd like to read some part of a file more than once or to skip parts of a file
    - Let's start with a function that allows us to move around in a file → fseek()
        - **Int fseek(FILE *stream, long int offset, int whence);**
        - Remember that every open file maintains its current position, which determines, where the next call of fread will read from or where the next call of fwrite will write to
        - Each read or write call moves the file position
        - fseek allows us to change that file position
        - Takes three parameters:
            - The first is the stream whose position we'd like to change
            - The second is a byte count indicating how much the file position should change
            - The third parameter determines how the second parameter is interpreted

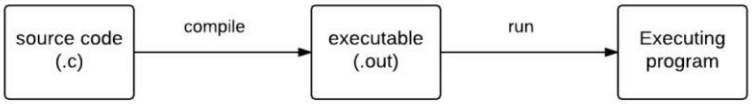
constant	meaning
SEEK_SET	from the beginning of the file
SEEK_CUR	from current file position
SEEK_END	from end of file

            - When moving backwards, use negative integers as the second parameter

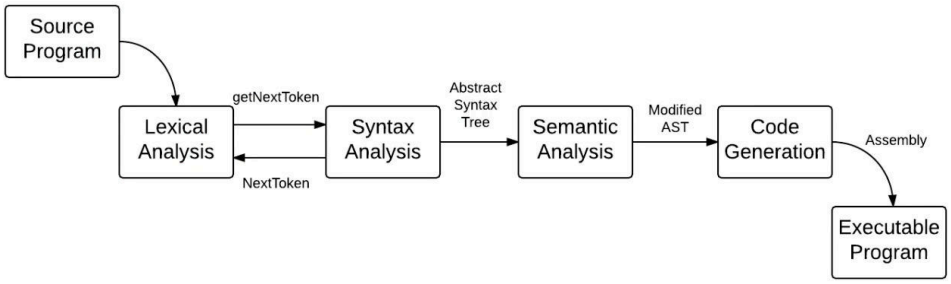
- For example, if we try to set the file position to byte 50 and the file has less than 50 bytes, what will happen?
  - In situations like this, the fseek call will succeed but a later read or write will fail, so it's particularly important to check for errors in your read and write operations after using fseek
- fseek itself can fail if we pass an invalid file pointer, but this can be avoided as long as you check your call to open
- When using fseek() to find a struct in an array of struct make sure to multiple the index by the sizeof(struct)
  - If we don't do that, then we will end up moving the file position to a particular byte -- not to the beginning of a struct
- If you wanted to read the entire file again. You can do that using fseek, by supplying a value of 0 for the second parameter and SEEK_SET as the third parameter
  - OR function rewind can be used to move to the beginning of a file
  - **void rewind(FILE *stream)**

Compiling

- The Compiler Toolchain
  - Compiler toolchain → set of applications that let you transform source code into a running program
  - The compilation process happens in a single step



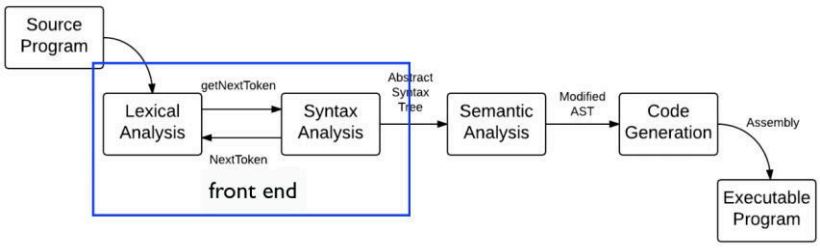
  - Recall: gcc → creates an executable for specified program called “a.out”
    - When a.out is executed, the program is executed
  - The complier isn’t a single program
  - The gcc complier has to work on many different systems, running on processors from different companies that speak different languages
  - While gcc compiles C, the gnu compiler project supports many different languages
  - Complier → any program that translates code in one language to a different language



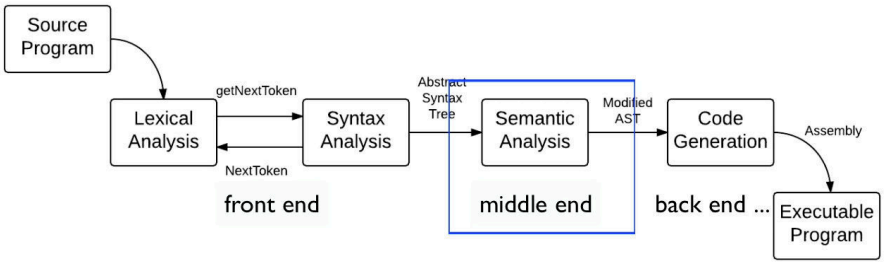
    - Accepting input from a high-level language and producing output in lower a language (like assembly code)

- You can run just this part of gcc using:
  - **gcc –S program.c**
- Result: Is assembly code
  - Assembly code → human readable language that represents the instructions that a computer actually runs

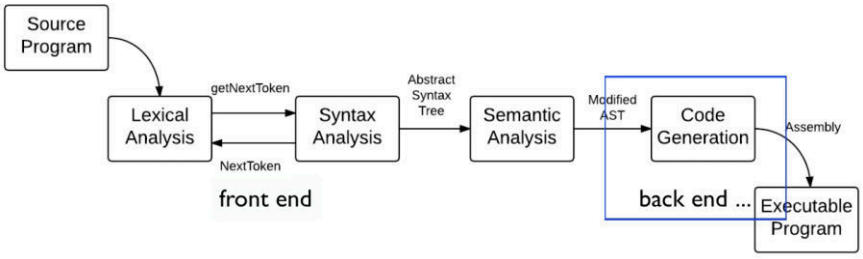
- Complier runs in 3 phases:
  - Front end that translates source code into largely language independent intermediate representation



    - gcc translates all languages into gimple and generic
    - Abstract syntax trees (from independent representation) or graph structures, where each node is an element of the program language into the assembly language that will run the program
  - Middle end, the complier optimizes the code, to make it run faster
    - SOME optimizations also happen in the front and back ends



  - Back end that translates the intermediate language into the assembly language of the computer that will run the program



  - **The toolchain in this example is idealized

- We need to assemble the assembly code into object code to produce an executable (.out) file by the assembler, by using the command **-as**

        - as program.s –o helloworld.o
        - After this command line, helloworld is an object file and is no longer human readable
        - This cannot be run
    - Need one more step to create an executable
        - Running gcc on helloworld.s produces a.out
        - Assembles code and invokes the final step to produce an executable
    - Using the **file** command we can see the format of the files and confirm that gcc is invoking more than the assembler, as it generates an executable file rather than an object
    - Final step in compilation process is linking → the linker takes one of more compiled and assembled object files and combines them to create a file in executable format
        - The final executable file is a package that contains all of the instructions in the program as well as a data section containing items like constants strings and links to dynamic libraries (which contains functions)
        - This executable program is not portable → you cant copy it to another machine and know it will run
            - Everything from the assembly code to the executable is made for a specific machine
            - Executable is specific to not only the type of machine but also the OS and even system configuration
    - The executable needs to be loaded into memory when you execute it → job of the loader (which is OS specific)
        - Before the complier, need a pre-processor to prepare the source code for the front end
        - Pre-processor → adds function prototypes for lib and system calls required for the program and interprets pre-processor directives like micros
        - Anything with a # has to be cleaned up by the pre-processor
- Header Files
    - What happens when you compile a program with multiple source files?
        - If you try to only compile the main file that implements the others, you will get a linkage error
        - Can compile all the files in one line
        - Ex. **gcc compare_sorts sorts.c**
    - How does gcc compile multiple files in one line?
        - When multiple files are required, each file is compiled and assembled separately and the object files are combined during the linking stage to produce an executable
    - Can also compile multiple files using: separate compilation
        - Ex.

gcc –c compare_sorts.c
gcc –c sorts.c
ls *.o
gcc compare_sorts.o sorts.o

- Separately generated object files for two source files and then linked them
- Good for large projects with many files when you want to make a change in a single file → can just recreate the single object file for the source code file that was changed, then relink
- Has some issues, when types are incompatible between files, and the source files are compiled separately, a warning does not appear → values are mis-sorted
  - Complier only checks for consistency in each of the files to itself rather than with other files in this case
- We can work around these issues using header files

- Header files → example of an interface
  - Declares what functions do and what types they require without explaining how they are implemented
  - Can think of it as declaring a design and the .c files as implementing that design
  - Place in prototypes and function types
  - **#include** the header file
    - Tells the pre-processor to insert the body of the header into the source code before the compiling begins
    - Ex. **#include "sorts.h"**
    - Using quotes rather than angled brackets to tell the complier to use the header file in the current directory rather than one in the system library
  - Will get an error if the header file is #included but the parameters in the prototypes don't match between the implementation and the header
    - Mis-match error
- You do not supply the name of the header file to gcc

- Header File Variables
  - When you include the declaration of a variable in the header file and include it in multiple files, it creates space for this variable in all of the object files
    - The compiler sees that each object file has a copy of this variable
    - Need to separate the declaration of the variable's name and type from the definition that creates it
  - Definition should be in source file where the functions are being defined
    - If another file needs to know the variables exist, the variables must still be declared in the header file

- Remove the assignment of values to the variables and declare them to be → **extern**
  - Stands for "externally defined"
  - Creates the name of the variables so that each source file can compile separately without actually defining space for the variables
- What happens if you create a global variable with the same name in multiple files (ex. int x) → you will get a duplicate symbol error
  - You may WANT x in both source files as it is a part of the implementation rather than the design
  - Then, use the static key word
    - By default, symbol names in C are externally visible aka available globally everywhere in the program
    - Static makes the variable name local → available only in the file that defined it
- Static keyword has many related meanings
  - Denotes global variable cannot be visible outside of the file in which it is defined
  - If static is used inside of a local rather than global, there is a different meaning → the variable should keep its value across function executions
  - So the first time the variable is declared/set, it stays at that value even if it is "re-declared" again to its first value
    - It can be updated however
- Guard condition: goal is to keep the header file from being included more than once in any compilation
  - A multiple included header will generate errors like duplicate symbols
  - Ex.
    #ifndef SORTS_H
    #define SORTS_H
    etc**.**
    #endif
  - #**ifndef** →if statement, "if not defined SORTS_H"
  - #**define** SORTS_H → definition is executed
    - The first time the header is encountered, SORTS_H does not exist
    - If the header file is included again, the condition is true and the contents of the header file will not be processed
  - #**endif** → telling the compiler where the body of the if statement ends
    - Add a comment to specific which if is being terminated

- After this you must compile the source files even though they are not changed
  - The header file that is included is changed → a dependency
  - Source code relied on the header file
- Complier toolchain has a tool called make to help track these dependencies and check which files needs to be recompiled

- Makefiles
  - Make is tool of the compiler toolchain that helps manage the process of multiple source code/object files with header files to track
  - Make files are composed of a sequence of rules

    target: dependencies…

    recipe
    - Each rules have a target → the file to be constructed
    - Each also have a recipe → the command or list of commands to execute the creation of a target
    - If dependencies/prerequisites are present, the recipes are executes if one or more of the dependencies are newer than the target
      - If none, the actions are only executed if the target doesn't exist
    - The white space before recipe is a tab
    - Example:

      compare_sorts: compare_sorts.c sorts.c

      gcc compare_sorts.c sorts.c –o compare_sorts
  - The command: make → looks for a file named "Makefile" in the local directory and looks for the rules it contains
    - In the example, the rule doesn't account for the header file
    - So make doesn't detect when it needs to remake the executable when the header is updated
    - Also aren't taking advantage of separate compilation with this makefile rule
    - Updated example:

      compare_sorts.o: compare_sorts.c sorts.h

      gcc –c compare_sorts.c sorts.c –o compare_sorts

      sorts.o: sorts.c sorts.h

      gcc –c sorts.c –sorts.o

      compare_sorts: compare_sorts.o sorts.o

      gcc compare_sorts.o sorts.o –o compare_sorts
    - Building rules for the object files
      - Each object file depends on the relevant source file and the header file

- When you run make with no command line arguments, it evaluates only the first rule in the file
  - If you specify a specific target, it will evaluate that rule
- Power of make: each time it evaluates a rule, it will first check all the dependencies
  - If the dependency is also a target in the make file, it will evaluate that rule first before checking the dependencies

- Make will execute the recipe if the target is older than the dependencies
  - Only the files that need to be updated are compiled
- Make files support wild cards of various sorts
- New rule replaces all source files to object file rules (example:)

%.o %.c sorts.h
gcc –c $< -o $@

compare_sorts: compare_sorts.o sorts.o
gcc compare_sorts.o sorts.o –o compare_sorts

- **%.o %.c** → each object file that needs ot be built depends on a source file of the same name
- **$<** → variable containing the first name in the list of dependencies
- **$@** → variable containing the name of the target
- All together this rule checks any required object file and builds it based on a similarly named source file

- Another rule to clean up (example:)

%.o %.c sorts.h
gcc –c $< -o $@

compare_sorts: compare_sorts.o sorts.o
gcc compare_sorts.o sorts.o –o compare_sorts

.PHONY: clean
clean:
rm compare_sorts *.o

- **.PHONY** → indicates that clean isn't actually a file, it is just a legal target
- Building the target clean executes the remove command
- The next make rebuilds all the required files because clean is not a file
  - The actions are executed when the clean rule is evaluated

- Another improvement (example:)

OBJFILES = compare_sorts.o sorts.o

```
%.o %.c sorts.h
        gcc –c $< -o $@

compare_sorts: $(OBJFILES)
gcc $(OBJFILES) –o compare_sorts

.PHONY: clean
clean:
        rm compare_sorts *.o
```

- Make files can include variables
- This new line defines a variable called **OBJFILES**
  - Instead of needing to add to the list of object files everywhere
  - Can just updated the variable when a new source file is added to the project

CSC209 – Week 6

Useful C Features

- Typedef
  - 2 C language features that allow you to define aliases for types
    - Typedef → allows you to define an aliases for a type and is evaluated at compile time
    - Macros → allows you to define a key word that is replaced by a specified string when your program is processed before being compiled
  - size_t → type of malloc's parameter and return type of sizeof()
    - Usually found in the C standard libarary file standard def.h
  - **typedef usigned int size_t;**
    - Declare size_t to be an unsigned int
    - typedef is the keyword → allow you to define a name, that refers to an existing type
      - Now any function that includes the type size_t will result in unsigned int values
      - This existing type can easily be changed based on preference
    - Doesn't create a new type in the sense that structs to
      - Not creating a new composite type with components with their own names
    - Providing a new name for an existing type
  - If a variable is passed to a function with a differently named typedef parameter with the same type (ex. unsigned int age_t vs. unsigned int shoe_size_t) the compiler won't catch this
    - Can't rely on the compiler to point out these errors
    - Any external users must find the typedefs so they know how to use the declared types
  - Typedef's make structs easier to read, creating an alias for our struct

```
typedef struct student {
   char first_name[20];
   char last_name[20];
   int year;
   float gpa;
 };
 typedef struct student Student;
```

THEN can declare structs like…

```
Student s;
```

Student *p;

- Macros
  - Used to replace constant values to evaluate simple expressions
  - The directive for a macro starts with #define
    - Ex. #define MAX_NAME_LENGTH 40
    - Tells the pre-processor to replace the occurrences of MAX_NAME_LENGTH with 40
  - Macro is evaluated during pre-processing, just like #include
  - The Macro language isn't C, so you don't need an equal sign or semi colon when defining your variables
    - If you do include an "=" it too will be included in the contents of the variable
    - Will get an expected expression error
  - Macro's can take parameters
    - Ex.
      #define TAX_RATE .80
      #define WITH_TAX(x) ((x) * 1.08)
      THEN with the code
      double purchase = 9.99;
      printf("%f \n", WITH_TAX(purchase));
    - Works as a "mini function", but happens even before compilation
    - Some rules:
      - No space between the macro name and the opening parentheses
        - A space will be interpreted as having no parameters
        - Replaces WITH_TAX with the rest of the line including the parenthesized x that was supposed to be the parameter
      - Should place the parameter variable as well as the expression within parentheses
        - If this isn't done, any arithmetic done in the "function" call will not be done first and will be done according to rules of BEDMAS

Function Pointers

- Introducing Function Pointers
  - Functions can be treated as data, passed as arguments or used in structs
    - Function pointers
  - clock() → returns the amount of CPU time used so far
  - Problem: in this example, need to change the source code every time a different sort-type is explored

```
void check_sort(int *arr, int size) {
     for (int i = 1; i < size; i++) {
```

```
            if (arr[i - 1] > arr[i]) {
                printf("Mis-sorted at index %d\n", i);
                return;
            }
        }
    }
    // fill arr with random values
     void random_init(int *arr, int size) {
        for (int i = 0; i < size; i++) {
            arr[i] = rand();
        }
    }
```

- Both of the initialization functions have the same signature → they have the same return type and have the same number and type of arguments, therefore have the same type
- What is the type of a function? void …(int *, int)
- Declaring a variable of type pointer to a function: **void (*func_name)(int*, int)**

```
void check_sort(int *arr, int size) {
    for (int i = 1; i < size; i++) {
        if (arr[i - 1] > arr[i]) {
            printf("Mis-sorted at index %d\n", i);
            return;         }
    }
 }
void random_init(int *arr, int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = rand();
    }
 }
// Initialize the values in arr to be in decreasing order.
// This is the reverse of what our sorting function will provide.
void max_to_min_init(int *arr, int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = size - i;
    }
 }
// Now our time_sort takes both the sorting function and the
initializer
// function as parameters. This function does not need to be
recompiled
// to change these values.
double time_sort(int size, void (*sort_func)(int *, int), void
(*initializer)(int *, int)) {
    int arr[size];
    // We are calling the function received as a parameter.
    initializer(arr, size);
    clock_t begin = clock();
    // We are calling the function received as a parameter.
    sort_func(arr, size);
    clock_t end = clock();
```

```
    check_sort(arr, size);
    return (double)(end - begin) / CLOCKS_PER_SEC;
}
```

- You do not have to dereference the pointer
- When calling the time_sort function, can just use the name of the other function like it is a regular variable name
- WITHIN time_sort the function can be referred to by using the initialized name – in this case it is "initializer"

System Calls

- What is a System Call?
  - A function that requests a service from the operating system
  - **void exit(int status);**
    - exit() a request to the OS to terminate the program
    - When called the OS executes instructions to clean up the data structures that represent the running process and terminate the process
  - Low-level IO calls read() and write() are system calls
    - A lot of higher level IO calls use these system calls in their implementation
  - Library function → executes many helper functions before calling system calls
    - i.e. printf() uses write()
  - Calling a library function is similar/identical to calling a function you have written
  - When a system call occurs, control is given to the OS and the OS executes code on behalf of the program
  - System calls:
    - exit()
    - read()
    - write()
  - Library functions:
    - fgets()
    - fopen()
    - printf()
    - scanf()

Errors and Errno

- When System Calls Fail
  - What happens when a system call doesn't work correctly?
  - Because system calls interact with resources outside the running program, it is possible for them to fail

- The reasons for a system call failing might not be something the program itself has any control over
- When using fopen to open a file where the file name is given as a command line argument, the file might not exist or might have the wrong permissions
  - With a filename that doesn't exist, the program crashes
  - Leads to a segmentation fault
- It is the programmer's responsibility to check that a call succeeds before using its results – particularly true for system and library calls
- Here are a few examples of system calls:
  - fopen opens a file and returns a file pointer that will be used for later operations on the file → if it cannot open the file, it returns NULL
  - malloc (library system function) returns a pointer to a region of memory that it has allocated → it it cannot allocate enough memory, it returns NULL.
    - Uses system calls to allocate memory, and it checks those calls and returns appropriate errors
  - stat system call finds the information about the file pointed to by path, and stores that information in buf → returns 0 if it succeeds and -1 if it encounters an error
- The main way to indicate an error occurred is to return a special value, and nearly all system calls follow the same pattern
  - I.e. return -1 to indicate that an error occurred for an integer return type
  - System calls that return a pointer will return a value of NULL to indicate that an error occurred
- The return value itself cannot be used to indicate the error type, so a global variable called errno is used to store the type of the error
  - errno is a global variable of type int
  - When a error occurs in a system call, the system call returns -1 or NULL depending on the return type, *and* sets a value for errno to indicate the type of error
  - For example, if malloc fails, it returns NULL and sets the value for errno to ENOMEM
  - perror, is a commonly used function that will map the error code to a string that explains the error
    - prints a message to standard error
    - The message includes the argument s followed by a colon and then the error message that corresponds to the current value of errno
    - Do NOT use perror as a generic error message reporting function

- Real purpose is to display an error message based on the current value of errno
- Used for cases when the set variables returned from functions signal a failure to complete their specified task → then use this variable as the argument for perror display

▪ For other types of error messages, you should still use fprintf to stderr

- In the case of in error in main, a return will usually terminate the program -- but it's a good habit to use exit()
  - Errors you detect in other functions won't terminate the program if you simply return, rather than invoking exit
- If reading from an empty file, should the error be displayed using printf or perror?
  - Printf shows a customized message
  - Perror would show "undefined error: 0" as errno is not set
    - Errno is set in cases of failure, not in errors or invalid cases

- Checking for Errors
  - Error checking and appropriate error messages are important for several reasons
    - Users of your program need useful feedback if they use your program incorrectly, or if the program fails for any reason
    - Checking for errors helps you discover bugs more quickly
  - If when executing a program with required command-line arguments, suppose we forgot the order of the arguments and tried an incorrect ordering → we get no output
  - Or suppose we run it with no arguments? →We get a segmentation fault
  - argc → can check the value of this to verify the number of command-line arguments
    - NOTE: the executable name is considered to be one of the arguments
  - But what about getting the arguments in the wrong order?
    - In this case since we are using fopen, we should use perror to print the message
  - If a program expects a number as an argument and a string is passed instead, strtol does not do its job and the desired num value is NULL
    - Usually will also result in NULL for negative numbers as the negative symbol is a char and strtol will not handle/convert farther than this
    - Numbers with decimals or trailing/existing characters within the number will only convert up to the character (as strtol does)
      - Can use the second argument of strtol to determine how successful the conversion from string to number was

- In the case where a number it too large for its expected type, strtol will even set errno to ERANGE

- If we tell our program to print more lines than are in the file without checking if we have reached then end of the file, then the last line is repeated
  - If we want to fix this, add a NULL check
- What happens when the length of the line is larger than BUFSIZE (valid line length in the given file)?
  - Two options to test this case
    - Construct a test input file where a line has more than 256 characters
    - Set BUFSIZE to a much smaller value, like 10 (better)
  - FIX: Can check to see if fgets read a newline character, and keep calling fgets until it finally reads a newline character
- You should introduce tests when it makes sense to do so

CSC209 – Week 7: Process Model

Process Models

- “What does it mean to *run a program*?"
- As users, we run programs by typing a command into the shell and hitting return or by double-clicking on an icon
    - Therefore must be possible to write programs - like a shell - that can run other programs.
- Before we can write a program, need to define what *programs* and *processes* are, and we need to understand how the OS works with them
- Program → executable instructions of a program
    - The human-readable source code
    - The compiled machine level code
- Process → running instance of a program
    - Includes the machine code of the program, plus information about the current state of process such as what instruction to execute next, or the current values of variables in memory
- When a program is loaded into memory in preparation for executing the program, the memory layout has a structure that will already look familiar
- The program code, global variables, stack, and heap all have their own regions of memory, and taken together, they represent → state of the program
    - Stack → tells us which function we are executing and holds the current values for any local variables
    - Heap and global variable regions → hold the current values for other variables in the program
    - OS → keeps track of some additional state for a process
        - Each process has a unique identifier, or process ID (PID)
        - Each process is also associated with a data structure called a process control block/task control block
            - PCB → stores the current values of important registers, any open file descriptors, and other state that the operating system manages
            - The registers stored include the stack pointer, which identifies the top of the stack, and the program counter, which identifies the next instruction to be executed
- On Linux, run the program "top" to see some of the active processes on your machine
    - When run can see user level processes running including "top" itself, and operating systems processes running owned by root

- Also gives us some information about the some of the state information the operating system has about a process

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
8	root	20	0	0	0	0	S	0	0.0	12:03.39	rcu_sched
2196	root	39	19	0	0	0	S	0	0.0	47:40.80	kipmi0
13365	reid	20	0	17732	1768	964	R	0	0.0	0:00.54	top
23412	root	20	0	0	0	0	S	0	0.0	0:21.41	kworker/2:0
51571	c4user	24	4	30668	5960	1228	S	0	0.0	8:47.88	tmux
1	root	20	0	24452	2372	1336	S	0	0.0	0:09.76	init

  - A process has a PID or process identifier (number)
    - In the far left column in top
    - The command or program that the process is executing is in the far right column
    - Notice process with PID 1 and the Command "init"
      - This is a special operating system process
- The number of active processes is much larger than the number of processes executing instructions at a particular instant of time
- The number of processors → CPU(s), on the computer determines how many processes can be executing an instruction at the same time
  - 4 CPUS then 4 processes can be run simultaneously
    - → Running state
  - But there are also other processes that could execute if a CPU is available
    - →"Ready" state
  - Finally, there will be other processes that are waiting for an event to occur
    - → Blocked or Sleeping state
    - They might have made a read call and are waiting for the data to arrive, or they might have explicitly called sleep and are waiting for a timer to expire
  - A single process will move from one state to another throughout its lifetime
  - For example, gcc may have been Running, but then needed to access a file. gcc is moved to the Blocked state while it waits for data from the file to arrive. Once the data arrives, gcc is moved to the Ready state. A processor isn't available yet, but when it is, gcc could be executed. Finally, once a processor becomes available, gcc is executed, and it's back in the Running state
- The OS gives us the illusion of running dozens or hundreds of processes simultaneously by switching between Ready and Running processes very quickly
- OS scheduler → responsible for deciding which processes should be executed and when
- Knowing a little about how the operating system schedules processes, and understanding when they block is important when you write programs that create or manage cooperating processes

Creating Processes with Fork

- We will learn how to create a new process using the *fork* system call
- Creating a new process requires a system call
    - The operating system must set up data structures, such as the process control block, needed by the new process
- When a process calls **fork()** → passes control to the operating system and the OS makes a copy of the existing process
    - Returns the process id of the child to the parent
    - Return in the child is 0
    - If fork fails, then the return value of fork in the parent is -1 and the new process is not created
        - Fork might fail if there are already too many running processes for the user or across the whole system.
- To duplicate a process, the OS copies the original process's address space (data), and the process control block that keeps track of the current state of the process
    - As a result, the newly created process is running the same code, has the same values for all variables in memory, and has the same value for program counter and stack pointer. They're almost identical
    - **2 important differences:
        - The newly created process gets a different process id (PID)
        - The return value of fork is different in the original process and the newly created process
    - The original process → parent
    - Forked process → child result
        - Has the same value for the program counter as the original process, so when runs, it starts executing just after fork() returns
    - We don't know whether the parent process or the child process will execute first, after the new process is created, OS can give control to either the parent OR the child
    - Parent and child processes are completely separate processes and don't share memory
- Simple example
    - It starts executing as you would expect by setting i to a value of 5, and then printing 5
    - Then, when it makes the fork call, it is as if time stops while the operating system creates the child process and gives it a new process id (680 in this case)
    - It begins executing at the point when fork returns. The variable i still has the value 5 because the new process is a copy of the old one, but neither the assignment statement nor the printf() statement are executed by the child process

- From the programmer's point of view we now have two processes ready to execute exactly the same code, with the return values used to distinguish between the two
- In this example, the parent process now sets i to be i + 2, or 7, and the child process sets i to i - 2, or 3
- If we run the program, the output is 5 7 3 or 5 3 7

Parent Process
pid == 677

```
int main() {
    int i;
    pid_t result;

    i = 5;
    printf("%d\n", i);

    result = fork();
    /* result == 680 */
    if (result > 0)
       i = i + 2;
    else if (result == 0)
       i = i - 2;
    else
       perror("fork");

    printf("%d\n", i);
    return 0;
}
```

Child Process
pid == 680

```
int main() {
    int i;
    pid_t result;

    i = 5;
    printf("%d\n", i);

    result = fork();
    /* result == 0 */
    if (result > 0)
       i = i + 2;
    else if (result == 0)
       i = i - 2;
    else
       perror("fork");

    printf("%d\n", i);
    return 0;
}
```

- Output of a program with fork:
  - Functions/calls in the parent before fork()
  - Either remaining functions/calls in the parent OR functions/calls in the child
    - This depends on the control that the OS designates
    - Therefore, if both the child and the parent are producing output, the order of the output may be different each time we run the program.

Process Relationship and Termination

- Recall that the programmer can't control whether the child process or parent process executes first after they return from fork (controlled by OS)
- The **usleep()** system call is there to slow the processes down enough to see what can happen when they run concurrently
  - Also introduces more variability in how the processes run because every time usleep is called, control is passed to the OS, which gives the OS opportunity to decide which process will run next
- From the operating system's point of view, the parent process is no different than any of the child processes, so the operating system schedules the parent process to run the same as it does for the child processes
  - There's no reason for it wait for its children to execute, for example
  - Therefore parent prints/calls can appear in the middle of prints/calls of the children

- Normally, when you run a program the shell waits until the process has finished and then prints a prompt for the next command
    - That is happening in the example too
    - The shell waits for the original parent process to finish, and then prints a shell prompt
    - But the shell is just another process that the operating system has to schedule, so a few child processes get to print some output before the shell prints its prompt. Once the shell runs, it prints it message -- but some child processes haven't finished yet, so they print more output afterwards
    - Shell prompt may not appear, but if you press enter, you will see it appear
- When a program ends with no shell prompt → "hanging"
- The example program we ran from the shell is a child of the shell process
    - The shell used the wait system call to suspend itself until its child finished
- The **wait()** system call → forces the parent process to wait until its children have terminated
    - "suspends execution of the calling process until one of its children terminates."
    - Information about how the child terminated - cleanly or with a particular error - is stored in the int passed in using the stat_loc argument
    - If wait returns successfully, its return value is the process id of the child that terminated
    - If wait fails, it returns -1
- If you call wait on the parent process, it will wait until all of the child processes are completed before terminating itself
- The child processes may or may not terminate in the same order as they were created, in our example, they do
- When a program calls **exit()**, , or calls return from the main function, then we provide an argument like -1 or 0
    - This value makes up part of the status value in wait
    - Enables limited communication from the child back to the parent
    - Status of 0 represents a successful run of the process
    - Non-zero values represent various abnormal terminations
    - But the exit value is only part of the value of the status returned by wait
- The status isn't just the argument to the exit function
    - Various bits in the status argument are divided up and used for different purposes
- The lowest 8 bits tell us whether the child process terminated normally, or whether it terminated because it received a signal (like control C)
    - If it terminated due to a signal, the lower 8 bits tell us which signal

    - The exit value of the child process is in the next 8 bits, which is why the process that exited with a 1 has status 256 (2^8) and why the process that exited with a 2 has status 512 (2^8 x 2)
- Extract the values we want from the stat_loc argument, and use the macros correctly. How?
    - First, we need to add an include so that the macros are available for use in our program: **#include <unitstd.h>**
    - Next, we use the macro **WIFEXITED** to check if the process terminated normally
        - If it did, then we use WEXITSTATUS to extract the exit value of the process
    - Otherwise, if a process exits as a result of a signal, then we use WIFSIGNALED and WTERMSIG to find out the number of the signal that caused the process to terminate
- To end a program abnormally, use abort() rather than exit
    - Signal for abort is 6
- If we need a bit more control, the **waitpid()** system call lets you specify which child process to wait for
    - Pass in the process id of a child process or the WNOHANG option to waitpid if the parent process just wants to check if a child has terminated, but doesn't want to block
- But both wait and waitpid have a limitation: they only wait for child processes. You cannot wait for an unrelated process or even a child of a child process
    - Only give us a *limited* form of synchronization between processes

Zombies and Orphans

- Question of what happens when a child process terminates before the parent calls wait
- An example… program the original process calls fork once
    - The parent process sleeps for 20 seconds before it calls wait
    - The child process prints only one line and then exits
    - If we are quick, we can use top() to see the state of the child process before the parent process calls wait
    - The state (column s) of the child process is given as Z and the process is shown as "defunct"
        - The Z stands for → zombie, a good analogy for the state of the process
            - The child process has called exit, so it is dead ... but not quite
        - The operating system needs to keep the exit information of the process somewhere in case the parent calls wait to get this value. So the operating system can't delete the process control block of the terminated process until it knows it is safe to clean it up

        - A zombie process → process that is dead, but is still hanging around for the parent to collect its termination status
- What happens to a child process when its parent terminates?
    - Naturally, we call the child process an "orphan" when its parent terminates first
        - But who is the parent?
        - When a process becomes an orphan, it is "adopted" by the init process (note: has a ppid of 1)
- A zombie is exorcised (put to rest), when its termination status has been collected
    - The main task of the init process as a parent is to call wait in a loop, which collects the termination status of any process that it has adopted
    - After init has collected the termination status of an orphaned process, all of the process's data structures can be deleted, and the zombie disappears.
- Checking processes based on ppid from command-line:
    - **ps –p (id number)**

Running Different Programs

- The parent process can wait for the child process to terminate and collect its exit value
- The next step is to load and execute a different program
- Linux provides the "exec" family of functions to replace the currently running process with a new executable
    - Several variants of exec, but all perform the same task, differ in how the arguments to the function are interpreted
- Example: the do_exec program prints a message and then calls **execl()**
    - **execl(“./hello”, NULL);**
    - The first argument to execl is the path to an executable
        - The name of a compiled program
    - The remaining arguments to execl are the command line arguments to the executable named in the first argument
    - If no commandline arguments, include NULL as the final argument
    - When called, control is passed to the OS
        - When performing execl, the operating system finds the file containing the executable program and loads it into memory where the code segment is
        - It also initializes a new stack - the program counter and stack pointer are updated so that the next instruction to execute when this process returns to user level is the first instruction in the new program
    - When control returns to the user level process, the original code that called exec is *gone*, so it should never return

- execl will return if an error occurs and it wasn't able to load the program
- Remember: the operating system does not create a new process -- that's a job for fork
  - exec asks the operating system to modify the calling process
  - The process has the same process id after exec, and retains some state from the original process such as open file descriptors
- Family of exec functions:
  - l → list, so the command line arguments to the program are passed as a list of arguments to the exec function
  - v → vector, so the command line arguments are passed in as an array of strings, just as we see with argv parameter of main
  - p → path, which means that the PATH environment variable is searched to find the executable
    - Without the p, execl and execv expect that the first argument is a full path to the executable
  - e →environment
    - You can pass in an array of environment variables and their values so that the program executes within a specific enivironment
- How the shell executes the commands we enter:
  - The shell is just a process, and it uses fork and exec just as we have described in these videos
  - When you type a command at a shell prompt, the shell first calls fork to create a new process, and then the child process calls exec to load a different program into the memory of the child process. Typically, the shell process then calls wait, and blocks until the child process finishes executing. When the wait call returns, it prints a prompt indicating it is ready to receive the next command.

CSC209 Week 8 – Pipes

Unbuffered I/O

- The only IO operations we have been using so far operate on streams using file objects
  - Functions include:
    - Fopen(), fget(), fclose() etc.
  - Functions are good to use with files as they hide some of the actual complexities of the IO system call – they buffer data
  - Buffer data → the system calls may read or write larger chunks of data than the user specifies; collecting data into larger chunks allows the file system to decide exactly when to send the data to the disk, network or screen
  - Reduces number of system calls made
  - Delegating these tasks allows the user to ignore these complexities
- In the example: how many write system calls are made?
  - In order for the data to be written to the disk, a system call like write() must happen somewhere in the library code
- On Linux we can use a program called strace → run our program and see which system calls are made
  - Will let us determine how many write calls are generated
  - Spits out a lot of output (when using our example) because even a simple program makes a number of system calls

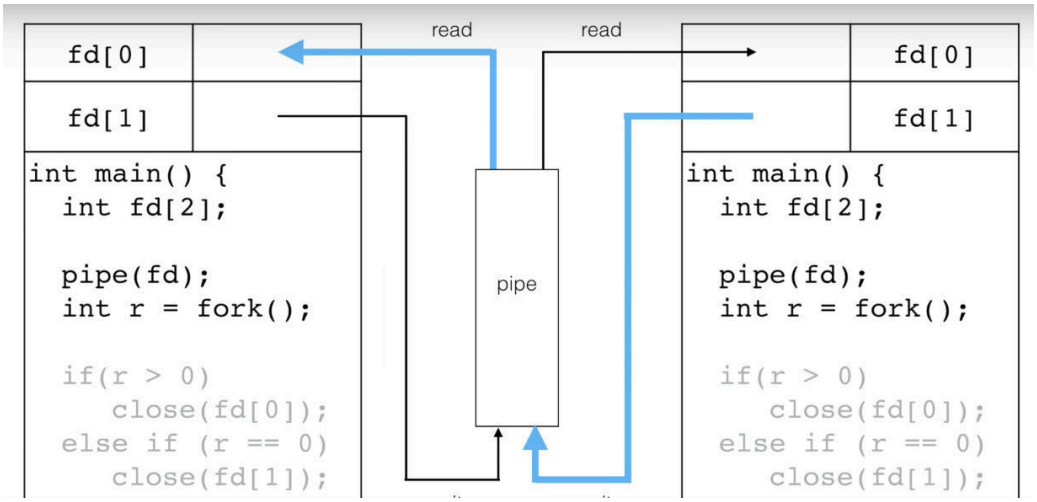
```
#include <stdio.h>
#include <stdlib.h>

int main() {

    FILE *outfp = fopen("tmpfile", "w");

    if(outfp == NULL) {
        perror("fopen");
        exit(1);
    }

    fprintf(outfp, "This is ");
    fprintf(outfp, "one of several ");
    fprintf(outfp, "calls to fprintf.\n");
    fprintf(outfp, "How many write ");
    fprintf(outfp, "system calls are generated?\n");
    fclose(outfp);
    return 0;
}
```

  - (In the example), in all of the output displayed by strace, only a single write called is generated:
    write(3, "This is one of several calls to "…, 84) = 84

        - Writes 84 bytes → the length of the string produced by all 5 of the printf calls
        - What is the value 3? → file descriptor, an integer that represents an open file or open communication channel
            - The file we write to, is assigned 3 rather than 0 or 1 because 3 files are automatically opened for you (stdin, stdout, stderr)
    - File descriptors → type int, because the OS uses them as indexes into a table of open files
- When using gdb to print vars, using a command such as:
  print *stdin, print *stdout or print *stderr
    - Displays the contents of the file struct and can see the file descriptor for the specified key (after the label "_fileno")
    - Knowing the value of a file descriptor is sometimes useful for debugging, but in general but you do not have to look at the value
- Think of the value returned by fopen() or the low-level open() system call as a name that is required when required when performing IO
- Pipes → a form of interprocess communication that also uses file descriptors
    - open() is used to open a file, and pipe() is used to set up a communication channel, they both operator on file descriptors
    - Convenient, because the same read and write system calls can be used for both files and pipes

Introduction to Pipes

- Fork() system call gives us the ability to create multiple processes
    - If a problem can be divided up so that multiple processes can work on it at once, then we can solve the problem faster
    - Can take advantage of machines with multiple processors
    - To cooperate, processors need to be able to communicate
- Pipes → one from of communication mechanism that can be used to send data between related processes
    - Specified by an array of 2 file descriptors
        - 1 for reading data from the pipe
        - 1 for writing data to the pipe
- When a program calls the pipe() system call, the OS creates the pipe data structures and opens 2 file descriptors on the pipe (as specified above) → stored in fd in our example
    - Array of two ints that are passed into the pipe



- The 0th element of the array is always the file descriptor used for reading the pipe, and at index 1, used for writing to the pipe
- After the pipe call returns, the process can now read and write from/to the pipe
  - But isn't useful as a process doesn't need a pipe to communicate with itself
  - Can now take advantage of an aspect of fork()
- When fork() makes a copy of an existing process, it also makes a copy of all open file descriptors
  - Means that the child process inherits all open file descriptors
- Pipes are unidirectional → one process to write to the pipe and another process will read from the pipe
  - Parent can write to the pipe and the child can read from it, or vice versa
  - Once you decided which direction the data should flow, need to close the file descriptors not used
  - Ex. If the parent is to read from the child, close the write descriptor in the parent, and close the read file descriptor in the child
    - Now can use the read and write systems calls to send and receive from the pipe
- Can close stdin from the shell by typing: control d
- When all the write file descriptors are closed, a read on the pipe will return 0 → indicates there is no more data to read from the pipe
  - (In our example) the child process uses this to detect when the parent is finished writing lines to the pipe
  - **while(read(fd[0], other, MAXSIZE) > 0)** → terminates when the parent closes the write end of the file descriptor
    - Will also terminate if an error occurs and read returns -1
- The child process uses a different char array that is the same size as the one used in the parent process → didn't just reuse the line variable is to remind the user that memory is not shared between the processes
- When the child has finished reading all the data from the pipe, we close its read file descriptor for the pipe and exit

- When a process exits, all of its open file descriptors are closed and all memory is free
- Therefore, isn't "necessary" to close all the file descriptors before exiting, but is a good habit
  - Ensures you are thinking of which file descriptors are open
  - In a long running program it is important to close the descriptors that are no longer needed, as the number of open file descriptors is limited – it is possible to run out!

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

#define MAXSIZE 4096

/* A program to illustrate how pipe works.  The parent process reads
 * from stdin in a loop and writes each line it reads to the pipe.  The
 * child process reads from the pipe and prints each line is reads to
 * standard output.  Extra print statements are included so you can see
 * what each process is doing.
 *
 * Notes:
 *  - to close standard input from the shell type ctrl-D
 *  - In this example, we are writing the same number of bytes each time
 *    and reading the same number of bytes each time from the client.
 *    This simplifies the client side.
*/

int main() {
    char line[MAXSIZE];
    int fd[2];
    // create a pipe
    if (pipe(fd) == -1) {
        perror("pipe");
    }

    int r = fork();

    if (r > 0) {
        // Parent reads from stdin, and writes to child
        // close the read file descriptor since parent will write to pipe
        close(fd[0]);

        printf("Enter a line > ");
        while (fgets(line, MAXSIZE, stdin) != NULL) {

            printf("[%d] writing to pipe\n", getpid());
            if (write(fd[1], line, MAXSIZE) == -1) {
                perror("write to pipe");
            }

            printf("[%d] finished writing\n", getpid());
            printf("Enter a line > ");
        }

        close(fd[1]);
        printf("[%d] stdin has been closed, waiting for child\n", getpid());

        int status;
        if (wait(&status) != -1)  {
            if (WIFEXITED(status)) {
                printf("[%d] Child exited with %d\n", getpid(),
                        WEXITSTATUS(status));
            } else {
                printf("[%d] Child exited abnormally\n", getpid());
            }
        }

    } else if (r == 0) {
        close(fd[1]);
        printf("[%d] child\n", getpid());
        // Child will read from parent
        char other[MAXSIZE];

        while (read(fd[0], other, MAXSIZE) > 0) {
            printf("[%d] child received %s", getpid(), other);
        }
        printf("[%d] child finished reading", getpid());
        close(fd[0]);
        exit(0);

    } else {
        perror("fork");
        exit(1);
    }

    return 0;
```

Concurrency and Pipes

- Important details of how pipes really work
- Pipes are used to communicate between two independent processes
  - It is up to the OS when these processes are even run
  - It is possible/likely that the processes won't run within log step of eachother
- Producer Consumer problem
  - One process is producing or writing data
  - The other is consuming or reading data
  - These two processes are connected by a queue
  - The producer and consumer may not work at the same rate
  - If the producer is running and the consumer is not, or the consumer isn't running fast enough to keep up, the data can pile up
    - Problem if there is limited buffer space in the queue
      - We don't want the producer to put data in a full queue
    - If the consumer is running but the producer is not, the consumer will not have enough data to operate on

        - We need to make sure that the consumer doesn't try to remove data frmo an empty queue or a queue where a producer is just starting to add data
    - The cases we're worried about:
        1. Producer adds to the queue when it is full
        2. Consumer removes from the queue when it is empty
        3. Producer and Consumer operate on queue simultaneously
- How the Producer Consumer problem applied to pipes:
    - The pipe is a queue data structure in the OS
    - The process writing to the pipe is the Producer, and the one writing is the consumer
    - Since the OS manages the pipe data structure, it ensures that only one process is modifying it at a time
        - Eliminates the case where the consumer removes data as a producer is writing it
    - Case: if the producer takes longer to write to the pipe than the consumer can read it
        - The consumer process will can read on the pipe when it contains no data
        - The OS helps in this case: the read() call will block, if the pipe is empty
        - Ex. The parent process writes from stdin and writes to the pipe, the child process just reads from the pipe and writes the message to stdout
            - The parent has to wait for the user to type something, thus the child process, spends a bit of its time blocked – waiting for the parent
    - Case: if the producer operates more quickly than the consumer
        - The pipe will eventually fill up
        - To prevent data from being lost, the OS will cause a write to the pipe to cause a block until there is space in the pipe
        - **Can see this in action when we place a sleep statement in the child to slow down its reading
            - Which allows the parent to write multiple times before the child can read
    - To make sure the input is fast, redirect from a file so that the process doesn't have to wait for the user to type each type of input
    - To ensure a pipe is filled, use a large amount of bytes to fill it (duh)

Redirecting Input and Output with Dup2

- Redirecting input and output from one file descriptors to another using Dup2 system call
- Why you might want to do this

- Use grep in the shell to search for lines in a file that contain a specified phrase: i.e. **grep L0101 list.txt**
  - Prints the output to stdout/the screen
  - Printing output to the screen only allows us to read it
- The power of the shell includes using redirection operators to send the output to different places without modifying the program
  - Which allows use to save the output to a file
    i.e. grep L0101 student_list.txt > day.txt
    - Indicates we want the output to go to a file, rather than the screen
  - Allows use to combine programs to complete complex tasks
    i.e. grep L0101 student_list.txt | wc
    - Here the pipe operator is used to pipe the output from grep as the input for the wc program
- Need a way to change what stdout means, so when grep writes to stdout, it actually writes to a file
  - Int dup2(int oldfd, int newfd)
  - System call
  - Makes a copy of an open file descriptor
    - Can be used to reset the stdout file descriptors so that writes to stdout will go to an output file
- A file descriptor is really an index to a table
  - Each process has its own set of file descriptors
    - Means each has its own file descriptor table
    - Table is stored in the process control block
    - Contains pointers to data structures that contain information about opened files
  - Index 0 usually contains a link to the console
- To create a new process: fork() (as we learned)
  - When a child process is created, it obtains a copy of the file descriptor table from its parent
  - Even though the file descriptor tables are separate, the pointers in them may point to the same object
    - File objects like the console (I the example) are shared
    - Changes to the console will be observed by both processes
  - When the child runs, it opens the file that will receive the output of the program
    - Note: we have opened the file with write permissions so that it can receive output
  - Can redirect output using dup2()
    dup2(filefd, fileno(stdout));
    - Extract the file descriptor for stdout (a global var of type file pointer) using the file no function

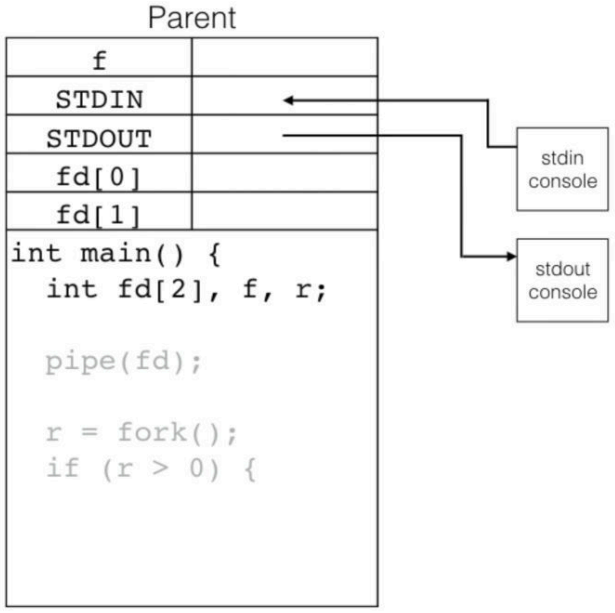
        - After the call, file descriptor 2 points to the newly opened file rather than the console
    - When the process writes to stdout, the output will be written to the file rather than the console
        - It is a good idea to close the file descriptor that are not in use

```
/* Demonstrate how file redirection is implemented using dup2.  */

int main() {
    int result;

    result = fork();

        /* The child process will call exec  */
        if (result == 0) {
        //int filefd = open("day.txt", O_RDWR | S_IRWXU | O_TRUNC);
        int filefd = open("day.txt", O_RDWR | O_CREAT | O_TRUNC, S_IRWXU);
                if (filefd == -1) {
                        perror("open");
                }
        if (dup2(filefd, fileno(stdout)) == -1) {
                        perror("dup2");
                }
                close(filefd);
                execlp("grep", "grep", "L0101", "student_list.txt", NULL);
                perror("exec");
                exit(1);

        } else if (result > 0) {
            int status;
                printf("HERE\n");
                if (wait(&status) != -1) {
                    if (WIFEXITED(status)) {
                            fprintf(stderr, "Process exited with %d\n",
                                    WEXITSTATUS(status));
                        } else {
                            fprintf(stderr, "Process teminated\n");
                        }
                }

        } else {
            perror("fork");
                exit(1);
        }
    return 0;
}
```

Implementing the Shell Pipe Operator

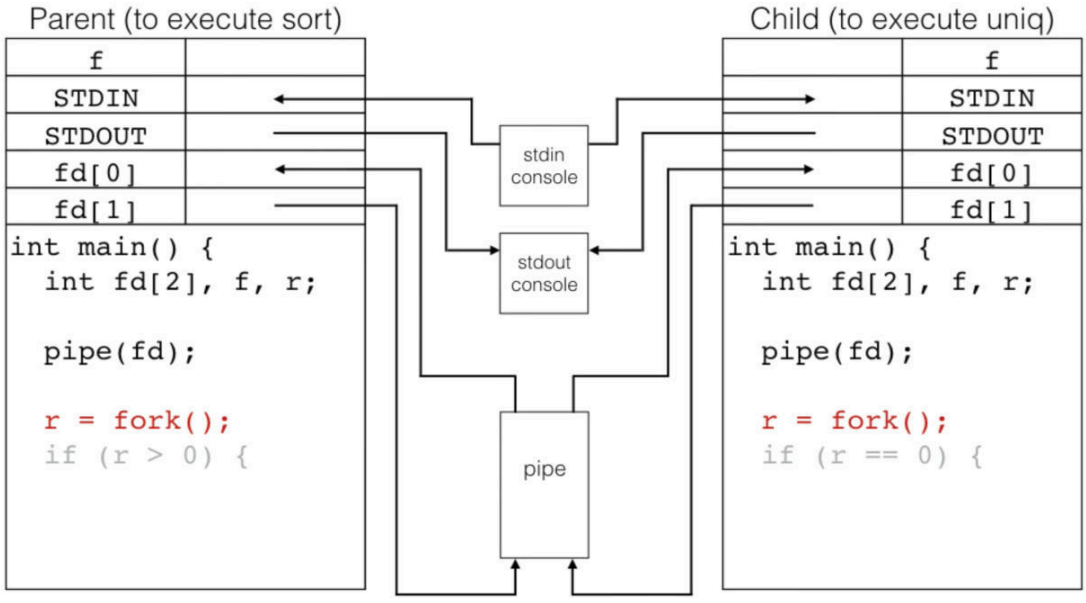
- The shell pipe operator allows us to connect 2 processes so that the stdout from one process becomes the stdin for the next processes
- sort → sorts its input and writes to stdout
    - Can operator on stdin or as files passed in as command-line arguments
- uniq → only omits lines that are different from the line that proceeds them

- When the program starts running, only the standard descriptors to the console are available




```
STDIN = STDIN_FILENO;
          OR
STDIN = fileno(stdin);

STDOUT = STDOUT_FILENO;
          OR
STDOUT = fileno(stdout);
```

- The first step: call the pipe system call to create a pipe that will connect sort and uniq
- Next: call fork
  - Notice: the child process inherits all the same open file descriptors
- Now we must set the file descriptors correctly and close the ones that are not needed



- Parent process: will be executing sort → we want stdin to come from the file (in this case "file1.txt")
  - Open the file for reading
  - Then use dup2 to reset the stdin so that the data will come from the file

        - Now stdin is set correctly
    - Close f as that file descriptor will not be used directly
    - Close fd[0], as the example file will not be reading from the pipe
    - Next step: connect stdout to the pipe so we can send the output from sort to the input of uniq
        - Close fd[1], as we wont be writing to the pipe using this file descriptor
            - **Very important: if we don't close all of the write descriptors, the read end of the pipe won't know when there's nothing left to read from the pipe
        - The communication channels for the parents are now set correctly
- Child process: connect stdin to the read end of the pipe so when the process reads from stdin, the data comes from the pipe
    - Close fd[0], as we won't be using that file descriptor to read from the pipe
    - Close fd[1], as this process will not be writing to the pipe
    - All the file descriptors now properly set in both parent and child
    - Call exec to load the executables for sort and uniq into the appropriate processes
        - The exec call preserves all open file descriptors

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/wait.h>  /* equivalent to sort < file1 | uniq */

int main() {
        int fd[2], r;

        /* Create the pipe */
        if ((pipe(fd)) == -1) {
          perror("pipe");
          exit(1);
        }
        if ((r = fork()) > 0) { // parent will run sort
                // Set up the redirection from file1 to stdin
                int filedes = open("file1", O_RDONLY);

                // Reset stdin so that when we read from stdin it comes from the file
                if ((dup2(filedes, fileno(stdin))) == -1) {
```

```
            perror("dup2");
            exit(1);
        }
         // Reset stdout so that when we write to stdout it goes to the pipe
        if ((dup2(fd[1], fileno(stdout))) == -1) {
            perror("dup2");
            exit(1);
        }

        // Parent won't be reading from pipe
        if ((close(fd[0])) == -1) {
            perror("close");
        }
        // Because writes go to stdout, noone should write to fd[1]
        if ((close(fd[1])) == -1) {
            perror("close");
        }

        // We won't be using filedes directly, so close it.
        if ((close(filedes)) == -1) {
            perror("close");
        }
        execl("/usr/bin/sort", "sort", (char *) 0);
        fprintf(stderr, "ERROR: exec should not return \n");
} else if (r == 0) { // child will run uniq

        // Reset stdi so that it reads from the pipe
        if ((dup2(fd[0], fileno(stdin))) == -1) {
            perror("dup2");
            exit(1);
        }

        // This process will never write to the pipe.
        if ((close(fd[1])) == -1) {
            perror("close");
        }
        // SInce we rest stdin to read from the pipe, we don't need fd[0]
        if ((close(fd[0])) == -1) {
            perror("close");
        }
        execl("/usr/bin/uniq", "uniq", (char *) 0);
        fprintf(stderr, "ERROR: exec should not return \n");
} else {
        perror("fork");
```

```
                exit(1);
        }
        return 0;
}
```

CSC209 – Week 9: Signals

Introduction to Signals

- In this example dots.c just prints out dots to stderr, so we can see that the program is running
- Termination signal to process → hold down the ctrl key and type the letter C; ctrl-C
- Suspend signal to process → hold down ctrl key and type the letter Z; ctrl-Z
    - fg (at command line) → starts the program again
- Signals → a mechanism that allow a process or the OS to interrupt a currently running process and notify it that an event has occur
- Each signal is identified by a number between 1 and 31
    - Defined constants are used to give them names
- Each signal is associated with a default action that the process receiving the signal will normal perform when it receives that signal
    - Ctrl-C from the terminal sends the SIGINT signal to the process, where the default action is the process to terminate
    - Ctrl-Z from the terminal sends the SIGSTOP signal to the process, and the default action is the process to suspend execution
- How to send arbitrary signals to a process?
    - Can do it using a library function called “kill” or from the command-line also using a program called “kill”
        - Before we send signals to a process, we need to know its process id (PID)
            - Can find this by running ps
                - i.e. ps aux |grep dots
                - The number in the second column is the process id
        - Can then use kill to use the signal STOP
            - i.e. kill –STOP 3819
                - In this case sent the SIGSTOP signal to the dots program (AKA signal 17)
                - Can also send signal 17 directly
            - This then suspends the program
        - To get it running again use the signal CONT
            - i.e. kill –CONT 3819
        - To terminate the program send a SIGINT signal
            - i.e. kill –INT 3819
    - The library function kill, provides the same functionality as the kill program except that it is a C function, therefore inside your C code you can send a signal to another process
        - You would need to know the PID of the process you are signalling
        - In most cases we use signals to send messages to our children so we can get the PID from the return value of fork

- A child process could get the PID of its parent with get PPID

Handling Signals

- We talked about how signals can be used to notify a process of an event
  - Each signal has a default action associated with it
- There are times when we would like to be able to change the default behaviour of the signal
  - Ex. to print a message, save some state, or possibly ignore the signal
  - We can write a function to be executed when the signal is received by the process
    - But how does this function get called?
    - Have to talk about what happens when a signal is delivered to a process
- The process control block contains a **signal table**, which is similar to the open file table
  - Each entry in the signal table contains a pointer to code that will be executed when the operating system delivers the signal to the process
    - This is called the signal handling function
- We can change the behaviour of a signal by installing a new signal handling function
  - **sigaction()** → system call will modify the signal table so that our function is called instead of the default action
    - The first argument is the number of the signal that we are going to modify
    - The second argument is a pointer to struct initialized before sigaction()
    - The third argument is also a pointer to a struct, but in this case the system call fills in the values of the struct with the current state of the signal handler before we change it
      - Sometimes useful to save the previous state of the signal
  - The first field of the sigaction struct is the function pointer for the signal handler function that we want to install
  - Signal handler → a function that will execute when the process is signalled
    - We can select the name
    - Needs to take a single integer parameter and have a void return type
    - Ex.

```
void handler(int code) {
      fprintf(stderr, "Signal %d caught\n", code);
}
```

    - Now we need to add code to install our new function in the signal table

```
int main() {
        // Declare a struct to be used by the sigaction function:
        struct sigaction newact;
        // Specify that we want the handler function to handle the
        // signal:
        newact.sa_handler = handler;

        // Use default flags:
        newact.sa_flags = 0;

        // Specify that we don't want any signals to be blocked during
        // execution of handler:
        sigemptyset(&newact.sa_mask);

        // Modify the signal table so that handler is called when
        // signal SIGINT is received:
        sigaction(SIGINT, &newact, NULL);

        // Keep the program executing long enough for users to send
        // a signal:
        int i = 0;

        for (;;) {
                if ((i++ % 50000000) == 0) {
                        fprintf(stderr, ".");
                }
        }
        return 0;
 }
```

- When we send the INT signal to the process, we can see the new message printed to standard error
- But the process doesn't terminate!
  - We didn't call exit in handler, so when the signal handling function finishes, control returns back to the process at the point where it was interrupted

- There are two signals that you can't change: SIGKILL and SIGSTOP
  - **SIGKILL** will always cause the process to terminate
  - **SIGSTOP** will always suspend the process

CSC209 – Week 10

Bit Manipulation and Flags

- Introducing Bitwise Operations
  - C is really a low-level programming language
    - We can read and modify each bit in a value directly
    - Allows us to work directly with raw data, ignoring the type of the variable
  - Recall: C represents
    - False as the number 0
    - True as the number 1
  - C provides a logical **and** operator - two ampersands &&
    - Looks at the whole number and returns true if both arguments have a value other than 0
  - C also provides a bitwise **and** operator that is a single ampersand &
    - Performs the AND operation on each pair of bits
    - For a single bit, the two operators look identical

```
1 && 1 == 1       1 & 1 == 1
1 && 0 == 0       1 & 0 == 0

0 && 1 == 0       0 & 1 == 0
0 && 0 == 0       0 & 0 == 0
```

    - But if you look at values other than 0 and 1, you'll see that they operate differently

```
8 && 8 == 1       8 & 8 == 1
8 && 1 == 1       8 & 1 == 0
8 && 0 == 0       8 & 0 == 0
```

```
1 is 0001 in binary
8 is 1000 in binary
```

  - Similarly, although you're already familiar with logical **or**, which is two vertical bar ||, C also provides a bitwise **or** - one bar |
    - Like bitwise and, bitwise **or** works by performing the OR operation on each pair of bits

```
1 | 1 == 1
1 | 0 == 1
0 | 1 == 1
0 | 0 == 0
```

- The bitwise **exclusive or** operator is represented by a carat or hat symbol ^
  - It is 1 only if exactly one operand is 1
  - The result is 0 if both operands are 0 or both operands are 1
  - Exclusive or is sometimes called conditional negation:
    - If the first operand is a 1 (true) then the result is the complement, the negation, of the second operand
    - If the first operand is 0 (false) the result is the second operand WITHOUT being negated

```
1 ^ 1 == 0
1 ^ 0 == 1
0 ^ 1 == 1
0 ^ 0 == 0
```

- C also provides a bitwise negation, or complement, operator, the tilde ~
  - It takes a single value and flips every bit in the value
  - Since it's a bitwise operator, while the bit 0 becomes the bit 1, the integer 0 will NOT become the integer 1*****

```
~0 == 1
~1 == 0
```

- The first problem is how to store bits in a variable
  - While the C language doesn't define non-decimal constants, gcc allows you to enter binary constants by prefacing the number with 0b
    - Ex. 0b00010011
  - You can also use hexadecimal constants, prefaced with 0x
    - Ex. 0x14
    - Hex is convenient because the binary numbers are usually large, and it is easy to translate from hexadecimal to binary

- The Shift Operator
  - Given a variable b, set the third bit to 1 and leave the other bits unchanged
    char b = 0xC1; //1100 0001
    - We count bits from right to left starting at position 0, so the bit at position 3 in our variable b has the value 0
    - We can use the bitwise or operator here
      - Let's use the value 8 //1100 1001
      - Using bitwise or, any value we combine with 8 will end up with a 1 at the 3rd bit
    - **b | 0x8**
  - Given a variable b, check if the second bit is has the value 1

        - We will use the & operator, because the only way that bitwise and results in 1 is if both bits are 1
        - We use the value 4, since it has a 1 at the position we want to check
          //0000 0100
    - Suppose we want to make these problems more general
        - To create a value with an arbitrary bit set to one
        - For that will use C's shift operator
            - There are actually two shift operators - shift left << and shift right >>
            - Think of them as arrows pointing in the direction of the shift
            - Takes two operands
                - The first - to the left of the operator - is the value to shift
                - The second - to the right - tells us how many places to shift
            - Ex. 1 << 3 → 0000 0001
                - Transforms into 0000 1000
        - So, by replacing the second operand with a variable, we can generate a value with any single bit set
            - To set the kth bit of var, we will bitwise-or var with 1 shifted k times
            - Similarly we can check if the kth bit is set by bitwise **and**-ing var with 1 shifted k times
        - Ex.
          char a = 0xAE //1010 1110
          a = a << 2; // becomes 1011 1000

- Bit Flags
    - Flag bits are commonly used by system calls, when a single argument is used to transmit data about multiple options
        - That variable is treated as an array of bits, where each bit represents an option -- or flag -- that can be turned on and off
    - How file permissions work in Linux
    - After running ls –l, you get the contents of a directory
        - The first column in the output is the permission string
            - Represents who can read, write, or execute the file
        - The third column is the owner of the file
        - The fourth column is the group
    - The owner and group fields are relevant to permissions because Linux allows us to set separate permission for the owner of the file, the group, and every other user in the system
    - Let's look more closely at the permissions for syscall_cost
        - The leftmost dash in the permission string identifies the file type

- The dash means that it is a regular file
- "results" which is a directory, has a d in this position, and a link would have an l

- How does this relate to bit manipulation operators and flag bits?
  - If you ignore the first dash that represents the file type, each file permission is essentially an on/off switch, so we can represent each one with a flag bit
- We need nine bits to represent each permissions setting, so we need a variable with enough space for 9 bits
  - This rules out storing it in a byte
  - The system calls that operate on file permissions use a mode_t type for the permissions string
    - On my system, this is defined as an unsigned int - a 32-bit value
    - The permissions will be stored in the 9 low-order bits -- or rightmost – bits
    - The first step is to see how to turn the permissions string into a sequence of bits
      - Each character in the permissions string becomes a bit in our variable representing the permissions
      - Bits 8, 7, 6 represent the read, write, and execute permissions for the user
      - Bits 5, 4, and 3 represent the permissions for the group
      - Bits 2, 1, and 0 represent the permissions for everyone else.
    - What if we wanted to represent a file with read-only permissions? The permissions string for a read-only file would be r--r--r-- So we'd represent it with the binary value 100100100
- One of the system calls that operates on file permissions is the chmod system call
  - The arguments to chmod are the path to file we want to set permissions for, and the mode or permissions to set
  - There is a set of defined constants which will make it easier to set a value for the mode
    - **Note that these constants are defined in octal or base 8
- Back to our problem: we can create the permission value using the bitwise operators we learned about in previous videos
  - In this case, to set bits, we use bitwise OR with the appropriate constants
  - Similarly, we can check if bits are set using bitwise AND
- Flag bits are a compact, efficient method for representing data with multiple boolean components

    - The use of individual bits, rather than whole characters or integers, saves space, while the use of bitwise operators is clean and fast.

- Bit Vectors
    - We'll explore the use of bit flags to implement a *set*
    - If we use each bit in a variable to denote the presence -- or absence -- of a particular element in the set, then we have a very compact implementation that can use fast boolean instructions as set operations
    - In the case where the possible set elements are represented by small positive integers…
        - For example, you might want to store colours in a set
        - By giving each colour a numeric code, so that red is 2, and purple is 7 for example, we can then store the codes in the set 0000000010101100
        - When our set contains small positive integers, we can represent the set as an array of bits where the elements of the set are the indexes into the array and the value at a given index location tells us whether the element is in the set
        - To add another element, say 10, to bit_array we first use the shift operator to create a value where the 10th bit is 1 and all the other bits are 0 → **1 << 10**
        - Bitwise ORing it with bit_array adds 10 to the set
        - What you've just seen is a common programming technique called **bit masking**
            - A carefully constructed value where specific elements are set or not set - and then applying the mask to set or unset those values
        - To remove 10 from the set, we want to bitwise AND a zero in the 10th bit with bit_array, but if the other bits are zero, that would have the side effect of removing all items from the set
            - Create a value where all of the bits are one except the 10th bit, and bitwise AND that value with bit_array
        - In this example, we used an unsigned short to hold the bit array
            - Means that the range of values the set can contain is 0 to 15
            - We can double the range of the set by changing the type of our variable to an unsigned int, but even 31 is pretty small to be the maximum value
            - We can generalize the data structure that we use to store the bits by making an array of unsigned ints
            - Of course, the operations to add a value to a set and remove a value from the set are a little more complicated now, so let's look at an example of how to identify a particular bit in the bit array (See PCRS)

Multiplexing I/O

- The Problem with Blocking Reads
  - Pretty much this video talks about how blocking reads occurs when there is more than 1 child to read from, and the earlier calls to read from a specific child do not execute as there is nothing written by that specific child, even if the others do
  - This prevents proceeding code from executing
- Introducing Select
  - Can avoid read blocks by using a system call named select
    - The caller specifies a set of file descriptors to watch
    - The second parameter is the address of the set of descriptors from which the user wants to read
      - Select blocks until one of these file descriptors has data to be read or until the resource has been closed
      - In either case, the user is certain now that calling read on that file descriptor will not cause read to block
      - We will say that a file descriptor like this (with data or with a closed resource) is "ready"
    - Select modifies the descriptor set so that when it returns, the set only contains the file descriptors that are ready for reading
  - We insert our select call before calling read on either child
    - We check the return code and call perror if we detect a problem (== -1)
  - Recall: read_fds is a *set* of descriptors to be watched
    - Need to declare and initialize this set before calling select
    - We'll declare a variable of type "fd_set" -- for "file descriptor set"
      - Is implemented as a bit field stored in an array of integers
    - Macros have been provided to perform standard set operations -- like inserting an item into the set, removing an item from the set, or checking if an item is a member of the set. This macro initializes the set of descriptors to be empty. And these two lines add the file descriptors for the read ends of the two pipes into the formerly-empty set
      - Now we have a set that contains exactly these two file descriptors
  - But what about the first parameter of select?
    - Set this parameter to be the value of the highest file descriptor in your set ** + 1
    - Select can be implemented more efficiently if we specify that our set can only contains file descriptors between 0 and numfd – 1
    - So in our case we add one to the larger of the two file descriptors
  - But how does the call send a response back to us?

- In our code we've used the return value only for error checking which is set to the number of file descriptors that are ready, but doesn't tell us which ones
- Notice that we sent the *address* of the file-descriptor set to select → select modifies read_fds
- When select does finally return, the descriptor set read_fds will ONLY contain the file descriptors that are ready
    - So we need to use another set operation - checking set membership - to determine which descriptors remain in the set. The macro FD_ISSET takes a single fd and returns whether or not it is in the fdset pointed to by the second parameter. So in our case, we want to know if the pipe from child 1 is in read_fds after the select call. If it *is* in the set, then we know that this pipe is ready, so we can read from it without fear of blocking. We add a similar check to see if the file descriptor from child 2's pipe is ready and then read from it. Note that this is an independent if statement. It is possible that after select returns, BOTH pipes are ready for reading. That's enough to demonstrate the basic use of select, but we should say something about those other parameters that we set to NULL. Like the second parameter, the third and fourth are also file descriptor sets. You can use these sets to check which file descriptors are ready for writing or have error conditions pending, respectively

- The final parameter is a pointer to struct timeval. You can use it to set a time limit on how long select will block before returning (even if no file descriptors are ready.) You can use this, for example, to interrupt the blocked select call after some number of seconds to do some other task. Then you could call select again and resume watching the file descriptors for activity
- One other issue: read_fds is modified by select, so we can't just use it again in a second select call. If we want to read from our pipes multiple times, in a loop, we'll need to re-initialize the set