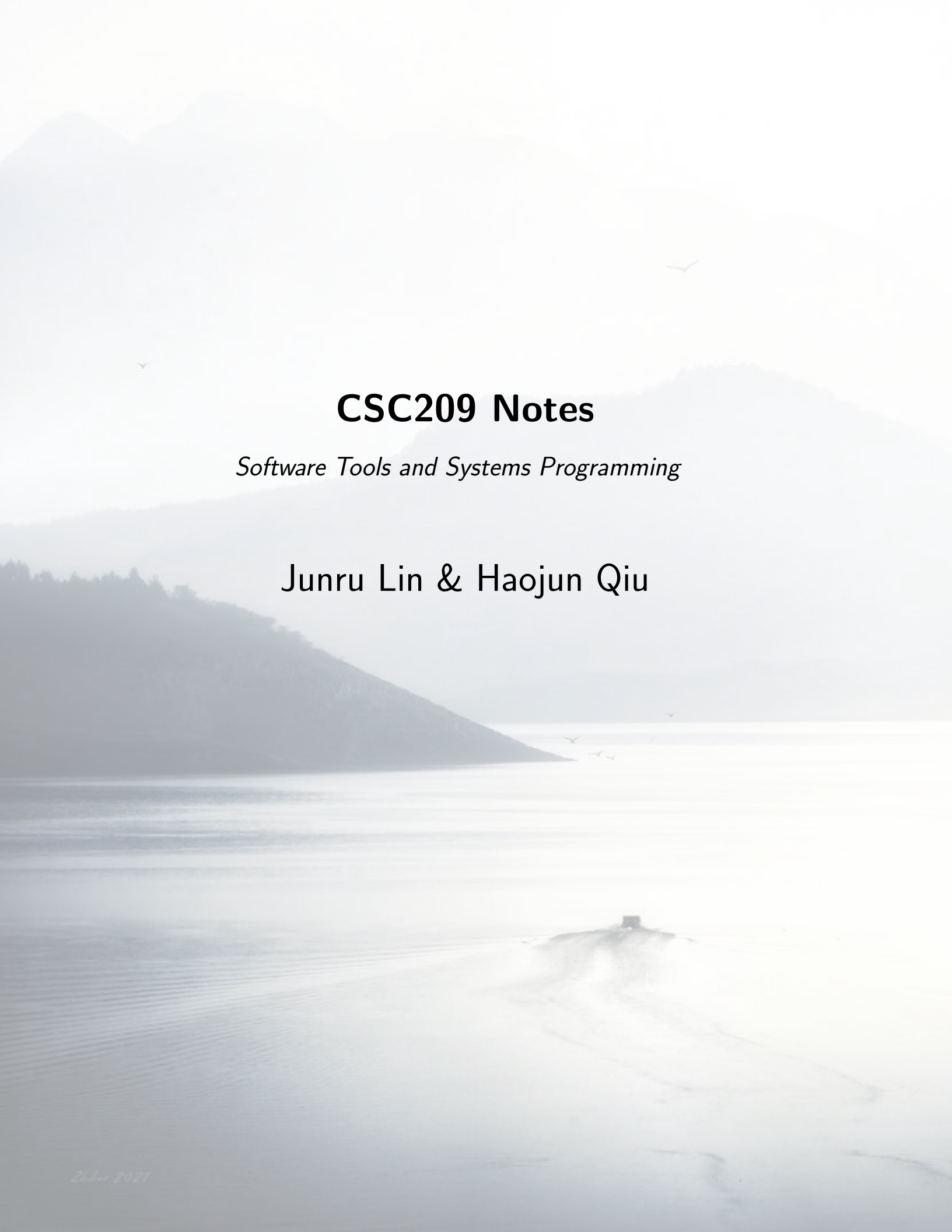


CSC209 Notes

Software Tools and Systems Programming

Junru Lin & Haojun Qiu

Contents

1 Arrays **6**

1.1 Introduction 6

1.2 Accessing Array Elements 6

1.3 Iterating Over Arrays 6

2 C Memory Model **8**

2.1 Code and Stack Segments 8

2.2 Heap and Global Segments 9

3 Command-line Arguments **10**

3.1 Converting Strings to Integers 10

3.2 Command-line Arguments 11

3.3 A Worked Example 11

4 Strings **12**

4.1 Introducing Strings 12

4.2 Initializing Strings (Variable) and String Literals 12

4.3 Size and Length 13

4.4 Copying Strings 13

4.5 String Concatenation 14

4.6 Searching with Strings 14

5 Structs **15**

5.1 Using Structs in Functions 15

5.2 Using Structs in Functions 15

5.3 Pointers to Structs 16

6 Linked Structure **17**

6.1 Introduction 17

6.2 Traversing a List 17

6.3 Insertion into the Middle of a List 18

6.4 Testing Insertion 19

7 Streams **20**

7.1 Introduction 20

7.2 Redirection 20

8 File **21**

8.1 An Introduction to Using Files 21

8.2 Reading From Files 22

8.3 The scanf Function 22

8.4 Writing to Files 23

8.5 Putting It All Together 23

9 Low-Level I/O **25**

9.1 Introducing Binary Files 25

9.2 Writing Binary Files 26

9.3 Reading Binary Files 27

9.4 Putting it Together: wav Files 27

9.5 Increasing the Sound in wav Files 28

9.6 Reading and Writing Structs 29

9.7 Moving Around In Files 29

10 Compiling **30**

10.1 The Compiler Toolchain 30

10.2 Header Files 31

10.3 Header File Variables 32

10.4 Makefiles 33

11 C Features **35**

11.1 Typedef 35

11.2 Macros 35

12 The C Preprocessor **37**

12.1 Simple Macros and Header Files 37

12.2 Function-like Macros 39

13 Function Pointers **40**

13.1 Introducing Function Pointers 40

13.2 Extending the Worked Example 41

14 System Calls **42**

14.1 What is a System Call? 42

15 Errors and Errno **43**

15.1 When System Calls Fail 43

15.2 Checking For Errors 43

16 Process Model **45**

16.1 Process Models 45

16.2 Creating Processes with Fork 46

16.3 Process Relation and Termination 46

16.4 Zombie and Orphans 47

16.5 Running Different Programs 48

17 Signals **49**

17.1 Introduction to Signals 49

17.2 Handling Signals 49

18 Bit Manipulation and Flags **52**

18.1 Introducing Bitwise Operations 52

18.2 The Shift Operator 53

18.3 Bit Flags 53

18.4 Bit Vectors 54

19 Multiplexing I/O **57**

19.1 The Problem with Blocking Reads 57

19.2 Introducing Select 57

20 Sockets **60**

20.1 Intro to Sockets 60

20.2 Socket Configuration 61

20.3 Setting Up a Connection 63

20.4 Socket Communication 64

21 Shell Programming **66**

21.1 Part 1 (Command Line Substitution & Variables) 66

21.2 Part 2 (Control Constructs - If-Statement & While-Loop) 67

21.3 Part 3 (Control Constructs - For-Loop & Case-Statement) 71

21.4 Part 4 (File Redirection) 74

Chapter 1

Arrays

1.1 Introduction

```
// Declare an array of 4 float values
float daytime_high[4];

// Initialize these values one at a time
daytime_high[0] = 16.0;
daytime_high[1] = 12.8;
daytime_high[2] = 14.6;
daytime_high[3] = 19.1;
```

- Type of `daytime_high`: `float []`

1.2 Accessing Array Elements

```
int A[3] = {13, 55, 20};

// It might cause an error and stop the execution (crash)
// or it might appear to work and assign a random value to out_of_bounds.
int out_of_bounds = A[3];

// We can assign to something out of bounds as well.
// It is wrong but may or may not cause a visible error.
A[5] = 999;
```

- May cause `segmentation fault`
- address of `A[i]` = address of `A` + `i` * size of one element of `A`

1.3 Iterating Over Arrays

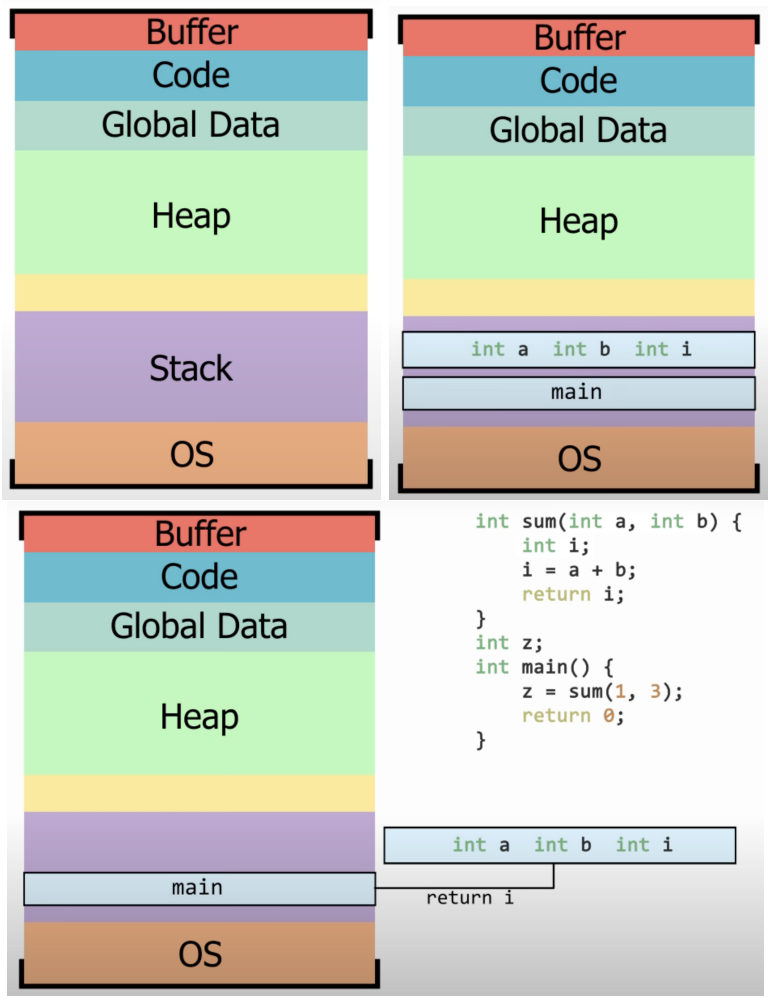
```
    float daytime_high[DAYS] = {16.0, 12.8, 14.6, 19.1};
    float average_temp = 0;

    // loop here over the array and add each element to the running total
    int i;
    for (i = 0; i < DAYS; i++) {
        printf("adding element %d with value %f\n", i, daytime_high[i]);
        average_temp += daytime_high[i];
    }
```

Chapter 2

C Memory Model

2.1 Code and Stack Segments

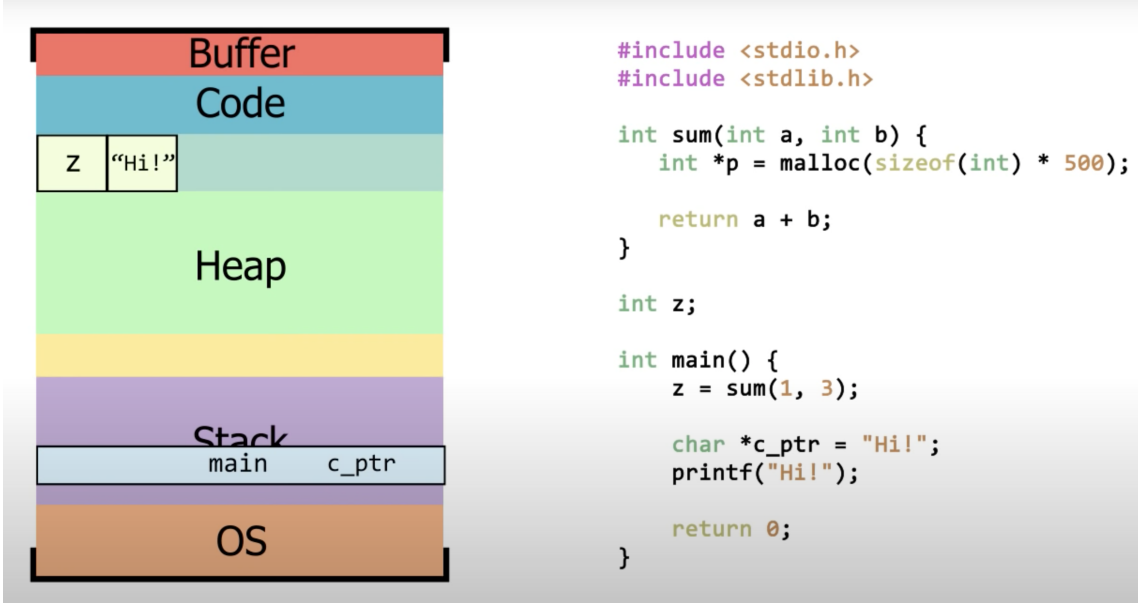


- The most recent function call is at the top of the stack

- Function calls are removed in last-in-first-out order.
- Local variables are only accessible while the function that defines them is active. Once the stack space has been deallocated, the variable is no longer a valid memory location.

2.2 Heap and Global Segments

Global Data



- Global variables
- String literals

Errors

- If the program exceeds the maximum size of the stack or heap, it will encounter an out of memory error, or `ENOMEM`.
- The OS segments cannot be accessed by normal programs. If you do somehow try to address OS memory, you will get something called a `Segmentation Fault` – or `segfault`.
- `Segmentation Fault`: When we try to access something we don't have access to or something that does not exist. You can get a segfault in a few other ways.

Chapter 3

Command-line Arguments

3.1 Converting Strings to Integers

Function

```
long int strtol(const char *str, char **endptr, int base);
```

- `strtol`: string to long
- `str`: the string that we want to convert
- `endptr`: where the "leftover" piece of string starts (see Example 2).
- `base`: the base of the number system that we want to use to interpret the string we are converting (e.g., `10` and `2`)

Example 1

```
    //char *s = "17";
    char *s = "  -17";
    int i = strtol(s, NULL, 10);

    printf("i has the value %d\n", i);
```

Example 2

```
    s = "  -17 other junk.";
    char *leftover;
    i = strtol(s, &leftover, 10);

    printf("i has the value %d\n", i);
    printf("leftover has the value %s\n", leftover);
    return 0;
```

Result of Example 2

```
i has the value -17
leftover has the value  other junk.
```

3.2 Command-line Arguments

```
int main(int argc, char **argv)
```

```
int main(int argc, char *argv[]);
```

- argc: the number of commandline arguments
- argv: argument vector, stores an array of strings.
- argv[0]: the name of the executable
- rest of argv: the commandline arguments

3.3 A Worked Example

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

int main(int argc, char **argv) {

    // Ensure that the program was called correctly.
    if (argc < 3) {
        printf("Usage: sum_or_avg <operation s/a> <args ...>");
        return 1;
    }

    int total = 0;

    int i;
    for (i = 2; i < argc; i++) {
        total += strtol(argv[i], NULL, 10);
    }

    if (argv[1][0] == 'a') {
        // We need to cast total to float before the division.
        // Try removing the cast and see what happens.
        double average = (float) total / (argc - 2);
        printf("average: %f \n", average);
    } else {
        printf("sum: %d\n", total);
    }

    return 0;
}
```

Chapter 4

Strings

4.1 Introducing Strings

- String is not a new data type, is still a char array, but with the last element being `'/0'`.
- A string (a char array ends with `'/0'`) can be displayed without garbage followed by it.

4.2 Initializing Strings (Variable) and String Literals

- Both of below initialize text to be a string value `"hello"`, the remaining 15 slots of the array are all assigned with `'/0'`.

```
char text[20] = "hello";
char text[20] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

- We can change a char in the string anytime, by indexing into the array. E.g.,

```
text[1] = 'j';
```

- We can also initialize it by:

```
char text[] = "hello";
```

  so now text is an array of size (number of characters in the string literal = 5) + 1 for `'/0'`, a total of fixed 6 slots is assigned automatically.

- A **string literal** is a string constant that cannot be change (differs from string variable - an array of chars, talked above). It is initialized as

```
char *text = "hello";
```

  so that text points to the first character of string constant `"hello"`. Since `"hello"` is constant, change it using, e.g., `text[0] = 'j';` results an (bus) error.

4.3 Size and Length

- `sizeof()` is **not** a way to determine # of chars in a string (or in general, not for determining of elements in an array). It always gives the size of memory allocated for the array variable.
- Use string functions by `#include <string.h>` in our program
- `strlen()` can be used to determine string lengths
  - Prototype: `size_t strlen(const car *s)`
  - It returns number of characters in the string s **NOT** including the `NULL` termination character (`size_t` is just an unsigned integer type, we just treat it as a integer)
  - It counts number of characters in string before the first `'/0'` occurs (once encounters, stops counting ignore the rest of chars)

4.4 Copying Strings

- Copying: overwrite what was previously there
- Function prototype: `char *strcpy(char *s1, const char *s2)`
  - If `s1` is just a pointer to a char, it should be initialized (e.g., by `malloc`) before being called by `strcpy`.
  - Copies the characters from string `s2` into the beginning of array `s1` (`s1` isn't required to be a string when `strcpy` is called, that is, it can just be an array of chars with no `'\0'` character, but `s2` is required to be a string - either a string literal or a char array that includes a null terminator `'\0'`)
  - It's unsafe, below is problematic (it copies into s1 anyway, but different machine have different outcomes that can cause problem)

```
char s1[5];
char s2[32] = "University of";
strcpy(s1, s2);
```

- `char *strncpy(char *s1, const char *s2, int n);`
  - The new `n` parameter indicates the maximum number of characters that `s1` can hold (in best, always set to capacity of `s1`), and therefore the maximum number of characters including any null character that can be copied from `s2` into `s1`
  - It copies up to `n` characters, stopping with the first null terminator in `s2` or once it reaches `n` characters (so, can copy less than `n` characters).
  - However,
    - `strncpy(s1, s2, sizeof(s1));` is also unsafe since it is not guaranteed to add a null terminator, unless it finds one in the first `sizeof(s1)` characters of `s2`.
    - Add manually by `s1[4] = '0';` to make it safe
  - INVALID usage (assignment to expression with array type)

```
char delicious[30];
delicious = "ice cream";
```

– Correct usage

```
char delicious[30];
strcpy(delicious, "ice cream");
```

4.5 String Concatenation

- `char *strcat(char *s1, const char *s2);`
  - Adds string `s2` to the end of string `s1`, both `s1` and `s2` must be string prior to calling `strcat`.
  - It is unsafe since `s1` may not have enough space to store both its own and `s2`'s contents
- `char *strncat(char *s1, const char *s2, int n)` (safe version)
  - `n` parameter indicates the maximum number of characters not including the null terminator, that should be copied from `s2` to the end of `s1`
  - Unlike `strncpy`, `strncat` always adds a null terminator to `s1`
  - Correct way: `strncat(s1, s3, sizeof(s1) - strlen(s1) - 1);` (space left - 1 to make room for the null terminator)

4.6 Searching with Strings

- Searching for single character: `char * strchr(const char *s, int c);`
  - `s` is the string to search, `c` is the character (in ASCII code) to search for in the string, from left to right (when calling it, we can input a character however, e.g., `strchr(s, 'a')`)
  - Returns a pointer to the first occurrence of character found, or `NULL` if the character is not found in the string
  - Although index is not directly return, we can use pointer arithmetic to determine that
- Searching for substring in string: `char *strstr(const char *s1, const char *s2);`
  - Searches left to right in `s1` for the first occurrence of the substring `s2`
  - If `s2` is found in `s1`, `strstr` returns a pointer to the character of `s1` that begins the match with `s2`

Chapter 5

Structs

5.1 Using Structs in Functions

- Array helps organize values of the same type.
- Structs is a collection that can contain different data types.
- Declare a structure type with `struct` keyword, and a struct tag `student`:

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

- Initialize a struct with `struct` keyword follow by struct tag and the variable name for struct, e.g.,

```
struct student good_student;
```

- We should explicitly initialize the member of struct, accuse them using dot, e.g.,

```
strcpy(good_student.first_name, "Jo");
```

5.2 Using Structs in Functions

- Struct are not passed in a function the same way as an array
- When passing an array by the array name, we passed in the pointer to its first element (rather than a copy of the array)
- But for structs a function does get a copy of the struct if passing by name of struct variable (hence any change that the function make is a change to the copy)

- Note that a function gets a copy of the entire struct, including its array members if exist (i.e., it is a deep copy)
- Two methods to change a struct through a function
- Method 1, pass in by struct variable name, function modifies a copy and returns that copy, reassign the variable to that copy. E.g., `good_student = change(good_student);`
- Method 2 (elegant), passing a pointer to the struct as a parameter:

  Inside the called function, dereference the pointer passed in to get the struct itself, so we can change struct itself. E.g., `strcpy((*s).first_name, "Adam");` where s is the name of the input variable, pointer to a student structure

5.3 Pointers to Structs

- New operator to access a member of a structure through a pointer to the structure. Let p be the pointer to a student structure, then below are equivalent

```
(*p).year = 2
p->year = 2
```

- The first one needs the bracket since dot has higher precedence (access member of a pointer is not we want (neither makes sense), we want to access member of the structure pointed by the pointer)

Chapter 6

Linked Structure

6.1 Introduction

	Array	Linked Structure
Implementation	built into C language	user-defined
Access and Storage	use indices to fetch and store	requires a "traverse" function to go over elements in the structure
Size	fixed size	dynamic size

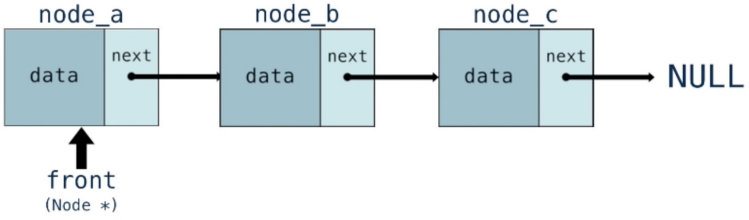
- Nodes are represented using struct

```
struct node{
    int value;
    struct node *next;
};
```

- Initialize a pointer to a node structure called `front`, as a representation of a linked list (access from front node), e.g., `front = &(node_a);` where `node_a` is initialized as struct node type.

6.2 Traversing a List

- Create a linked list in reverse order that looks like



```c
typedef struct node {
    int value;
    struct node *next;
} Node;

Node *create_node(int num, Node *next) {
    Node *new_node = malloc(sizeof(Node));
    new_node->value = num;
    new_node->next = next;
    return new_node;
}

int main() {

    Node *front = NULL;

    front = create_node(3, front);
    front = create_node(2, front);
    front = create_node(1, front);

    ...
    return 0;
}
```

- To traverse and print the element of the list, we replace following codes with `...` above

```c
    Node *curr = front;
    while (curr != NULL) {
        printf("%d\n", curr->value);
        curr = curr->next;
    }
```

6.3 Insertion into the Middle of a List

1) Find the location where want to insert (stops at the location before where we want to insert).
2) Create a new node.
3) Link the new node into the list.

Insert a new node with the value `num` into this `position` of list `front`.

```c
void insert(int num, Node *front, int position) {
    Node *curr = front;
    for (int i = 0; i < position - 1; i++) {
        curr = curr->next;
    }
    printf("Currently at: %d\n", curr->value);

    // create a new node that points to the curr->next
    Node *new_node = create_node(num, curr->next);
    // set curr->next to the new node
    curr->next = new_node;
}
```

6.4 Testing Insertion

To include all cases of insertion (begining, middle, end, illegal index), we change code from 8.3 as

```
int insert(int num, Node **front_ptr, int position) {
    Node *curr = *front_ptr;

    // beginning
    if (position == 0) {
        *front_ptr = create_node(num, *front_ptr);
        return 0;
    }

    // illegal index check included
    for (int i = 0; i < position - 1 && curr != NULL; i++) {
        curr = curr->next;
    }
    if (curr == NULL) {
        return -1;
    }

    Node *new_node = create_node(num, curr->next);
    curr->next = new_node;

    return 0;
}
```

- Since we need to change the list itself (front pointer in main), we pass in the pointer to pointer to a Node.
- We also change how we invoke the `insert`, in main.c,

  ```
      insert(2, &front, 1);
  ```

  where `front` is a pointer to the front node of the list (in 8.3 we do not need the ampersand).

Chapter 7

Streams

7.1 Introduction

`printf` writes to a stream called standard output. `scanf` reads from a stream called standard input which refers to the keyboard by default. `fprintf` writes to a specified output stream.

7.2 Redirection

Example:

```
\$ sort < names.cat > students.txt
```

The program being run is sort. Standard input is redirected from names.cat. Standard output is redirected to students.txt.

Chapter 8

File

8.1 An Introduction to Using Files

- We can open our own stream
- `FILE *fopen(const char *filename, const char *mode)`, open a file and make it available as a stream
- The second argument describe how we want to use the file,
  - "r" - File open for reading (file must exist)
  - "w" - File opened for writing (empty created if non existed, (over)write from beginning of the file)
  - "a" - File opened for appending (append to the end of file)
- `fopen` may fail for different reasons, e.g. file not exist, not having permission, if fails it returns NULL.
- Good practice to output message on standard error, like

```
    FILE score_file = fopen("top10.txt", "r");
    if (scores_file == NULL) {
        fprint(stderr, "Error opening file \n");
        return 1;
    }
```

- Close file after reading/writing, `int close(FILE *stream)` - returns 0 if successfully close the file, non-0 if fail

```
    if (fclose(scores_file) != 0) {
        fprintf(stderr, "close failed\n");
        return 1;
    }
```

8.2 Reading From Files

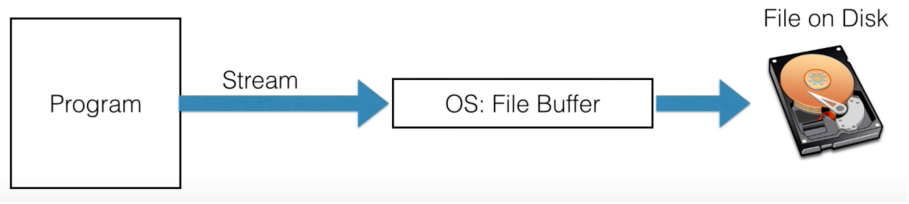
- Functions used to read data from file depends on the type of data you are reading
- When reading set or complete lines of data, fgets is often the right choice
  - `char *fgets(char *s, int n, FILE *stream)`
  - The 3rd argument is the source of data
  - first argument s is a pointer to memory where text can be stored. E.g., it can be a char array or blocks of memory allocated by malloc.
  - In success, fgets return s. fgets signals an error by returning NULL
  - 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 string. E.g., if you pass a value of 5 for n, then fgets will get at most 4 characters from the source of the data and end the string with null terminator.
  - It does not always read 4 characters $\Longleftrightarrow$ it reads **at most** n-1 (5-1) characters from the stream. It always stops reading when it reaches the end of a line of text in the stream
  - When reading from a file, think there is a cursor tracking your position in the file.
  - Before reading anything, cursor is before the first character of the file
  - After a successful `fgets`, the cursor move to the starts of the next line
  - `fgets` can also be used to read from standard input.
    - More useful than `scanf` when reading from keyboard since `scanf` stops reading at a space character whereas fgets stops reading after a newline character
    - `fgets(line, ..., stdin);`

8.3 The scanf Function

- Another function to get input from sources/streams
- `fscanf` is very similar to `scanf`, the only difference is that `scanf` is forced to read from `stdin`, whereas `fscanf` can read from any stream
- `int fscanf(FILE *stream, const char *format, ...)`, it returns number of items succesfully read, the number of items equal will be equal to number of format specifier, unless something goes wrong when reading
- `fscanf` stops reading a string at a space character
- A space + %d indicating we want read a space and then a integer fro each line
- Just name and not since name of an array is interpreted as a pointer to its first element, so name is already a char pointer. We do use total since we do need the location where `fscanf` should store the integer
- Compare the returned value to 2, if does not read 2 values, then this program ends (expect this to occur at the end of the file but it might also occur if we provide a badly formatted top 10 list)

8.4 Writing to Files

- `fprintf`
  - `fprintf(FILE *stream, const char *format, ...)`
  - Write second input (a string content) into first input (a stream).
- **How string content is written:**



  - The content is firstly written into the File buffer from the stream
  - OS will perodically write content in the buffer into file
  - Not reliable to use print function for debugging (print not fully execute in abnormal cases)
  - `int fllush(FILE *stream)` requests all content in the buffer being written into stream

8.5 Putting It All Together

Use `fscanf` and `fprintf` together. Read from a file and write content into a new file.

```c
#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;
}
```

Chapter 9

Low-Level I/O

9.1 Introducing Binary Files

- All files ultimately contain binary data. Why binary file?
  - no convenient way of storing something as text (e.g., image)
  - the intended consumer is a computer, rather than a human
  - binary format typically leads to smaller files than text format
- `fgets`, `fprinf`, `fscanf` not useful for binary file for two reasons
  - binary files have no notion of "line"
  - not for reading and writing binary data, but text instead
- extension of the binary file
  - no extension at all
  - ".dat", ".jpg", ."mp3", etc
- Opening a binary file
  - `rb`
    - Open for reading in binary mode.
    - If the file does not exist, `fopen()` returns NULL.
  - `wb`
    - Open for writing in binary mode.
    - If the file exists, its contents are overwritten.
    - If the file does not exist, it will be created.
  - `ab`
    - Open for append in binary mode.
    - Data is added to the end of the file.
    - If the file does not exist, it will be created.

  - `rb+`
    * Open for both reading and writing in binary mode.
    * If the file does not exist, `fopen()` returns NULL.
  - `wb+`
    * Open for both reading and writing in binary mode.
    * If the file exists, its contents are overwritten.
    * If the file does not exist, it will be created.
  - `ab+`
    * Open for both reading and appending in binary mode.
    * If the file does not exist, it will be created.

9.2 Writing Binary Files

- `size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);`
  - `ptr` a pointer to the data that we want to write to the file
    * typically the starting address of an array
    * can also be a pointer to an individual variable
  - `size` the size of each element that we're writing to the file
  - `nmemb` the number of elements that we are writing to the file
    * take 1 for an individual variable
    * take the number of elements for an array
  - `stream` the file pointer (open in binary mode) to which we will write
  - returns the number of elements successfully written to the file, or 0 on error.
- characters are displayed the same way in both text AND binary file
- Example: write entire arrays to a binary file

```c
#include <stdio.h>

int main(void) {
    FILE *data_file;
    int error;
    int numbers[] = {400, 800, 1200, 1600, 2000};

    data_file = fopen("array_data", "wb");
    if (data_file == NULL) {
        fprintf(stderr, "Error: could not open file\n");
        return 1;
    }

    error = fwrite(numbers, sizeof(int), 5, data_file);
    if (error != 5) {
        fprintf(stderr, "Error: array not fully written to file\n");
        return 1;
    }
```

```
    error = fclose(data_file);
    if (error != 0) {
        fprintf(stderr, "Error: fclose failed\n");
        return 1;
    }

    return 0;
}
```

9.3 Reading Binary Files

- `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);`
  - The parameters has the same meaning as `fwrite` except that `ptr` points to location storing what is read from the file.
  - `fwrite` takes a constant pointer as its first argument, `fread` takes a non-constant pointer
    - `fwrite` is writing data from memory to the file, so the memory does not need to be modified
    - `fread` is writing data from the file to memory, so the parameter cannot be const

9.4 Putting it Together: wav Files

- wav files are binary files
- Two parts of a wav file
  - header: 44 bytes of data, contains information about the wav file
  - samples: 2-byte values after the header
- Example of a wav file (`od` as file viewer for binary files):

```
>od -A d -j 44 -t d2 short.wav
0000044         2       2       2       2       2       8       8       8
0000060         8       8      16      16      16      16      16       4
0000076         4       4       4       4
0000084
```

  - `od` print out the values in a binary file
  - `-A d` translates od's output from the default of base 8 to a more convenient base 10
  - `-j 44` skips the first 44 bytes of the file
  - `d2` tells od that the file consists of two-byte values
  - `short.wav` filename
  - `0000044` starting at byte 44 of the file
  - `2, 2, 2, 2, 2, 8, ...` two-byte integers found at bytes 44, 46, 48, 50, 52, 54, ...

9.5 Increasing the Sound in wav Files

- Example: Increases the volume of a `.wav` file (specified by `argv[1]`), and saves the altered version as a file whose name is specified by `argv[2]`.

```
#include <stdio.h>

#define HEADER_SIZE 44

int main(int argc, char *argv[]) {
    char *input_name, *output_name;
    FILE *input_wav, *output_wav;
    short sample;
    short header[HEADER_SIZE];
    int error;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s inputfile outputfile\n", argv[0]);
        return 1;
    }

    input_name = argv[1];
    output_name = argv[2];

    input_wav = fopen(input_name, "rb");
    if (input_wav == NULL) {
        fprintf(stderr, "Error: could not open input file\n");
        return 1;
    }

    output_wav = fopen(output_name, "wb");
    if (output_wav == NULL) {
        fprintf(stderr, "Error: could not open output file\n");
        return 1;
    }

    fread(header, HEADER_SIZE, 1, input_wav);
    error = fwrite(header, HEADER_SIZE, 1, output_wav);
    if (error != 1) {
        fprintf(stderr, "Error: could not write a full audio header\n");
        return 1;
    }

    while (fread(&sample, sizeof(short), 1, input_wav) == 1) {
        sample = sample * 4;
        error = fwrite(&sample, sizeof(short), 1, output_wav);
        if (error != 1) {
            fprintf(stderr, "Error: could not write a sample\n");
            return 1;
        }
    }

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

    error = fclose(output_wav);
```

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

    return 0;
}
```

9.6 Reading and Writing Structs

- Example: Writing structs to a file

```
    FILE *student_file;
    struct student s;

    student_file = fopen("student_data", "wb");
    ...
    fwrite(&s, sizeof(struct student), 1, student_file);
```

9.7 Moving Around In Files

- `int fseek(FILE *stream, long int offset, int whence)`
  - change the file position
  - `stream` the stream whose position we'd like to change
  - `offset` a byte count indicating how much the file position should change
  - `whence` determines how the second parameter is interpreted
    - `SEEK_SET` move `offset` bytes after the beginning of the file
    - `SEEK_CUR` move `offset` bytes from the current file position (`+` means forward and `-` means backward)
    - `SEEK_END` move `offset` bytes before from the end of the file
- Example: Reading Structs from a file

```
    while (student_num >= 0) {
    fseek(student_file, student_num * sizeof(struct student), SEEK_SET);

    error = fread(&s, sizeof(struct student), 1, student_file);
    ...
    printf("Enter the index of the next student to view: ");
    scanf("%d", &student_num);
    }
```

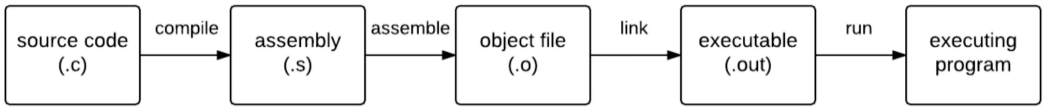
- `void rewind(FILE *stream)` (that returns nothing) is more commonly used than `fseek(my_file, 0, SEEK_SET)`, but both are valid for moving to begining of file.

Chapter 10

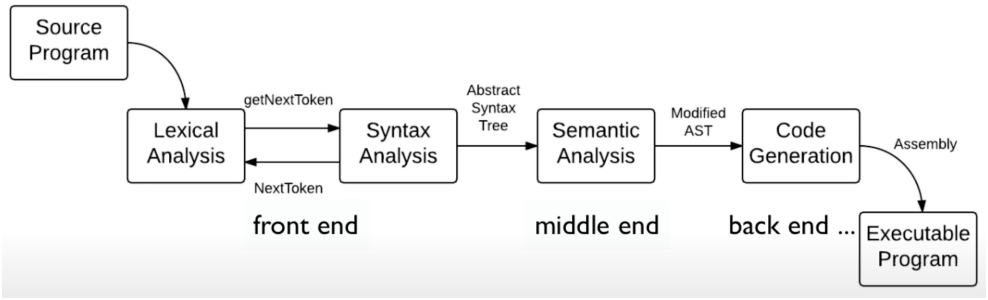
Compiling

10.1 The Compiler Toolchain

- Source code to executing program:



- Compile (Compiler)
  - A compiler is any program that translates code into one language
  - Three Phases:

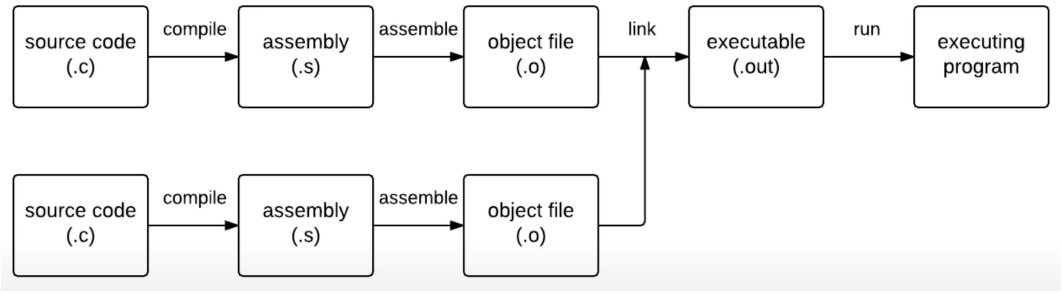


  - Using gcc to compile only: `gcc -S helloword.c`.
  - Output: `helloworld.s`, assembly code (human readable)
- Assemble (Assembler)
  - `as helloworld.s -o helloworld.o`
  - Output: Object file (machine code instructions and data)
- Link (Linker)
  - takes one or more object files and combines them to create an executable file

10.2 Header Files

Separate Compilation:

- Linking multiple object files (`.o`) to an executable:



- use `gcc -c` on source file (`.c`) to produce object file (`.o`)

```
>gcc -c compare_sorts.c
>gcc -c sorts.c
>gcc compare_sorts.o sorts.o
```

- Beneficial when working with large projects. When make change in a file (say sorts.c), then to save time

```
>gcc -c sorts.c
>gcc compare_sorts.o sorts.o
```

- Dangerous as well since when compile separately, `gcc` only checks consistency of prototypes in one file, but not the consistency of prototypes between different files (can still successfully compile).

Header file (`.h`)

- Used for type and function declaration (much like an interface). Allows gcc to inspect mismatch between file uses function prototypes from a header file and the prototypes in hearder file itself.
- Use function prototypes from an header file by

```
#include "header.h"
```

- Good for keeping project organized (e.g., prototype consistency)
- Don't include name of header file (.h) in gcc command in compilation. The `#include` statement tells the preprocessor to insert the body of the header file into the source code before the compiling even begins.

10.3 Header File Variables

Need to separate the global variable's declaration (can be in .h) and definition (in a .c)

Extern

- Stands for externally defined

CODE	DECLARE	DEFINE	INITIALIZE
`;`	No	No	No
`extern int opengenus;`	Yes	No	No
`int opengenus;`	Yes	Yes	No
`int opengenus = 1;`	Yes	Yes	Yes

- Example:
  - In a .h file, `extern const int NUM`
  - In a .c file, `const int NUM = 1`

  Now, when compile multiple files with the same header file included, no duplication error since `extern` only tells the program the existence of a variable and only one (and at least one) of source files defines and initializes the variable. Thus, only one object file allocate space for it, and after linking, every other programs can use it as well (int the sense of global).

Static

- Use 1: Make it only available in the file where it is defined
- Use 2: Keep the value across function executions
- Should rarely use `static`

Guard Condition

- Keep the header file from being included more than once in a compilation.
- Example: `sorts.h`

```
#ifndef SORTS_H
#define SORTS_H
...
#endif // SORTS_H
```

  If not defined `SORT_H`, execute next line; else jumps over next line.

- Always add guard condition in a header file

10.4 Makefiles

- Help manage the compilation process

```
OBJFILES = compare_sorts.o sorts.o

%.o: %.c sorts.h
    gcc -c $< -o $at

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

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

- Syntax

```
target: dependencies
    recipe
```

  - Execute `recipe` (commands) to make `target` (a file) if any file(s) in dependencies are newer than `target`.
  - If `dependencies` is empty, only execute `recipe` when `target` DNE.

- Wildcard

```
%.o: %.c sorts.h
    gcc -c $< -o $at
```

```
compare_sorts.o: compare_sorts.c sorts.h
    gcc -c compare_sorts.c -o compare_sorts.o

sorts.o: sorts.c sorts.h
    gcc -c sorts.c -o sorts.o

compare_sorts: compare_sorts.o sorts.o
    gcc compare_sorts.o sorts.o -o compare_sorts
```

  - `%.o:%.c` each object file needs to be built depends on a source file of the same name
  - `$<` a variable containing the first name in the list of dependencies
  - `$@` a variable containing the name of the target

- Clean

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

  - `.PHONY:` indicate that `clean` is only a legal target, not a file
  - Remove all object files

- Execute (under the same directory of `Makefile`)

```
>make
```

With no argument. Looks for a file called Makefile and just evaluate the first rule

```
>make compare_sorts
```

With argument. Looks for a file called Makefile and looks for target `compare_sorts` and evaluates it.

Chapter 11

C Features

11.1 Typedef

- Use 1: aliasing a type to an existing (built-in) type

```
typedef unsigned int size
```

  In essence, it is just providing a new name to an existing type.

- Use 2: aliasing with struct

```
typedef struct student {
    ...
} Student;

int main() {
    Student s;
    Student *p;
}
```

  - A convention: struct type itself is lowercase (student), typedef type has capital initial (Student)
  - `student` after `struct` can be dropped.

11.2 Macros

- Simple use:

```
#define label value
```

  All label are placed by value after pre-processing.

- Function use:

```
#define WITH_TAX(x) ((x) * 1/08)
```

`x` can be a expression, so remember the brackets around `x`, it guarantees expression `x` is evaluated first, and then the multiplication.

Chapter 12

The C Preprocessor

12.1 Simple Macros and Header Files

We can run c pre-processor on a source file (`.c`) by `cpp`, like

```
> cpp test.c
```

Some properties of macros:

- All defined macors are capitalized
- Just replace text with text (even text in the comments)
- Patterns found within the quote (" ") are not replaced
- Would not replace part of a word that matches a macro
- Macros are expanded even with other macros (fully expanded)

Pre-processor includes several predefined (system) macros. A convention representing they are system defined: surrounded by double underscore.

- Some are general across all systems e.g.,

  `__LINE__, __FILE__, __DATE__, __TIME__`
- Some macros are defined by specific systems, e.g,

  `__APPLE__, __gnu_linux__`
- Are useful when you need your code behave differently based on the system that is compiling and running it
- Conditional directives are supported `#if`, `#elif`, `#else`.
- Two programs below are equivalent:

```c
#include <stdio.h>

#if __APPLE__
const char OS_STR[] = "OS/X";
#elif __gnu_linux__
const char OS_STR[] = "gnu/linux";
#else
const char OS_STR[] = "unknown";
#endif

int main() {
  printf("Compiled on %s\n", OS_STR);
  return 0;
}
```

```c
#include <stdio.h>

#ifdef __APPLE__
const char OS_STR[] = "OS/X";
#elif defined(__gnu_linux__)
const char OS_STR[] = "gnu/linux";
#else
const char OS_STR[] = "unknown";
#endif

int main() {
  printf("Compiled on %s\n", OS_STR);
  return 0;
}
```

  - First program, if the macro not defined, evaluates to 0/False. (Else evaluate to the value that macro takes, typically non-zero)
  - Second program, only check if a macro is defined (`#ifdef ...` and `defined(...)`), value of macro does not matter
  - Can be utilized to set system-specific constants and to include system-specific libraries

Define macro using command line:

- At command line,

```
> gcc -D DEBUG=3 test.c
```

- In test.c we have a block code as

```c
#ifdef DEBUG
printf("Running in debug mode at level %d\n", DEBUG);
#endif
```

- The print statement in .c file would not be included after preprocessing if macro `DEBUG` is neither defined at command line argument nor in the file.

12.2 Function-like Macros

Chapter 13

Function Pointers

13.1 Introducing Function Pointers

- A function pointer points to code, not data.
- We do not allocate or de-allocate memory using function pointers.

Example of declaring `func_name` as a function pointer (either in function prototype or assignments statements):

```
    void (*func_name)(int*, int)
```

Function parameter example:

```
double some_function(int size, void (*sort_func)(int *, int)) {
    int arr[size];

    // We are calling the function received as a parameter.
    sort_func(arr, size);
    ...
}
```

Assignment statements example:

```
    void (*func_name)(int*, int) = my_func;
```

where `my_func` is the name of a function.

- `void` The return type of the function
- `func_name` The name of the function (the name of the variable)
- `int*, int` The arguments that the function takes
- We do not dereference a function pointer inside the function with input being function pointer (using `func_name` to call the function directly)

- Just pass the name of the function (**DO NOT** "make it pointer" by adding ampersand) as a parameter to function with input being function pointer
- A rule of thumb: The `func_name` is treated as a pointer to that function (just like how array name is treated as the pointer to its first element)

13.2 Extending the Worked Example

A function[1] that returns a function[2] pointer

```
    void (*func(arg1...))(arg2...)
```

- `void` The return type of function[2].
- `arg2...` The arguments of function[2].
- `func` The name of function[1].
- `arg1...` The arguments of function[1].

Example

```
    void (*parse_command_line(int argc, char *argv[]))(int *, int)
```

typedef of a function pointer

```
typedef void (*func_name)(arg...);
```

Example (returning a function pointer)

```
    SortFunc_t parse_command_line(int argc, char *argv[])
```

Example (a variable of a function pointer)

```
    void (*sort_func)(int *, int); //without typedef
```

```
    SortFunc_t sort_func; //with typedef
```

Chapter 14

System Calls

14.1 What is a System Call?

- A system call is a function that requests a service from the operating system.
  - `exit()`
  - `read()` and `write()` (low level I/O calls)
- Higher level calls
  - `scanf()`, `printf()`, `fopen()`, and `fgets()`
  - may use `read` and `write` in their implementations
  - e.g., `printf` itself is not system calls but it calls the `write` system call.
- A system call is different from (library or self-defined) function,
  - when a system call occurs, control is given to the operating system
  - the operating system executes code on behalf of the program

Chapter 15

Errors and Errno

15.1 When System Calls Fail

System calls returns a special character to indicates that an error occurs. Typically,

- system calls that returns integer, returns `-1` to indicate an error
- system calls that returns pointer, returns `NULL` to indicate an error

We don't just want to know that an error occurred, we also want to know **why** the error occurred so that we can print a sensible error message to the user.

- When a error occurs in a system call, besides returns `-1` or `NULL`, it sets a value for errno (error code) to indicate the type of error. For example, if malloc fails, it returns `NULL` and sets the value for errno to `ENOMEM`.
- A few library functions will map the error code to a string that explains the error. The one that you will use most often is perror, has prototype

  – `void perror(const char *s)`

  `perror` 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.

In other cases where `errono` is not set to be a specific error code, we still use `fprintf` print message to stderr.

15.2 Checking For Errors

Check the number of command line arguments

```
    if (argc != 3) {
        fprintf(stderr, "Usage: %s NUM FILE\n", argv[0]);
        exit(1);
    }
```

Check the value of the command line arguments

```
    long numlines = strtol(argv[1], NULL, 0);
    if (numlines <= 0) {
        fprintf(stderr, "ERROR: number of lines should be positive.");
        exit(1);
    }
```

Check `fopen` (the value of the command line arguments)

```
    if ((fp = fopen(argv[2], "r")) == NULL) {
        perror("fopen");
        exit(1);
    }
```

Check whether reach the last line of a file

```
     if ((fgets(buf, BUFSIZE, fp)) == NULL) {
        fprintf(stderr, "ERROR: not enough lines in the file\n");
        exit(1);
    }
```

Chapter 16

Process Model

16.1 Process Models

Program vs Porcess

- Program: The executable instructions, like source code or complied machine code
- Process: A running instance of a program

Each process is associated with a a data structure called process control block (**PCB**), which stores:

PID	Unique ID of a process
PC	Program Counter, identifies the next instruction to be executed
SP	Stack Pointer, identifies the top of the stack
Open file table	See chapter Pipe
Signal table	See chapter Signal

# of active processes >> # of process executing instructions at a particular instant of time. The number of **processors** (a.k.a., CPUs) on the computer determines how many processes can be executing an instruction *at the same time*.

Process State: A process will move between below states throughout its lifetime. It is in

- **Running state** if it is currently executing process
- **Ready state** if it could be executing once a CPU is available
- **Blocked state** if the process that are waiting for an even to occur. For example, may have made a read call and are waiting for data to arrive. Or, explicitly called sleep and are waiting for a timer to expire.

The role of OS in place:

- The OS gives us the illusion of running many processes simultaneously by switching between running and ready states really quickly.
- The OS scheduler is responsible for deciding which process should be executed and when

16.2 Creating Processes with Fork

- When a process calls `fork()`, it passes control to the OS. The OS create a new process by setting up a new PCB, which is almost identical to original one except below differences
  - **PID**
  - **Return value** of `fork`. If succeed,
    - parent (original) process gets PID of the new process,
    - child (new) process get 0

    If failed (due to too many processes) parent gets -1, and no child process is created.
- The starting point of execution for new process is the line of `fork()` call and onwards (since this is where PC of original process points to when `fork()` copies the PCB of the new process).
- We don't know which of parent and child process would go first after `fork()` system call - it is decided by OS.
- **Important:** the parent and child processes are completely separate processes so they don't share memory

16.3 Process Relation and Termination

Shell program:

- The program we run from the shell (its running instance - process) is a child of the shell process. The shell uses the `wait()` system call to suspend itself until its child terminate.
- If parent and child both printing out some messages, the order of the output may different for each time you run the program.
- The shell waits for the parent process to finish and then prints a shell prompt. It would not wait for the children of its child - the parent process, because it explicitly call `wait()` (see below) once for the its child only. However, you can still see some child processes printing lines in between since shell is just a process so OS needs to schedule for its termination.

Parent-Child communication via `status`:

- Calling system call `wait(&status` where status is an `int`, then we force parent (calling) process to wait until one of its child have terminated, and then the information the child terminate status is stored in the `status`. (We need to encode it in some way to get back the info, see below).

- The information in the status argument is only useful if the wait system call is successful, so we must check return value of `wait(&status)` before using `status`. E.g.,

```
if ((pid = wait(&status)) == -1) {
    perror("wait");
} else {
    printf("Child %d terminated with %d\n", pid, status);
}
```

- When a program calls `exit()` or return from the main function, then we provide a value (-1 or 0). This value makes up part of the `status` value. By convention, 0 represents a success and non-zero values represent various abnormal termination.

- There are several macros are provided to help you extract the value/info from the `status`.

  `WIFEXITED(status)` to check if the process terminated normally, returns 1 or 0. If 1, we use `WEXITSTATUS(status)` to extract the exit value of the process

  `WIFSIGNALED(status)` to check a process exit b/c a signal, returns 1 or 0. If 1, we use `WTERMSIG(status)` to to find out the signal number that caused the process to terminate

- We can add a bit more control to specify which process to wait for by `waitpid(pid, &status)`

- You can also pass in the `WNOHANG` (e.g., `waitpid(pid, &status, WNOHANG)`) option to wait pid if the parent process just wants to check if a child has terminated but doesn't want to block.

16.4 Zombie and Orphans

When a child process calls `exit()`, the OS needs to keep the existing info of the process somewhere in case the parent calls `wait()` to get this value. So OS can't delete the PCB of a terminated process until it knows it is safe to clean it up.

A **zombie** process is a process that is dead but still hanging around for its parent to collect its termination status.

But what happens if the parent never calls `wait()`? Why don't we have dozens of zombie processes using up the process table?

We call a child process whose parent has terminated the **orphans**. When a process becomes a orphan it is adopted by the `init` process

A zombie process is exorcised to put to rest when its termination status has been collected, the main task of the `init` process is to call `wait()` in a loop which collects the termination status of any process that is has adopted. After `init` has collected the termination status of an orphaned process, all of the process's data structure can be deleted and the zombie disappeared.

Zombies that are not orphan yet would eventually become orphan (after its parent terminated) hence its termination status (which is ready to be collected) will be collected by init.

16.5 Running Different Programs

The use of `exec` functions: replace the currently running process with a different executable.

Several variants of the exec function all perform the same task, just different in how the arguments to the function are interpreted.

An example:

- `execl()` — the first argument to it is the path to an **executable**, the remaining arguments to `execl()` are the command-line arguments to the executable named in the first argument. (If no argument to the executable, give `NULL` as the second argument)
- When the process calls `execl`, control is passed to the OS.
  - Before execl is executed, PCB has its
    - PC points to the the line of `execl`
    - SP points to the function contains `execl` (e.g. main)
  - After execl is executed, (same) PCB has its
    - PC points to the first line of the new program
    - SP points to the function of where the line PC points to
- If `execl` is in success they never return back to this program, but if it fails it would return hence we can do error checking.
- `execl` does not create a new process (this is the job for `fork`), but instead it asks OS to modify calling process

Variants are distinguished by

- `l` for list: Command line arguments passed as a list of arguments to exec
- `v` for vector: Command line arguments pass as an array of string (like `char *argv[]`)
- `p` for path: the PATH variable is searched for the executable
  Without `p` (`execl` and `execv`): expects that the first argument is the full path to the executable
- `e` for environment: Additionally pass in an array specifying the environment variables

Chapter 17

Signals

17.1 Introduction to Signals

ctrl c = force termination
ctrl z = suspend
To wake up from suspend (ctrl z), call program `fg`

Signals are mechanism that allow a process or the operating system to interrupt a currently running process and notify it that an event has occurred.

Each signal is identified by a number between 1 and 31, and defined constant (e.g., `SIGNINT`) are used to given them names.

Each one is associate with **a default action** that a process will perform when it receives that signal. E.g., when we type control c, the terminal sends `SIGNINT` signal to the process, and the default action is for the process to terminate

To send arbitrary signals to a process, we can do it using a library function called "kill" or from the command line using a program also called kill. E.g., at terminal, running `ps` to find the process id of interest. then

- send `SIGSTOP` by `> kill -STOP pid` to suspend the process
- and sent `SIGCONT` signal by `> kill -CONT pid` to restart
- `> kill -INT pid` to terminate the process

17.2 Handling Signals

Each PCB (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. That said, we can change the behaviour of a signal by installing a new signal function written by ourselves.

The `sigaction()` system call will modify the signal table so that our function is called instead of the defualt function.
`int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)`

- `signum` is the signal being modified
- `act` is a pointer to a `sigaction` struct that we need to initialize before we call the `sigaction()`
- `oldact` is also a pointer a `sigaction` struct. The system call fills in the value of the struct with the current state of the signal handler before we change it.

A `struct sigaction` has definition:

```
struct sigaction {
    void       (*sa_handler)(int);
    void       (*sa_sigaction)(int, siginfo_t *, void*);
    sigset_t     sa_mask;
    int          sa_flags;
    void       (*sa_restorer)(void);
};
```

The first field is a function pointer for the signal handler we want to install. The others are illustrated in below example.

Procedure of installing signal handling function:

- Write a function that will execute when the process is signal, this is called the signal handler. It must have a single int parameter and have a void return type. e.g.,

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

- Next, add code to install our new function in the signal table, e.g., add following code in `main` function

```
    struct sigaction newact;
    newact.sa_handler = handler;
    newact.sa_flags = 0;
    sigemptyset(&newact.sa_mask);
    sigaction(SIGINT, &newact, NULL);
```

  - create a `sigaction` struct
  - set its `sa_handler` member to be the handler function we wrote
  - set default flags
  - set the `sa_mask` filed to empty then no signal are blocked during the handler (execution)

– `newact` is all set, call the sigaction system call to install handler for the `SIGINT` signal

When the signal handling function finishes, control returns to the process at the point where it is interrupted (if handler does not call `exit()`).

For `> kill [-QUIT, -KILL, -STOP] pid`,
the former two also terminate a process (like `-INT`), the latter two's default behaviour is not changeable (always terminates / suspend the process).

Chapter 18

Bit Manipulation and Flags

18.1 Introducing Bitwise Operations

`&&` (logical and): Look at whole number and see if both have non-zero value

Some bitwise operators:

- `&` is bitwise and operator, perform and operation on each paired bits
- `|` is bitwise or operator
- `^` is bitwise exclusive or operator
  `^` has other interpretations (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.
- `~`: bitwise negation operator (aka complement operator)
  `~0 == 1`
  `~1 == 0`
  It takes a single value and flip every bit in the variable (be careful: negate integer 0 would not become integer 1, this is a **bitwise** operator. Indeed, negate integer 0 which is 32 bits of 0's, gives 32 bits of 1's)

Store binary and hexadecimal numbers in C:

- Store the binary constant by prefacing the number with `0b`, e.g.,
  `char a = 0b00010011;`
- You can also use preface `0x` to store Hexadecimal constant,
  `unsigned char b = 0x14`
  Hexadecimal is convenient since 1 hexadecimal digit is 4 bits

18.2 The Shift Operator

- Given a variable b, set the third bit to 1 and leave the other bits unchanged.

```
int main() {
    char b = 0xC1;    // 1100 0001
}
```

  `b | 0x8` gives the result (where 0x8 = 0000 1000)

- Write an expression that results in 0 if the second bit of b is 0 and a non-zero value if the second bit is 1.

  `b & 0x4` gives the result (where 0x4 = 0000 0100)

- Restate these problems in a more general way:

  Given a variable b, set the **kth** bit to 1 and leave the other bits unchanged.

  Write an expression that results in 0 if the **kth** bit of b is 0 and a non-zero value if the second bit is 1

- We need to create a value that sets an arbitrary bit to 1, for that we will use C' shift operaters
  `<<` Shift left
  `>>` Shift right
  It takes two operends, e.g.,

  – to the left of the operator – is the value to shift,

  – to the right – tells us how many places to shift.

  E.g., `1 << 3` gives an 0000 1000 (an 8)

- With these shift operators, to solve first question in general,

```
unsigned char setbit(unsigned char var, int k) {
    var = var | (1 << k);
    return var;
}
```

```
int checkbit(unsigned char var, int k) {
    return var & (1 << k);
}
```

- The left shift will fills 0 on the right and discard bits shifted out of 8 bits on the left.
- Left shift operator works like multiplication with 2^k where k is the of bits shifted
- Right shift operator works like division by 2^k where k is the of bits shifted.

18.3 Bit Flags

Concept

- variable treated as an array of bits
- each bit represents an option (flag) that can be turned on and off
- commonly used by system calls

File permission in Linux: `-rwxr-xr-x 1 reid instrs 9710 Sep 30 2014 sb*`

- the permission string: `-rwxr-xr-x`
  - leftmost (9): file type. regular file: `-`; directory: `d`; link: `l`
  - The owner (8, 7, 6): read, write, and execute permissions
  - The members of the group instrs (5, 4, 3): read and execute permissions.
  - Everyone else (2, 1, 0): read and execute permissions.
- owner of the file: `reid`
- group: `instrs`

Example: file permission and bit flags

- ignore the first dash; need nine bits to represent each permissions setting
- `chmod`: `MODES`
  - `04 == 100` read
  - `02 == 010` write
  - `01 == 001` execute
- Generate a read-only permission for a file

```
r--r--r--
100100100
```

```
mode_t = 0400 | 0040 | 0004 = 0444
```

- Given a `mode` variable, do the group or other users have read permission?

```
mode & (040 | 004)
mode & 044
```

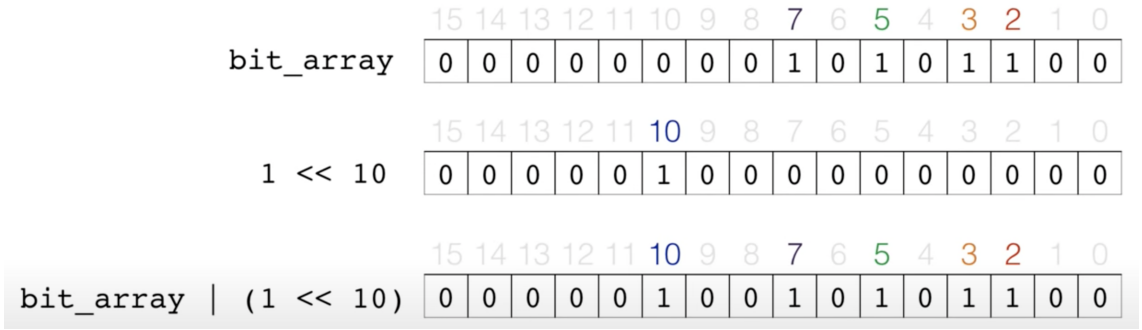
18.4 Bit Vectors

Idea: Using flag bits to implementation a set — each bit in a variable denotes the presence or absence of a element in a set. This is a very compact implementation and can quickly perform set operations using bitwise operators.

Elements of set are (represented as) indices of the array, and value at a given index location tells whether the element is in the set.

- Add 10 to `bit_array`

```
bit_array = bit_array | (1 << 10)
```



- Remove 10 from `bit_array`

```
bit_array = bit_array & ~(1 << 10)
```

- different size of bit array
  - `unsigned short bit_array`: 0 to 15
  - `unsigned int bit_array`: 0 to 31
  - `unsigned int bit_array[N]`: (0 to N-1) × (0 to 31)
- Set bit at index 34 to 1
  - Step 1: Determine which element of the unsigned int array holds the bit: `34/32==1`
  - Step 2: Determine which bit to modify: `34%32==2`

```
int index = 32 / 32;
bit_array[index] = bit_array[index] | 1 << (34 % 32);
```

- Function Examples

```
#define INTSIZE 32
#define N 4

struct bits {
    unsigned int field[N];
};

typedef struct bits Bitarray;
```

Initialize the set b to empty or all zeros

```
int setzero(Bitarray *b){
  return (memset(b, 0, sizeof(Bitarray)) == NULL);
}
```

Add value to the set b

```
void set(unsigned int value, Bitarray *b) {
    int index = value / INTSIZE;
    b->field[index] |= 1 << (value % INTSIZE);
}
```

Remove value from the set b

```
void unset(unsigned int value, Bitarray *b) {
    int index = value / INTSIZE;
    b->field[index] &= ~(1 << (value % INTSIZE));
}
```

Return true if value is in the set b, and false otherwise.

```
int ifset(unsigned int value, Bitarray *b) {
    int index = value / INTSIZE;
    return ( (1 << (value % INTSIZE)) & b->field[index]);
}
```

Chapter 19

Multiplexing I/O

19.1 The Problem with Blocking Reads

`read` on a pipe blocks until there is something to read, or the other end of the pipe is closed (then the `read` returns).

If a parent has two children, and want to read from either child (which ever is ready).

No matter how you order the `read` calls, we can always get into a situation where the parent is blocked waiting to read from one child, while the other child has data ready to be read.

`select` solves this problem, it let us specify a set of file descriptors and then blocks until at least one of them is ready for attention, then `select` tells us which file descriptors are ready so that we can avoid blocking on the non-ready ones.

19.2 Introducing Select

`select` has prototype
```
int select(numfd, read_fds, write_fds, error_fds, timeout);
```

The basic idea is `select`

- sets up a set of file descriptors to watch,
- and `select` blocks until one of these file descriptors has data to be read or until the resource has been closed (in either case calling we know that calling `read` on that file descriptor will not cause read to block), this type of file descriptor are call **ready**.
- `select` **returns** the number of file descriptors that are ready

Starts simple, we use only first two parameters, so
```
int select(numfd, read_fds, NULL, NULL, NULL);
```

- `numfd` should be set to be the value of highest file descriptor in my set, **plus 1** (so that `select`) knows our file descriptor can only be value between `0` to `numfd - 1`
- `read_fds` is the set of file descriptor to be watched, it will be modified so that when `select` returns the set only contains the file descriptors that are ready for reading.

An example:

```
fd_set read_fds;
FD_ZERO(&read_fds); \\ initialize the set of fild descriptors to be empty
FD_SET(pipe_child1[0], &read_fds); \\ add the file descriptor of pipe1 readend
FD_SET(pipe_child2[0], &read_fds);


int numfd;
// Initialize numfd to be the larger of two plus 1
if (pipe_child1[0] > pipe_child2[0]) {
    numfd = pipe_child1[0] + 1;
} else {
    numfd = pipe_child2[0] + 1;
}

if (select(numfd, &read_fds, NULL, NULL, NULL) == -1) {
    perror("select");
    exit(1);
}

// Read first from child 1
if (FD_ISSET(pipe_child1[0], &read_fds)) {
    if ((r = read(pipe_child1[0], line, MAXSIZE)) < 0) {
        perror("read");
    } else if (r == 0) {
        printf("pipe from child 1 is closed\n");
    } else {
        printf("Read %s from child 1\n", line);
    }
}

// Now read from child 2
if (FD_ISSET(pipe_child2[0], &read_fds)) {
    if ((r = read(pipe_child2[0], line, MAXSIZE)) < 0) {
        perror("read");
    } else if (r == 0) {
        printf("pipe from child 2 is closed\n");
    } else {
        printf("Read %s from child 2\n", line);
    }
}
```

The other three parameters

- `write_fds` a set of file descriptor, used to check which file descriptor is ready for writing
- `error_fds` a set of file descriptor, used to check which file descriptor is has error conditions pending

- `timeout` A pointer to struct `time_val`, can use it to set a time limit on how long `select` will block before returning, even if no file descriptors are ready.

[*Remark 1.*] It may be the case that more than one file descriptor are ready when returns.

[*Remark 2.*] `read_fds` is modified by `select` so we can not just use it again in a second `select` call (we need to re-initial the set if we read from multiple children for multiple times in a loop).

Chapter 20

Sockets

20.1 Intro to Sockets

Sockets: used for communication between two processes running on two different machines.

Internet: below two together consist of a full address in the space of internet,

- Each machine has an internet protocol (aka **IP address**)
- Each machine may run many programs, each has a **port**

Message is communicated from program on a machine to a program on another machine by both IP address (building address) and port (unit number), via routers.

A **sever** is a program running on a specific port of a certain machine waiting for another program (or maybe many other programs) to send a message.

A server provides service for webpage is `http` (so called web-server) which normally runs on port 80 (secure webpage use port 443).

A user runs a client program when they want to start interacting with a server. The client program sends a initial message

- In some cases the client sends only a single message
- In some cases the client begins a "connection" — the conversations between two machines that involves multiple messages. In this case the client sends the first message to initialize the connection. Once the communication channel is set, either machine can sent data/message to the other

We establish a communication channel using Sockets. There are a few different types of sockets (Datagram, Stream, Raw, etc.) all rely on the same system call.

We will use: Stream Sockets (builds on the TCP portocol):

- Connection oriented sockets
- No loss (of messages) guaranteed
- Delivery in order which they are sent

The first system call we need is socket, with prototype
`int socket(int domain, int type, int protocol);`
This system call is used to create endpoints (one in the client, one in the sever) for communication. So, **both** programs will independently invoke this system call.

- return value:
  - 1 if an error occurs
  - The index of an entry in the file-descriptor table if on success
- `domain`: sets the protocol (or set of rules) used for communication. Either assign it to a constant `PF_INET` or `AF_INET` (since we communicating over internet)
- `type`: We will use `SOCK_STREAM`
- `protocol`: configures which (specific) protocol the socket will use for communication. Stream socket uses the PCT protocol (this is the only protocol available for stream socket), so we just set this parameter to 0 to tells the socket system call to use the default protocol for this type of socket.

E.g., both the client and server programs will call socket like this

```
int listen_soc = socket(AF_INET, SOCK_STREAM, 0);
if (listen_soc == -1) {
    perror("socket");
    exit(1);
}
```

to create the socket end points. The file descriptors returned by the function will be used by the system calls that establish a connection.

20.2 Socket Configuration

An stream socket is set up in above (but the address is not set up yet). Now, we will configure that socket to wait for a connection at a specific address, using `bind` system call, whose prototype is
`int bind(int socket, const struct sockaddr *address, socklen_t address_len)`

- The first parameter is the socket we want to configure
- The second parameter is of type pointer to `struct sockaddr`. This type is generic, it is for all address family. But to out particular family (`AF_INET`), we will use a pointer to `struct sockaddr_in` (in for internet). It has definition

```
struct sockaddr_in {
    short   sin_family;
    u_short sin_port;
    struct  in_addr sin_addr;
    char    sin_zero[8];
}
```

We initialize it as

```
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;  // To be consistent with our socket
    addr.sin_port = htons(54321); // Any port # in 49152 - 65535 is fine)
    addr.sin_addr.s_addr = INADDR_ANY;
    memset(&(addr.sin_zero), 0, 8);
```

[*Remark 1.*] Notice the call `htons()`. It stands for host-to-network-short, converts byte order of host machine to network byte order

[*Remark 2.*] In memory, storing an integer takses more than one byte and different machine stores the bytes that makes up the integer in different orders (the little/big endian problem). The transmitting process should follow the same agreements (makes sure that different machine/programs communicate using the same "language"), we called this agreement **protocols**.

[*Remark 3.*] The third parameter that holds the machine address is a struct, the only filed in this struct is `s_addr`, and we can set it to pre-defined constant `INADDR_ANY`, it configures the socket to accept connections from any of the addresses of a machine. (**A machine** can have multiple network interface cards and can be plugged into separate networks, and it has **a different address per network**).
A machine also has an address for itself (not for a partiuclar network), called **localhost**, is 127.0.0.1 — But I can only connect to this localhost from a program running on the same machine (can't be from other machine).

[*Remark 4.*] The last field `sin_zero` is an extra padding. It makes `len(struct sockaddr_in) == len(struct sockaddr)`. (However, since when we `malloc` the space for the struct, those bytes are not reset, so we manully reset them to be 0's preventing exposing previous content to eavesdroppers).

- The last parameter is the length of the address (the second parameter) that we are passing.

An example call:

```
    if (bind(listen_soc, (struct sockaddr *) &addr, sizeof(struct sockaddr_in)) ==
        -1) {
        perror("bind");
        close(listen_soc);
        exit(1);
    }
```

Notice that the second input is cast to `struct sockaddr *`.

Now we have the socket bond to a particular port at a particular machine. Then, we use `listen` system call to let the machine start to looking for connections. It's prototype is
`int listen(int socket, int backlog);`

- The first parameter is the same socket used in `bind`
- The second parameter is the number of partial/pending connections the server can hold (listen set up a data structure that can holds `backlog` number of partial connections in maximum).

An example call:

```
if (listen(listen_soc, (struct sockaddr *) &addr, sizeof(struct sockaddr_in))
    == -1) {
    perror("listen");
    exit(1);
}
```

20.3 Setting Up a Connection

Now we will see how a server **accepts** connections from a client. The `accept` system call has prototype

```
int accept(int sockfd, struct sockaddr * address, socket_t *addrlen);
```

- The first parameter is the listening socket (`listen_soc`) that we have set up and configured.
- The second parameter is a pointer to a `struct sockaddr`, accept uses this parameter to communicate back the address of client that attempts to connect.

  I.e., when `accept` returns, the second parameter will point to a struct that holds the client address information (fields in the struct has been set by `accept`).
- The third parameter is the pointer to the length/size of client address.
- **Return value**
  - On success, an integer representing a new socket which we will use to communicate with the client. So we assign it to a variable and can use it later.
  - On failure, -1.

`accept` is a blocking system call. It waits until some clients is attempting to connect. An example call:

```
// allocate the memory for the struct as the second parameter to accept
struct sockaddr_in client_addr;
client_addr.sin_family = AF_INET;
unsigned int client_len = sizeof(struct sockaddr_in);

int return value = accept(listen_soc, (struct sockaddr *) &client_addr, &
    client_len);
```

When we run the above program, the accept is blocked to wait for someone to connect.

On the client side, first we set up a stream socket using `socket`, and then we initiate a connection over this socket to the server using `connect`, which has prototype

```
int connect(int sockfd, const struct sockaddr *address, socklen_t address);
```

- The first parameter is the socket created in the client program.
- The second parameter is the address of the socket on the server to which we want to connect.

  We will need to know server's address and port (the other set up of the struct would be similar as in server). The port is just 54321 like before. We will use a system call `getaddrinfo` to look up the IP address of a machine (on a particular internet) based on its name. `getaddrinfor` has prototype
  `int getaddrinfo(char *host, char *service, struct addrinfo *hints, struct addrinfo **result)`

  - The first parameter is the name of the host/server machine.
  - We completely ignore the second and third parameters and setting them to `NULL`
  - The last parameter is the address of a pointer to a linked list of structs (since not only one address is valid). We should pass in a pointer to `struct addrinfo` and `getaddrinfo` initialize the linked list (on the heap) for us.

An example client is:

```
int soc = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in server;
server.sin_family = AF_INET;
memset(&server.sin_zero, 0, 0);
server.sin_port = htons(54321);

struct addrinfo *result;

// Get servers' address info and store into result
getaddrinfo("teach.cs.toronto.edu", NULL, NULL, &result);

// Use the addrinfor struct (result) to set the server address.
server.sin_addr = ((struct sockaddr_in *) result->ai_addr)->sin_addr;
freeaddrinfo(result);

connect(soc, (struct sockaddr *)&server. sizeof(struct sockaddr_in));
```

20.4 Socket Communication

Pipe line overview:

- Server side: `socket` $\longrightarrow$ `bind` $\longrightarrow$ `listen` $\longrightarrow$ `accept`
- Client side: `socket` $\longrightarrow$ `connect`

Once we have the steam socket set-up between the cleint and the server, we use the socket descriptor `client_socket` just like a fild descriptor.

`\r\n` is called a network new line (different machine internet it the same way).

server.c

```c
listen_soc = socket(...)
bind(listen_soc, ...)
listen(listen_soc, ...)

int client_socket = accept(listen_soc, ...)

if (client_socket == -1) {
    perror(accept);
    exit(1);
}

write(client_socket, "hello\r\n", 7);

char line[10];
read(client_socket, line, 10);

line[9] = '\0';
printf("I read %s\n", line);

close(listen_soc);
```

client.c

```c
soc = socket(...)
if ((connect(soc, ...)) == -1){
    perror(connect);
    exit(1);
}

char buf[10];
read(soc, buf, 7);
buf[7] = '\0';
printf("I read %s\n", buf);

write(soc, "0123456789", 10);

close(soc);
```

Not that one `read` will always read all message that is written in a single `write`. Use the fact that `read` returns the number of bytes successfully read, to help you determine when you read the message in full.

Another fact about read, if you call `read` and `write` ends are all closed, `read` returns 0.

Chapter 21

Shell Programming

21.1 Part 1 (Command Line Substitution & Variables)

Some Useful Programs:

- `echo` command outputs all of its command line arguments.
- `cat` command outputs the content of a file.
- `sh` command executes a file (containing lines of commands) in shell.
- `expr` command do arithemetic in shell, e.g.

```
$ expr 4 + 1
5
```

- `read` command takes a list of variable names as arguments. So, if you type something after it executes, your input is going to get assigned to variable(s).

  It assign variable by spaces, and there are two cases:

  - More tokens than variables: it assigns the rest of input line to the last variable. (e.g., if only one variable, you type entire line with spaces, the entire line is assigned to that one variable)
  - More variables then tokens: then subsequent variable are assigned to be the empty string

Assignment Statement Syntax:

```
$ i=3
```

Remark. No spaces. `i = 3` means execute program `i` with `=` and `3` being 1st and 2nd arguments respectively.

Substitution Category 1 (with `$`): To substitute a variable with its value, use `$`.

- A simple example

```
$ echo $i
3
```

- Another example

```
echo What is your name '?'
read name
echo Hello, $name, nice to meet you.
```

Remark. At command line ? should be quoted to be considered as string, otherwise it will be considered as a filename wildcard.

Substitution Category 2 (with backquote `): What's inside backquote is interpreted as a command and being executed first. Then, the output (of that command) is captured by backquote and is substituted into this command line with the backquote module.

- An example, re-assign variable `i`:

```
$ i=`expr 4 + 1`
$ echo $i
5
```

- Hence, to perform `i = i + 1`, in shell we do

```
$ i=`expr $i + 1`
```

Run the command inside backquotes $\longrightarrow$ Substitute the right side of equal sign with the output $\longrightarrow$ Now assignment statement (e.g., `i=5`)

Variable PATH

- Variable PATH contains a list of directories that hold the programs (e.g. ls, cd) you would want the shell to search for when calling
- When you type in a command that does not contain any slashes in the command name, it looks through the list of directory listed by this variable, separated by columns, and it looks for the command in each one of the directory until it finds it.
- An example:

```
$ PATH=/bin:/user/bin:/user/local/bin
```

21.2 Part 2 (Control Constructs - If-Statement & While-Loop)

General Idea: In shell, a general idea is to **use command exit code as a boolean value**

- exit with **0** can be used as boolean **true**
- exit with **other values** can be used as boolean **false**

For exploration purpose, to check the exit status of a program after its execution, `$?` in shell. Let's see an example on boolean constant corresponds to exit values representing them,

```
$ false
$ echo $?
1
$ true
$ echo $?
0
```

Syntax of an if-statement:

```
if an arbitrary sh statement
then
    one or more statements
elif an arbitrary sh statement
then
    one or more statements
else
    one or more statements
fi
```

[*Remark 1.*] `if` in sh checks whether a program exit with success or failure, so **never** write something like

```
foo
if test $? -eq 0
then
    something
fi
```

since this is what `if` will be doing exactly. We should always go with

```
if foo
then
    something
fi
```

[*Remark 2.*] Same as in most other languages, `elif` and `else` branch is optional.

[*Remark 3.*] We can do nothing in a branch by adding sh null statement `:` after `then`. E.g.,

```
if foo1
then
    :
else
    bar1
fi
```

Evaluate Condition: `test` program can test a condition.

- Simple use:

```
$ test 2 -lt 3      # testing 2 < 3
$ echo $?           # see the exit status
0                   # represents success
```

- Combined with variable and substitution

```
$ x=5
$ test $x -lt 3      # testing 5 < 3
$ echo $?            # see the exit status
1                    # represents failure
```

- Combined with if-statement, in shell script

```
x=5
if test $x -lt 3  # testing 5 < 3, exit value != 0 so not enter then clause
then
    echo This is very surprising!
```

  When run this shell script, we expect no output since if statement evaluate to 1 (means false).

- `test` numeric comparison operators include:

  `-lt`: less than

  `-gt`: greater than

  `-eq`: equal to

  `-ne`: not equal to

  `-le`: less than or equal to

  `-ge`: greater than or equal to

- `test` string comparison operators include:

  `=`: equal to

  `!=`: not equal to

- A string comparison and numeric comparison obviously would gives different result at times, e.g., `test 03 = 3` evaluates to false (exit with non-0) but `test 03 -eq 3` evaluates to true (exit with 0).

- `test` file testing operators include:

  `-f file`: file exists and is a plain file

  `-d file`: file exists and is a directory

  `-s file`: file exists and is a plain file and is of non-zero size

  and others.

- `test` also has boolean operators many other more, see `man test`.

Syntax of an while-loop:

```
some initializations
while an arbitrary sh statement
do
    one or more statements
done
```

- An example of printing 1 to 10:

```
i=0                     # i = 0
while test $i -lt 10    # while (i < 10):
do
    i=`expr $i + 1`     # i = i + 1
    echo $i             # print(i)
done
```

- An example exhibits use of while loop and also program `read`:

  Say we have a data file called `data`

```
hello world
thank you
once upon a time
oneword
```

  And a shell script called `whileread`. Note that `read` fails (exit with non-0) when it reads until it reaches the end of file

```
while read x y
do
    echo x is $x and y is $y
done
```

  At terminal

```
$ sh whileread < data
x is hello and y is world
x is thank and y is you
x is once and y is upon a time
x is oneword and y is
```

- Another example: ...

Compound Condition: We can use `&&` and `||` operators much like in C (it also has the short-cut feature).

- An example of and (`&&`) operator

```
if test $x -gt 3 && test $x -lt 10     # same as if (x > 3) and (x < 10)
```

  `$x -gt 3` is evaluted first and

  – if it gives false, the right command would not be execute and whole condtiion is false.
  – if it gives true, the right command would execute and whole condition's truth is the truth of the second command's outcome

21.3 Part 3 (Control Constructs - For-Loop & Case-Statement)

Quoting in sh

In sh, we need to use quote or backslash to suppress the special meaning of some characters if we wanted in it be part of a string (e.g., `>`, `|`, space etc.)

Tree ways to output an actual greater than symbol (`>`) in a sentence:

- backslash to suppress the special meaning of a single following character, e.g.

```
$ echo Fwd to userhost, type: echo userhost \>.forward
```

- use single quote for entire sentence,

```
$ echo 'Fwd to userhost, type: echo userhost >.forward'
```

- use double quote for entire sentence,

```
$ echo "Fwd to userhost, type: echo userhost >.forward
```

However, single quote and double quote have different semantics:

- Double quotes suppress the interpretation of everything except for:
  - dollar sign
  - backquote
  - backslash
  - the closing double quote
- Single quote suppress the interpretation of everything except for the closing single quote

[*Remark.*] Space is also a special character (as to seperate command line argument) and its special meaning is also suppressed by quoting. For example, if we have a file named `hello world`, we should do

```
$ cat 'hello world'
```

instead of

```
$ cat hello world
```

since in the latter, `hello` and `world` will be treated as two arguments to `cat`.

Almost anytime you have a variable whose value might include a space, we want to put the variable interpolation in double quote (since double quote suppress the meaning of space but not the dollar sign). E.g.,

```
$ filename='hello world'
$ cat "$filename"
blah blah blah
```

[*Remark.*] When you use `$` to substitute `filename` with `'hello world'`, the quote is "lost", i.e., `cat $filename` is the same as `cat hello world`. That is why we want to quote the value substituted from a variable. Also notice that in this case, single quote does not work since it also suppresses the `$`, causing the desired variable substitution fails.

Syntax of for-loop:

```
for variable name in a list of string separated by spaces
do
    one or more statements
done
```

[*Remark.*] Notice that the part `in a list of string separated by spaces` in the first line is optional, we will see an example of for loop without it in below.

- Simple one:

```
for i in hello goodbye
do
    echo $i, world
done
```

Run this sh script it outputs

```
hello, world
goodbye, world
```

- Use `*`:

```
for i in *.c
do
    echo $i, world
done
```

Run this sh script it outputs

```
a1.c, world
a2.c, world
```

- Use command `seq` which is very similar to `range` in python. For example,

```
$ seq 1 4
1
2
3
4
```

Hence, two lines below are equivalent

```
for i in `seq 1 4`
for i in 1 2 3 4
```

So, let's see an more complicated script

```
sum=0                           # sum = 0
for i in `seq 1 100`            # for i in range(1, 101)
do
    sum=`expr $sum + $i`        # sum = sum + i
done
echo $sum                       # print(sum)
```

Run this sh script it outputs

```
5050
```

- Only use for variable without not explicit list of strings, to parse command line arguments

```
$ cat parsing
for i
do
    echo an argument is $i
done
$ sh parsing foo bar baz
an argument is foo
an argument is bar
an argument is baz
$ sh parsing    # no argument, the loop run 0 times, nothing is printed
```

[*Remark.*] The program receives the list of strings to loop over from the command line arguments.

Syntax of case statement

```
case some value in
    some label)
        one or more statements
        ;;
    some label)
        one or more statements
        ;;
    *)
        one or more statements
        ;;
esac
```

- An example of for loop with case statement

```
for i
do
    case $i in
        hello)
            echo Hi! Nice to see you!
            ;;
```

```
        goodbye)
            echo Thanks for visiting!
            ;;
        *)
            echo well, $i to you too!
            ;;
    esac
done
```

[*Remark 1.*] The syntax of bracket after every label mimics the listing in natural texts.

[*Remark 2.*] The label * is like the default case in other languages, which is executed if none of the above labels match.

21.4 Part 4 (File Redirection)

File Descriptors

- 0: standard input
- 1: standard output
- 2: standard error

I/O Redirection

Example 1

```
foo <file1 >file2
```

- open `file1` for input on file descriptor 0
- open `file2` for output on file descriptor 1
- The operations are before execution of program `foo`

Example 2

```
bar >>file2
```

- Open the file for append instead of overwrite

Example 3 (`<<` take your input right there from shell script text)

```
wc -l <<EOF
"It is ......,
...... "
"And, ....."
EOF
```

- It is followed by an arbitrary token (e.g. `EOF`), which when it appears later, will terminate the input (e.g., the `EOF` at the last line)
- With this syntax, the dollar sign and backquote are just be interpreted as inside double quote (so their special meaning are NOT suppressed)
- To make it more like inside a single quote (so all special meaning are suppressed), we add a back slash before the first token after `<<`. I.e., `\EOF`

Example 4

```
cat >file
```

- Write input from the terminal to `file`

Example 5

```
cat 2>file
```

- Redirect file descriptor 2 (Standard Error)
- No space between `2` and `>`

Example 6

```
cmd >file1 2>file2
```

- Redirect standard output to `file1`
- Redirect standard error to `-NoValue-`

Example 7

```
foo >&2
```

- Redirect standard output to standard error

Process Command line arguments to a sh script

- We can use `$1` and `$2` ... to get the 1st, 2nd, ... argument
- Or we can use for loop without keyword `in`, illustrated in the Part 3.

Quote and Space

```
$ echo contents of file three > 'file three'
$ cat file three
cat: file: No such file or directory
cat: three: No such file or directory
$ cat 'file three'
contents of file three
```

Variable Followed by Strings without Space

```
sed -n ${i}p file
```

- `i` is the variable name
- `p` is the string after the value of the `i` without space