CSC209 Notes

Jenci Wei

Winter 2022

Contents

1 Input, Output, and Compiling

1.1 Printing Values (printf)

To be able to use `printf`: add the standard input-output library

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

To print some text to the screen:

```c
printf("Hello, world!\n");
```

- Quotation marks around string
- String ends with `\n`

To include an `int` inside `printf`:

```c
int n = 50;
printf("%d\n", n); // 50
```

- `%d` is a *format specifier*
- Number of parameters after the string must equal the number of format specifiers

To include a floating point number inside `printf`:

```c
double n = 1.0 / 3.0;
printf("%f\n", n); // 0.333333
printf("%.3f\n"); // 0.333
```

1.2 Compiling From the Command Line

To compile `hello.c`, we type `gcc hello.c` on the command line

- The executable produced by gcc is saved as `a.out`
- To execute the program, type `./a.out`

Two ways of determining which files are executable:

1. List the files using `ls -F`, and the executable files will have a trailing asterisk, e.g. `a.out*`
2. List the files using `ls -l`, and the executable files will have `x` as the executing permission

Options for gcc

- `gcc -Wall` prints out additional warning messages
- `gcc -o` allows specifying the name of the executable file (i.e. instead of `a.out`)
- E.g. `gcc -Wall -o hello hello.c`

1.3 Reading Input (scanf)

Include the standard io library to be able to use `scanf`:

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

`scanf` takes format specifiers

```c
double n;
scanf("%lf", &n);
```

- `%lf`: long float
- Number of parameters must match number of format specifiers
- `&` gives the *memory address* of then variable `n`, then `scanf` places the value the user inputs into that location
- Can have multiple format specifiers in a `scanf` call

2 Types, Variables, and Assignment Statements

2.1 Program

Program: A sequence of instructions to a computer

Example of a program:

```
#include <stdio.h>

int main() {
    printf("Hello, World!");
    return 0;
}
```

- `int main()`: the main function is executed when we run a program
- Body of function is inside curly braces
- `printf("Hello, World!")` prints the string literal inside the quotation marks
- A semicolon `;` signifies the end of an instruction
- `return 0`: return statement tells the computer to finish executing the function
- `#include <stdio.h>` specifies that we want our program to be able to use input and output
  - This line is not an instruction, it adds code from another file to our code
  - Makes instructions such as `printf` available for our program to use
- The instructions are sent to the computer's processor sequentially in the order specified by our code

Compiler: converts C code into some machine instructions that a computer can understand

- C code needs to be compiled before we can run it

2.2 Variables

Information is stored in **memory**, which has millions of 'cells', and each cell has an address and contains a piece of information

- A program stores a value by reserving a small section of memory and giving it a name
- **Variable**: a named piece of memory, or equivalently, a placeholder for that piece of memory

Creating a variable: use *variable declaration statement*

```
int n;
```

- First specify the type, then specify the name
- Variable name may contain letters, numbers, and underscores, cannot begin with a number, and case-sensitive

Giving a variable a value: use *assignment statement*

```
n = 200;
```

- Take the value of the RHS and store it into the variable on the LHS
- Equal symbol `=` in C is the *assignment operator*
  - *Different* from `==`, which is the *mathematical equality* symbol

Expressions: ways of performing calculations, using arithmetic and logical operators

2.3 Using Variables in Expressions

Arithmetic operations in C:

addition	+
subtraction	-
multiplication	*
division	/
modulo	%

- *Operator precedence* (i.e. 'BEDMAS') applies

E.g.

```
int x, y;
x = 2; // 2
y = (x + 2) * (x + 5); // 28
```

2.4 Double Variables

The `double` data type can store floating point results

```
double n = 32; // 32
n = 99.5; // 99.5
```

- If `n` were to be an `int`, the second line would evaluate to 99
- `double` has limited precision

Division between two `int`s gives an `int`

```
double q = 9 / 4; // 2.00000 b/c RHS gives 2 and is converted to double
```

Modulo operator `%` gives the remainder of an integer division operation

```
int n = 9 % 4; // 1
```

2.5 Programming Style

- Operators should have space on both sides
- Statements should be on separate lines
- Meaningful variable names
  - `snake_case`
  - `camelCase`

- Avoid long lines
  - If we use a second line for a statement, add indentation
- Commenting
  - Comments are *not* executed as instructions

```
/* Comment that can have
   multiple
   lines */

// Single-line comment
```

2.6 Storing Characters

Character can be stored in `char` data type

```
char c;
```

Two ways of assigning

1. Assigning to a character enclosed in single quotes
2. Assigning to a numerical value
   - Each character is assigned a number
   - Most computers use ASCII, which assigns each character a value in the range of 0-255

```
char c = 'a'; // 'a'
c = 98; // 'b'
c = c + 1; // 'c'
```

3 Boolean Expressions and Conditionals

3.1 If Statements

If-else block

- If the condition is true, the `if` block is executed
- If the condition is false, the `else` block is executed

```
if (3 > 0) {
    printf("1");
} else {
    printf("2");
}  // 1
```

Comparison operators

less than	`<`
greater than	`>`
equal to	`==`
greater than or equal to	`>=`
less than or equal to	`<=`
not equal to	`!=`

3.2 Conditional Operators

Logical AND Operator

- `&&`
- Evaluates true when the left and the right of the operator are both true

```
if (gpa >= 0.0 && gpa <= 4.0) {
    printf("GPA is valid\n");
}
```

Logical OR Operator

- `||`
- Evaluates true when at least one side of the operator is true0

```
if (gpa1 == 4.0 || gpa2 == 4.0) {
    printf("One or both GPAs are 4.0");
}
```

NOT Operator

- `!`
- Negates the condition

```
if (!(gpa < 0.0 || gpa > 4.0)) {
    printf("GPA is valid\n");
}
```

Logical/Boolean operators: `&&, ||, !`
Relational operators: `<, <=, >, >=, !=`

3.3 Structuring If Statements

If block only (no else block)

- If the condition evaluates false, ignore this code and move on

Nested if statement

- If statement inside an if statement

Else if

- Behaves the same as a nested if-statement, but more readable
- Conditions are checked one at a time starting at the top
- At the *first* time a condition is true, the corresponding block is executed, and then no more conditions are checked

```
if (gpa == 4.0) {
    printf("A+ or A\n");
} else if (gpa >= 3.7) {
    printf("A-\n");
} else {
    printf("Not A\n");
}
```

3.4 Implementation of Rational and Conditional Operators

Relational operators return 1 for true and 0 for false

```
int x, y, z;
x = 4 < 5; // true
y = 5 < 4; // false
z = 2 < 3 || 5 < 4; // true
printf("%d %d %d\n", x, y, z); // 1 0 1
```

Every numeric value except for 0 is considered to be true

```
if (0) {
    printf("0\n");
}  // will not execute
if (1) {
    printf("1\n");
}  // 1
if (2) {
    printf("2\n");
}  // 2
```

4 Functions

4.1 Calling a Function

To use a function, we *call* it using its name, followed by a set of parentheses. Inside the parentheses, we provide the arguments (inputs) to the function.

Need to tell the computer where a function comes from

- E.g. to use `fmax`, we need `#include <math.h>`

`fmax` returns the larger of the two values

```
double larger_num = fmax(2, 3); // 3
```

`printf` prints the first argument, and insert the values of the subsequent arguments to the format specifiers in the first argument

```
printf("The larger number is %f\n", larger_num); // "The larger number is 3.000000"
```

We can feed expressions to function arguments, e.g.

```
double larger_num = fmax(2 * 8.1, 10 * 19.177); // 191.770000
```

- Can have variables
- Can nest function calls

4.2 Writing Functions

First declare the function as a *function prototype* (*function declaration*)

- Prototype includes a name, return type, and list of arguments
- *Function signature*: combination of the name, return type, and list of argument types

E.g.

```
double my_fmax(double x, double y);
```

- Name for each argument (i.e. `x` and `y`) may be omitted in the function declaration

Then define the function, i.e. write the instructions that will be executed when we call the function

- Header of function definition similar to function prototype
  - Followed by parentheses instead of semicolon
  - The parameters *must* have names
- Document what the function will do using comments
- Inside the curly braces is the function body
  - Parameters in the header are variables
- Return statement that returns the correct value

```c
/* Returns the larger of the two given values, x or y. */
double my_fmax(double x, double y) {
    if (x > y) {
        return x;
    } else {
        return y;
    }
}
```

Alternatively, we can declare and define the function at the same time before we use it

- So that we don't have to write a separate function prototype
- Analogous to declaring a variable and initializing it at the same time

4.3 Function Execution

When a C program starts running

1. The `main` function is executed
   - Function declarations and `#include` are instructions for the compiler, and are *not* executed
   - The stack frame for the `main` function is pushed onto the stack
2. A function call is reached
   - Its stack frame is pushed on to the stack
     - Variables introduced in the function only exist within the **scope** of that function, i.e. created when the function is called and deallocated when the function ends
     - These variables are *local* to the function
     - These variables go into the stack frame for the function
     - The function returns the value to whatever called it, then the stack frame for that function is popped from the stack, and all the local variables are deallocated
   - When we use a variable as an argument to a function:
     - The program will get the value of the variable, and then copy that value into the space allocated for the parameters
     - *Pass by value*
   - We have two sets of variables: one from the main function, and the other from the called function
     - The two sets are completely separate from each other

5 Iteration

5.1 For Loops

For loop structure:

```
for (INITIALIZATION; CONDITION; UPDATE) {
    LOOP BODY
}
```

- The initialization section is executed before the loop begins
  - Usually used to set a variable that is updated through the loop
- The update section executes *after* every iteration of the loop
- The condition is checked before the loop body is executed
  - If the condition is true, then the loop body executes
  - If it is false, then the loop terminates

For loops can be nested

- The loop variables should not have the same name

5.2 While Loops

While loop structure:

```
while (CONDITION) {
    LOOP BODY
}
```

- Condition behaves the same as the for loop
- If the condition is evaluated to true, then the loop body evaluates again

Do-while loop structure:

```
do {
    LOOP BODY
} while (CONDITION)
```

- The loop body is executed *before* the condition is checked

5.3 Break and Continue

`break` terminates the current loop iteration
`continue` causes the rest of the loop body to be skipped

6 Types and Type Conversions

6.1 Numeric Types

When we assign a `double` value to an `int`, the fractional part is dropped, or *truncated*

```
double d = 4.8;
int i = d; // 4
```

When we assign an `int` value to a `double`, then the correct value can be represented

```
int i = 17;
double d = i; // 17.000000
```

`sizeof` gives how many bytes are used by the compiler for a certain variable

- Resulting value can be printed with the `%lu` format specifier
- E.g. size of an `int` is 4, size of a `double` is 8

The largest integer an `int` can represent is the constant `INT_MAX`

- If we add to that constant, the program will result in an overflow

Trying to represent a large integer with `float` will result in a loss of precision

```
int i = 21247000000;
float f = i; // 2147000064.000000
```

- It is an *estimate* of the value it attempts to represent

If a type can hold any value another type can represent, then the former type is *wider* than the latter

- E.g. an `int` is wider than a `char`
- When we convert a type to a wider type, we can get the expected value
- When we convert a wider type to another type, the higher-order bytes may be *dropped*

6.2 Casting

When we perform division on two integers, the type of the result is integer

```
int i = 5;
int j = 10;
double k = i / j; // 0.000000 because the int 5/10 evaluates to 0 first
```

Ways to fix this:

1. Change either `i` or `j` to `double`
2. **Cast** `i` to a double before the division

   ```
   double k = (double) i / j; // 0.500000
   ```

7 Arrays

7.1 Intro

To declare an array, specify the type, the the name, followed by square brackets

```
float gpa[10];
```

- All elements of the array must have the same type
- Inside the square brackets, we specify the number of elements in the array

We can access an element of an array by providing the name of the array followed by by the specific index to access in square brackets

```
gpa[0] = 3.7;
```

- The indices start at 0
- We are assigning `3.7` to the first element of `gpa`
- No need to provide the type
- "`gpa` at 0 is assigned `3.7`"

To access the array values, use the same bracket notation

```
printf("%f", gpa[0]); // 3.7
```

Useful since the index can be a variable

7.2 Accessing Array Elements

Declaring an array and initializing it with values:

```
int arr[3] = {1, 2, 3};
```

- The memory addresses for each element of this array is 4 bytes apart
- When this array is declared, the compiler sets aside 12 contiguous bytes

The space for an array is allocated when the array is declared, and all of the elements are allocated in one place

- Therefore arrays cannot change in size
  - Can alternatively make a new larger array and copy all the elements of the original array into it
- Once we know the address where the array starts and the size of each element, we can calculate the address of each element
  - Address of the array is the address of element-0
  - Taking the last example:

    ```
    address of arr[1]
    = address of arr + 4
    = address of arr + 1 * sizeof(int)
    ```

- In general:

```
Address of arr[i] = address of arr + (i * size of one element of arr)
```

If we attempt to access an element of the array whose index exceeds the size of the array, then we get something unexpected

- C doesn't generally check whether an array access is within the bounds of the array
- We obtain what is held in the memory after the end of the array, it could be any data
- If we try to assign what is outside of an array to a value, it might replace the value that another variable is using, or cause a segmentation fault (i.e. the address that was accessed was not legal)

8 Pointers

8.1 Intro

To access the address of an object, use the ampersand `&` operator

```
int i = 5;
printf("Address of i: %p\n", &i)
```

If a variable is a pointer, then its value is a memory address

- When we declare a pointer, we need to specify the type of the value stored at that memory address

```
int *pt;
pt = &i;
```

- `pt` is a variable that will hold the address of an `int`
- The type of `pt` is "int star" or "pointer to int"
- Using the `&` operator, the address of `i` is assigned to `pt`
- "`pt` points to `i`"

When the star `*` operator is applied to a pointer, it evaluates to the value of the memory the pointer points to

```
printf("%d\n", *pt); // 5
```

- *Dereferencing* the pointer

`*` inside a declaration is part of the type, `*` inside an expression is the dereference operator

8.2 Assigning to Dereferenced Pointers

When a dereferenced pointer is on the LHS of an assignment statement, the value of the RHS should be stored in the location the pointer points to, not to the pointer itself

- The pointer does not change, but the value it points to changes
- The original variable and the dereferenced pointer are *aliases*

  ```
  int i = 7;
  int *pt = &i; // i = *pt = 7
  *pt = 9; // i = *pt = 9
  ```

- `*pt = *pt + 1` does the same action as `i = i + 1`

`int *pt = q;` is equivalent to `int *pt; pt = q;`

- *Not* `int *pt; *pt = q;`

8.3 Pointers as Parameters to Functions

Function variables are local variables

- Changing it has no effect on the argument that gave it the initial value

To mutate the value, have a pointer as the argument

- Dereference pointer to mutate the value
- Use parentheses to ensure that the order of operation is as desired

When we intend to pass an array to a function, we could declare `int sum(int arr[])` or `int sum(int *arr)`

- The latter better represents what the compiler is doing
- If the size of the array is not fixed, the size should also be passed in as a parameter
  - Since using `sizeof` on the array gives the size of the pointer

8.4 Pointer Arithmetic

When we add an integer n to a pointer whose type has size s, the result is an address bigger than the original pointer by sn bytes

```
type k;
type *p = &k;
int n;
p = p + n; // p + (sizeof(type) * n)
```

With this, we could access array elements using pointer arithmetic

```
int arr[3] = {1, 4, 9};
int *p = arr;
printf("%d %d\n", *p, *(p + 1)); // 1 4
```

We could also treat a pointer as an array

```
printf("%d %d\n", p[0], p[1]); // 1 4
```

- `p[k] == *(p + k)`

8.5 Pointer to Pointers

We could store the address of a pointer

```
int i = 1;
int *pt = &i;
int **pt_ptr = &pt;
```

- `pt` has type `int*`, therefore `pt_ptr` should have type `int**`, i.e. pointer to `int*`
- When we dereference `pt_ptr`, we get the type `int*`
- To get the original value, dereference `pt_ptr` twice

```c
int *r = *pt_ptr; // &i
int k = **pt_ptr; // 1, equivalent to int k = *r
```

We could have higher-order pointers

```c
int ***pt_ptr_ptr = &pt_ptr;
int x = ***pt_ptr_ptr; // 1
```

9 C Memory Model

9.1 Code and Stack Segments

Memory can be viewed as an array that stores all data

- This memory array is divided into segments, where each segment stores one particular type of data
- Layers:

Buffer
Code
Global Data
Heap
Stack
OS

Once the code is compiled, it is stored in the code segment of memory

- As the code executes, it calls various functions
- Each function invocation is allocated space in the stack segment to store local variables

The stack segment

- The most recent function call is at the top of the stack
- Functions are removed in last-in-first-out order
- Space for functions are allocated as *stack frames*, which has enough memory to store all the local variables
- Once a function has finished executing, we pop its stack frame from the stack and return a value to the caller
- The top of the stack is always the currently-executing function

9.2 Heap and Global Segments

If we assign a variable outside of `main`, then it is a global variable that exists everywhere

- Global variables are stored in the global variable segment
- Not connected to any particular function

The global data segment also hold other values:

- String literals, e.g. when we write `char *ptr = "Hi"`, then `"Hi"` is stored in the global data segment

`malloc` allows us to allocate memory while the program runs

- This is dynamically allocated memory
- Dynamically allocated data is stored in the heap segment
- Whenever we `free` an allocated piece of memory, it is marked as being available for allocation

The stack and the heap have maximum sizes

- If a program exceeds the maximum size of the stack or heap, there will be an *out of memory error*, or *ENOMEM*

If we attempt to access OS memory, we will get a *segmentation fault*, or *segfault*

- E.g. attempting to access OS memory
- E.g. accessing an uninitialized variable or pointer, which points to the zero address

10 Dynamic Memory

10.1 Intro

Inside a function, we can allocate space for variables on the *heap* so that they can last beyond the return statement of the functions in which they are declared

- The function `malloc` allocates memory on the heap

```
void *malloc(size_t size);
```

  - The size parameter indicates how many bytes of memory shoule be allocated
  - Its type is `size_t`, which is a type returned by `sizeof`
  - `size_t` is an *unsigned int*
  - Returns a pointer that holds the address of the memory allocated by `malloc` on the heap
  - A void pointer is used to return a pointer of generic type
  - When we store that address in a pointer, we would need an explicit type for the pointer, e.g.

```
int *ptr = malloc(sizeof(int));
```

- Heap memory remains available until the programmer explicitly deallocates it

10.2 Freeing Dynamically Allocated Memory

If heap memory is allocated in a function and the address is not returned, we would have no way to access such heap memory again since we don't have a pointer for it

- This is called a *memory leak*
- If memory leak continues, the program will eventually encounter an *out-of-memory error: ENOMEM*

To deallocate memory, use the `free` function

```
void free(void *ptr);
```

- This deallocates the entire block that was allocated in the `malloc` call that returned this address

A pointer that points to memory that has already been freed is a *dangling pointer*

- Such pointer is unsafe

10.3 Returning an Address With a Pointer

When a pointer is passed into a function as a parameter, the function creates a local pointer variable

- When we modify that pointer in the function, the pointer outside is unaffected

In the case where we want to modify a pointer, we pass a 2nd degree pointer to the function

10.4 Nested Data Structures

When we have a nested array in the heap, e.g.

```c
int **pointers = malloc(sizeof(int*) * 2);
pointers[0] = malloc(sizeof(int));
pointers[1] = malloc(sizeof(int) * 3);
```

we need to be careful when freeing memory

- Need to free the inner pointers first
- If we first free the outer pointer, then the inner pointers become dangling pointers

11 Command-Line Arguments

11.1 Converting Strings to Integers

C strings are special arrays of `char` elements

- We can declare and initialize a string literal variable directly:

```
char *s = "hello world";
```

- We can have a string of number:

```
char *s = "17";
```

  which can be interpreted as an array of `char`s: one, seven

We can use the function `strtol` to convert a string to the integer it represents

- Syntax:

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

  - First parameter: the string that we want to convert
  - Second parameter: suppose there are trailing characters not needed in the string, this parameter indicates where the ‘leftover’ piece of the string starts
  - Third parameter: the base of the number system (typically 10)

- Can handle leading spaces and leading plus/minus sign

11.2 Command-Line Arguments

`main` can have two arguments

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

- `argc` holds the number of command-line arguments
  - This is always *one more* than the number of arguments passed in
  - The first argument is the name of the program
  - The rest of the array elements are the command line arguments
- `argv`: argument vector, stores an array of strings
  - Command-line arguments are always strings
  - Use `strtol` if we want to use them as integers

12 Strings

12.1 Intro

Benefits of `string`s over array of `char`s

- Can manipulate content without using explicit loops
- Would not print anything else that follows the stored text in memory

C string: a character array that has a null character immediately after the final character of text

- The null character marks the end of text
- The null character is written as `'\0'`

To print a string, use the `%s` format specifier

12.2 Initializing Strings and String Literals

We can provide an array initializer with each of the string's characters, e.g.

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

- Yields an array with the first five characters holding "hello" and the remaining characters as null characters

We can also give the characters of the string in double quotes, e.g.

```
char text[20] = "hello";
```

- Abbreviation of the previous method
- To avoid bugs, have the number of characters *strictly* less than the specified size

We can also omit the string size, e.g.

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

- The compiler will allocate the memory size equal to the length of the string plus one more character for the null terminator

Once the array is declared, its size is fixed

To create a string literal, use pointer notation instead of array notation:

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

- `"hello"` is a string literal, which is a constant that cannot be changed
- `text` points to the first character of that string

12.3 Size and Length

Using `sizeof` on a string gives the number of bytes occupied by the array

- `sizeof` is a compile-time operation, it does not look at the contents

Many C string functions need the string header file:

```
#include <string.h>
```

`strlen` returns the number of characters in the string, not including the null terminator

- Prototype for `strlen`:

  ```
  size_t strlen(const char *s)
  ```

- `size_t` is an unsigned integer type that can be treated as an integer

12.4 Copying Strings

A string can be copied using `strcpy`

- Prototype:

  ```
  char *strcpy(char *s1, const char *s2)
  ```

- Copies the characters from `s2` into the beginning of the array `s1`
- `s1` is not required to be a string, but `s2` is required to be a string
- `strcpy` is an *unsafe function*, i.e., if we have `s1` not having enough size to fit the content of `s2`, different machines could produce different outcomes

Safe version of `strcpy` is `strncpy`

- Prototype:

  ```
  char *strncpy(char *s1, const char *s2, int n)
  ```

- `n` indicates the max number of characters that `s1` can hold, including any null characters
- Could still be unsafe, since the first `n` character of `s2` might not end with a null terminator
- To ensure that this function is safe, we explicitly add a null character at the end of `s1`

12.5 Concatenating Strings

We can concatenate strings using `strcat`

- Prototype:

  ```
  char *strcat(char *s1, const char *s2)
  ```

- Both `s1` and `s2` must be strings
- Adds `s2` to the end of `s1`

- `strcat` is an unsafe function since `s1` may not have enough space to store all the contents

Safe version of `strcat` is `strncat`

- Prototype:

```c
char *strncat(char *s1, const char *s2, int n)
```

- `n` is the maximum number of characters, not including the null terminator, that should be copied from `s2` to the end of `s1`
- `strncat` always adds a null terminator to `s1`
- `n` usually set to `sizeof(s1) - strlen(s1) - 1`

12.6 Searching With Strings

Search for single character: use `strchr`

- Prototype:

```c
char *strchr(const char *s, int c)
```

- `s` is the string to search, `c` is the character to search for
- Returns the pointer to the character that is found, or null if the character is not found
- The index can be determined with pointer arithmetic

Search for substring: use `strstr`

- Prototype:

```c
char *strstr(const char *s1, const char *s2)
```

- If `s2` is found in `s1`, then returns a pointer to the character of `s1` that begins the match with `s2`

13 Structs

13.1 Intro

Structs, i.e. structures, store collections of related data

Differences between arrays and structs:

	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

Example of struct:

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

- Use the `struct` keyword to declare a structure
- Structure tag gives the struct type a name so that we can late define variables of that type
  - Structure tag in the example: `student`
- Body of the struct declaration declares the members of the struct
  - In the example, we have four members

We can declare variables that has the type `struct student`, e.g.

```
struct student good_student;

strcpy(good_student.first_name, "Jo");
strcpy(good_student.last_name, "Smith");
good_student.year = 2;
good_student.gpa = 3.2;
```

- The word `struct` is required whenever we declare a variable of a structure type
- To access members of structs, use the name of a struct variable, a dot, then the name of the struct member that we want to access

13.2 Using Structs in Functions

When we pass a struct into a function, the function gets a copy of the struct

- Any change that the function makes is only a change to the copy, not the original struct
- Any array inside of a struct is copied

Two ways of retaining changes to a struct by a function:

1. Have the function return the changed struct
   - This method copies the struct twice, i.e. when the function is called and at the return statement

- Not preferable since structs can be large

2. Pass a pointer to the struct as a parameter

    - Preferred since nothing is copied
    - To access members using a pointer, first dereference the pointer, then use the dot operator to access the member, e.g. `(*p).gpa`
    - The arrow operator is identical to the above syntax in meaning, e.g. `p->gpa`

14 Linked Structures and Iteration

14.1 Intro

Differences between arrays and linked structures:

	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

Linked list

- Stores a sequence of items
- Has a front pointer which holds the address of the first node in the list
- Has nodes which are analogous to elements of array
- Each node contains the data stored and the next pointer, which points to the next node
- The next pointer for the last node has value null
- In C, nodes are represented using structs

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

14.2 Traversing a List

Creating a linked list

- Start from an empty list, i.e. front pointer being null
- Create nodes one at a time
- Allocate nodes on the heap

```
struct node *node_x = malloc(sizeof(struct node));
```

Can use `typedef` to shorten the type name:

```
typedef struct node {
    ...
} Node;
```

- Then there would not be a need to use `struct node` everytime we refer to this type
- Instead, we can use `Node`

Traversing the list

- Start at the front and then follow a trail of next pointers
- The null value at the last node's pointer tells us when to stop

- Traversal pattern:

```
Node *curr = front;
while (curr != NULL) {
    // some action
    curr = curr->next;
}
```

Inserting a node:

1. Create a new node
2. Duplicate the link to the new node after the insertion point
3. Replace the original link with the link to the new node

Testing

- Have four cases: middle, beginning, end, illegal index
- Write code for the last three edge cases accordingly

15 Streams

15.1 Intro

Input stream: source of data that provides input to our program

- `scanf` reads from *standard input*, which is an input stream set to the keyboard

Output stream

- `printf` writes to *standard output*, which is an output stream set to the screen
  - Used for normal program output
- *Standard error* is an output stream referred to the screen
  - Used for error output

Standard input, standard output, and standard error automatically open when a program runs

15.2 Redirection

To redirect the standard input, use the $<$ symbol when running the program, e.g. `./a.out < number.txt`

- We are redirecting standard input to read from the file `number.txt`

To redirect the standard output, use the $>$ symbol when running the program, e.g. `./a.out > result.txt`

- If a file already exists, and we use output redirection with that filename, then that file would be overwritten

Limitation: only one file can be used for input or output redirection

16 Files

16.1 Intro

`fopen` opens a file and makes it available as a stream

- Prototype:

```c
FILE *fopen(const char *filename, const char *mode)
```

- `mode` requires a string that indicates what we want to do with the file:

Mode string	Default location
"r"	File opened for reading
"w"	File opened for writing
"a"	File opened for appending

- Returns a file pointer that we will use when we want to close the file, read from the file, or write from the file
- If `fopen` fails, the it returns null

To close a file, use `fclose`

- Pass in the pointer that was previously returned from `fopen`
- Return 0 if the call was successful, and a nonzero value if failed

16.2 Reading From Files

When reading text or complete lines of data, use `fgets`

- Prototype:

```c
char *fgets(char *s, int n, FILE *stream)
```

- `stream` is the source of data
- `s` is a pointer where the text can be stored
- Returns `s` on success, and returns null when failed
- `n` is the max number of characters that `fgets` is allowed to put in `s`, including the null terminator
  - The value is usually is the desired number of characters + 1

Before reading anything, the file cursor is before the first character of the file

- After a successful call of `fgets`, the file cursor moves to the start of the next line

To read from the standard input, use `stdin` as `stream`

16.3 The `fscanf` Function

`scanf` can only read from the standard input, while `fscanf` can read from any input stream

- Prototype for `fscanf`:

```c
int fscanf(FILE *stream, const char *format, ...)
```

- Returns the number of items successfully read
- Has one extra argument than `scanf`, i.e. `stream`, that indicates which stream to read from

16.4 Writing to Files

When we intend to write into a file, use `'w'` or `'a'` as `mode` for `fopen`

- Mode `'w'` deletes the file if it already exists
- Mode `'a'` appends to the end of the file

`fprintf` is similar to `printf`, but allows specifying the stream where the output should go

When our program writes to a stream, it first writes to the *file buffer*, a location in memory controlled by the OS

- That memory is periodically written to the file on disk
- We don't know what will happen to the written content when the computer loses power in the middle of the program
- I/O for debugging is not recommended for the above reason

To ensure that modifications have been made to a stream, use `fflush`

- Prototype:

  ```
  int fflush(*FILE *stream)
  ```

- Requests that the OS write any changes that are in its buffer

17 Low-Level I/O

17.1 Binary Files

Can open with `fopen`, but append `b` to the mode

- E.g. `fopen("testing.dat", "rb");`

Extensions for binary files: no extensions at all, `dat`, `jpg`, `mp3`, etc.

Functions like `fprintf`, `fgets`, `fscanf`, etc. are not useful for binary files, since binary files have no notion of "line", and those functions read and produce text, not binary data

17.2 Writing Binary Files

Use `fwrite` to write binary data to a file

- Prototype:

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
```

- `ptr` is a pointer to the data that we want to write to the file
- `size` is the size of each element we are writing to the file
- `nmemb` is the number of elements we are writing to the file (i.e. 1 for an individual variable, or number of elements for an array)
- `stream` is the file pointer to which we will write (must refer to a stream open in binary mode)
- Returns the number of elements successfully written to the file, or 0 on error

17.3 Reading Binary Files

Use `fread` to read binary data

- Prototype:

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
```

- `ptr` is a pointer to the memory where the data from the file will be stored
- `size` is the size of one element
- `nmemb` is the number of elements to read
- `stream` is the stream to read from
- Returns the number of elements successfully read from the file, or 0 if no elements are successfully read

Using `fwrite` on one computer and attempting to `fread` on another might not work properly

- Different computers may represent data in different ways

17.4 `wav` Files

Have two parts:

1. Header
   - 44 bits of data
   - Contains information about the `wav` file, including parameters required to properly play the file in a music program
2. One or more two-byte values
   - Each two-byte value is a *sample*

To view binary files, use `od`, which prints out the values found in a binary file

- i.e.

```
od -A d -j 44 -t d2 example.wav
```

- `-A d` translates from base-8 to base-10
- `-j 44` skips the first 44 bytes of the file (i.e. the header)
- `-t d2` indicates that the file consists of two-byte values

17.5 Moving Around in Files

To move around in a file, use `fseek`

- Every open file maintains its current position
- Each read or write call moves the file position
- Such position can be changed by `fseek`
- Prototype for `fseek`:

```c
int fseek(FILE *stream, long int offset, int whence)
```

- `offset` is a byte count indicating how much the file position should change
- `whence` determines how the second parameter is interpreted
  - `SEEK_SET`: from the beginning of the file
  - `SEEK_CUR`: from current file position
  - `SEEK_END`: from end of file
- With invalid input (e.g. moving to a negative position), the `fseek` call may succeed but a later read/write attempt will fail

To move to the beginning of a file, use `rewind`

- Prototype:

```c
void rewind(FILE *stream)
```

18 Compiling

18.1 The Compiler Toolchain

Source code `.c` $\xrightarrow{\text{compile}}$ executable `.out` $\xrightarrow{\text{run}}$ Executing program

Compiler runs in 3 phases:

1. Front end
   - Lexical analysis $\underset{\text{NextToken}}{\overset{\text{getNextToken}}{\rightleftharpoons}}$ Syntax analysis
   - Source code is prepared to the front end by the preprocessor
   - Translates the source code to a language-independent intermediate representation
2. Middle end
   - Syntax analysis $\xrightarrow{\text{AST}}$ Semantic analysis
   - Optimizes code
3. Back end
   - Semantic analysis $\xrightarrow{\text{Modified AST}}$ Code generation $\xrightarrow{\text{Assembly}}$ Executable program
   - Translates the intermediate language into assembly language
     - Use `gcc -S` to compile to assembly language
   - Need to assemble the assembly code into object code, which becomes the executable
     - We can invoke the assembler using the command `as`
     - The output is not human readable – it is an object file that contains machine code instructions and data
     - Need one more step to produce the executable: *linking*
   - The linker takes one or more compiled and assembled object files and combines them to create a file that is an executable format
   - The final executable file is a package that contains all of the instructions in the program
     - This executable is not portable – we cannot copy it to another machine and expect the same behaviour
4. The executable needs to be loaded into the memory before we execute it
   - The loader does this job

Source code (`.c`) $\xrightarrow{\text{compile}}$ assembly (`.s`) $\xrightarrow{\text{assemble}}$ object file (`.o`) $\xrightarrow{\text{link}}$ executable (`.out`) $\xrightarrow{\text{run}}$ program

18.2 Header Files

When compiling the program, we have to list all of the files that contain the code used by the main program

- Each file is compiled to object file separately, then they are combined during the linking stage
- We can also use *separate compilation*, where we compile the source codes into object files separately, then use `gcc` to link them
  - To compile the source code into object files, use `gcc -c`

- E.g.

```
$ gcc -c one.c
$ gcc -c two.c
$ gcc one.o two.o
```

- Can be advantageous when we want to change one file in a large project
- Also can be unsafe since the changed object file may not be incompatible with the other files (e.g. function parameter types become incompatible) and the linking could still succeed

We can increase the organization of our projects using header files

- A header file (`.h`) is an interface
- A header should declare what functions do and what types they require, without defining how they are actually implemented
- Contains function prototypes
- To use the header file, include it in the source code, i.e. `#include "header.h"`
  - Double quotation means that we want to use the header file in the *current* directory
- When the declaration in the header file and the definition in the source file don't match, the type mismatch will be detected by the compiler
- We don't need to supply the name of the header file to gcc
  - The `#include` statement tells the preprocessor to insert the body of the header file into the source code

18.3 Header File Variables

We can separate the *declaration* and *definition* of variables

- When we declare variables in header files, we need to add `extern` keyword (which stand for "externally defined")

When we want a global variable to exist in one file only, use the `static` keyword

- Makes the variable local (only in the file that defines it)

We can add a guard condition when importing header files so that we don't double import (which leads to errors regarding duplicates)

- E.g.

```
#ifndef TEST_H
#define TEST_H
// ...some declarations...
#endif
```

Dependency: source files relying on the contents of the header file

If `static` is used on a local variable, it then keeps its value across function executions

18.4 Makefiles

When a header file changes, source file that depends on it needs to be recompiled

Makefiles are composed of a sequence of rules

- Each rule has the following structure:

```
target: dependencies...
	recipe
```

- Target: the file to be constructed
- Recipe: the command or list of commands to execute that creates the target
- If dependencies (or prerequisites) are present, the recipes are executed if one or more of the dependencies are newer than the target
- If there are no dependencies, the actions are only executed if the target does not exist
- The whitespace before the recipe is a *tab*
- E.g.

```
test: test_1.c test_2.c
	gcc test_1.c test_2.c -o test
```

- The makefile has filename `Makefile`
- When we run the command `make`, the OS looks for the `Makefile` file and checks the rules it contains
  - If there is no file named `test`, then the action for the `test` rule executes
  - If `make` is ran again, since `test` does not need to be rebuilt, nothing is done
  - If `make` is ran and either of `test_1.c` or `test_2.c` is modified since the last built, the action executes

We could use makefiles to take advantage of separate compilation

- E.g.

```
test_1.o: test_1.c test_1.h
	gcc -c test_1.c -o test_1.o

test_2.o: test_2.c test_2.h
	gcc -c test_2.c -o test_2.o

test: test_1.o test_2.o
	gcc test_1.o test_2.o -o test
```

- The executable depends on the object files
- Each object file depends on the relevant source files and header files

Each time `make` evaluates a rule, it first checks all the dependencies; if a dependency is also a target in the makefile, it will evaluate that rule first before checking the dependencies

Makefile supports wildcard

- E.g.

```
%.o: %.c %.h
    gcc -c $< -o $@
```

- Percent sign means that each object file that needs to be built depends on a source file (and header file) of the same name
- `$<` is a variable containing the first name in the list of dependencies
- `$@` is a variable containing the name of the target

We can add a rule that cleans up

- E.g.

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

- `.PHONY` indicates that clean isn't a file
- `make clean` executes the remove command

Can declare variable in makefile

- E.g.

```
OBJFILES = test_1.o test_2.o

test: $(OBJFILES)
       gcc $(OBJFILES)
```

- When a new file is added to the project, we can just update the variable

19 Useful C Features

19.1 Typedef

Typedef allows creating aliases for types and is evaluated at compile time

- E.g. `size_t` is defined in `stddef.h` as

  ```
  typedef unsigned int size_t;
  ```

- `typedef` allows defining a name (i.e. `size_t`) that refers to an existing type (i.e. `unsigned int`)
- This provides a new name for an exising type

Usually used for structs

- E.g.

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

  allows using `Student` instead of `struct student`

- The first `student` can be omitted

19.2 Macros

Macros create aliases that are evaluated during preprocessing

- E.g.

  ```
  #define MAX_NAME_LENGTH 40
  ```

- `#define` tells the preprocessor to replace occurrences of `MAX_NAME_LENGTH` with `40`
- By convention, all defined macros are capitalized

Macros are useful for constants

- Increases readability
- Constants can be easily updated

Macro language is not C

- We don't need equal signs or semicolons

Macros can take parameters

- E.g.

  ```
  #define WITH_TAX(x) ((x) * 1.13)
  ```

  Usage:

  ```
  printf("%f\n", WITH_TAX(9.99)); // replaced to printf("%f\n", ((9.99) * 1.13));
  ```

- Behaves like a function, but more efficient since it happens before compilation
- Cannot have space between macro name and parentheses
- Inside the macro definition, place parameter name (i.e. `x`) and the entire definition in parentheses
  - Need to ensure that the parameter is fully evaluated before other operations
  - E.g. if `1 + 1` is substituted in for `x`, then we need this addition to be evaluated first

20 The C Preprocessor

Any symbol beginning with `#` is a *preprocessor directive*

- Used to modify C source code before it is compiled
- E.g. macros

To execute the preprocessor on a c file, use command `cpp`

- E.g.

```
$ cpp test.c
```

- The preprocessor transforms the code by executing all the directives and expanding all the macros
- Usage of macros in the code are replaced text with text (string substitution), even text in comment
- Patterns found within quotes are *not* replaced
- Part of a word that matches the macro is *not* replaced
- Macros can expand within other macros – we can use a macro within the definition of another

The preprocessor includes several predefined macros

- E.g.

```
__LINE__, __FILE__, __DATE__, __TIME__
```

- System-defined macros are surrounded by double underscores

Some macros are defined by specific systems

- We could use this to check what system is running our program

A full set of conditional directives are supported, including `#if`, `#elif`, and `#else`

- E.g.

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

- Each block is terminated by the start of the next block, and the last block is terminated by `#endif`
- Can use `#ifdef` to check if the macro is defined, e.g.

```c
#ifdef __APPLE__
```

  is equivalent to

```c
#if defined(__APPLE__)
```

- Can be utilized to set system-specific constants and to include system-specific libraries

We can define macros on the command line using the `-D` flag

- E.g.

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

  where the c code contains

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

When we use `#include`, the header file is copied into the source file

- We might run into issues when the same variable is defined in different header files, or when a header file is included twice (e.g. A and B are included, and B instructs the preprocessor to include A (again))

Almost everything that can be done with macros can be done within the C system

- C functions are preferred to function-like macros

21 Function Pointers

We can pass functions into arguments or store functions in structures

The type of a function is its return type and its parameter types

- E.g.

  ```
  void ...(int *, int)
  ```

  To make it a pointer to a function:

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

- When we call a function via its pointer, we do *not* need to dereference it
- When we pass functions into arguments, we just pass its name, as if it is a normal variable
- A function name is treated as a pointer to the function

22 System Call

System call: a function that requests a service from the OS

- E.g.

```c
void exit(int status);
```

- `printf` is a library function, which is on a high level
  - `printf` → parse the format string and construct output string → set up buffer and copy output string to buffer → `write`
  - `write` is a system call
- When a system call occurs, control is given to the OS and the OS executes code on behalf of the program
- Library functions, e.g. `printf`, `scanf`, `fopen`, `fgets`, call system calls as part of their operation

23 Errors and Errno

E.g. when we use `fopen`, the file might not exist or might have the wrong permissions, in which cases the program might crash

- To indicate whether an error occurred, return a special value
  - -1 for integer
  - Null for pointer

To indicate error type, use the global variable `errno` which stores the type of the error

- `errno` is an `int`
- Different numeric codes are defined for different types of errors
- E.g. if `malloc` fails, it returns null and sets `errno` to `ENOMEM`

We can use `perror`

- Prototype:

```c
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`
- Main purpose is to display an error message based on the current value of `errno`
- Should *not* be used as a generic error reporting function, instead use `fprintf` to standard error

When using system calls or library functions, check for errors (by checking the return value) before moving on

- When an error is encountered, informative error messages are printed (rather than segmentation fault or a random result)

24 Processes

24.1 Process Models

Program: The executable instructions of a program

- E.g. source code, compiled machine code

Process: running instance of a program, including:

- Machine code of the program
- Information about the current state of the process, e.g.
  - What instructions to execute next
  - Current values of variables in memory

Process

- Each process has a **process ID (PID)**
- Each process is also associated with a data structure called a **process control block (PCB)** (or task control block) that stores
  - Current values of important registers
    - Include the **stack pointer (SP)**, which identifies the top of the stack
    - Include the **program counter (PC)**, which identifies the next instruction to be executed
  - Open file descriptors
  - Other states that the OS manages

To see the active process on the machine, run command `top`

- We can see the PID and the command program that the process is executing
- The process with PID 1 that is associated with command `init` is a special OS process
- The number of processors (i.e. CPUs), determines how many processes can be executing an instruction at the same time
- The processes that are currently running are in the **running state**
- The processes that could be executing if a CPU were available are in the **ready state**
- The processes that are waiting for an event to occur are in the **blocked state** (or sleeping state)
- The OS scheduler is responsible for deciding which process should be executed and when

24.2 Creating Processes with Fork

When a process calls `fork`, it passes control to the OS

- To duplicate a process, the OS copies the original process's address space, its data, and the PCB
- 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 the program counter and stack pointer
- Differences between the two processes:
  - The newly created process has a different PID

- The return value of `fork` is different in the two processes

- The original process and the newly created copy are related
  - The original process is the **parent process**
  - When `fork` is called, a **child process** is created
- When the child process runs, it starts executing after `fork` returns
- We don't know whether the parent process or the child process executes first
- The return value of `fork`:
  - Child process's PID for the parent process
  - 0 for the child process
  - -1 if `fork` fails, 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
- The parent and child processes do not share memory

24.3 Process Relationship and Termination

`getpid` returns the PID of the current process

`getppid` returns the PID of the parent process

`usleep` sleeps the current process

The OS treats the parent process and the child process the same way, and does not prioritize any of them

When we run a program, the shell waits until the process has finished and then prints a prompt for the next command

- When the original parent process finishes, a shell prompt is printed
- Since the shell is a process that the OS has to schedule, a few child processes could print some output before the shell prints its prompt
- By the time the shell prompt is printed, it is possible that some child processes haven't finished yet, so they print more afterwards

We can use the `wait` system call to force the parent process to wait until one of its children has terminated

- The shell uses the `wait` system call to suspend itself until its child terminates
- We need to call `wait` for *each* child that was created to wait for all of the child processes
- Return value of `wait`:
  - PID of the child that terminated on success
  - -1 on failure
- The return value of the child process is a part of the `status` value in `wait`
  - Status of 0 represents a successful run of the process
  - Nonzero status represents various abnormal terminations

    - There are other parts to status, e.g. part that represents how the processes terminated, whether normally or by a signal (i.e. Ctrl-C)

- To exit abnormally, use `abort()`

We can use `waitpid` to specify which child process to wait for

Can only wait for child processes, not unrelated processes such as child of child

24.4 Zombies and Orphans

When the child process terminates before the parent calls `wait`, then the state of the child process is `Z` (which stands for zombie)

- The child process has already called `exit` so it's dead
- The OS keeps this exit information in case the parent calls `wait` to get this value, so the OS cannot delete the PCB of the terminated process until it knows it is safe to clean it up
- A **zombie process** is a process that is dead, but is still there to wait for the parent to collect its termination status
- A zombie process is exorcised 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 it has adopted
    - After `init` has collected the termination status of an orphan process, all of the process data structures can be deleted, and the zombie disappears

When the parent process has terminated but a child process has not, then the child process is an **orphan**

- When we use `getppid` on orphan processes, we get 1
    - The process with PID 1 is the `init` process, i.e. the first process that the OS launches
- When a process becomes an orphan, it is adopted by the `init` process

24.5 Running Different Programs

We want to load and execute other programs

The `exec` family of functions replace the currently running process with a different executable

- Variants of the `exec` functions all perform the same task, but differ in how the arguments to the function are interpreted
- `execl`: the first argument is the path to an executable, and the remaining arguments are the command-line arguments to the executable
    - If we do not have any command-line arguments, we pass `NULL` as the second argument
    - When a program calls `execl`, the OS does the following:
        * The (machine) code is loaded to the code region of the address space
        * The program counter points to the `execl` function
        * The stack pointer points to the main function stack frame
    - While performing the `execl`, the OS finds the file containing the executable program and loads it into memory where the code segment is

- A new stack is also initialized
- 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 `execl` is gone, so it should never return and the lines following `execl` should never execute
  - `execl` will return if an error occurs and it is unable to load the program
- The OS does *not* create a new process, instead, the calling process is modified
  - The process has the same PID after `execl`
  - It retains some state from the original process, such as open file descriptors

`exec` variants

- `execl`
  - l – list: command line arguments passed as a list of arguments to `exec`
- `execv`
  - v – vector: command line arguments passed as an array of strings
- `execlp` `execvp`
  - p – path: the PATH variable is searched for the executable
  - Without `p`, the first argument is expected to be a full path to the executable
- `execle` `execvpe`
  - e – environment: additionally pass in an array specifying the environment variables

Shell process

- When we type a command at a shell prompt, the shell first calls `fork` to create a new process, then calls `exec` to load a different program into the memory of the child process
- 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

25 Pipes

25.1 Unbuffered I/O

We can use the `strace` command to see which system calls we made

- Many printing are combined into one big `write` system call
- In the system call `write`, 3 is passed in as the first parameter (file descriptor parameter)
  - The 3 files that are automatically opened are standard in, standard out, and standard error
  - Each of them have file descriptor 0, 1, 2, respectively
- File descriptor is contained in the `FILE` struct

25.2 Intro to Pipes

Pipes can be used to send data between related processes

A pipe is specified by an array of two file descriptors

- One for reading data from the pipe
- One for writing data to the pipe

When a program calls the `pipe` system call, the OS creates the pipe data structures, and open file descriptors on the pipe

- These two file descriptors are stored in the array of two integers that we pass into the pipe
- Index 0 of the array is the file descriptor used for *reading* from the pipe
- Index 1 of the array is used for *writing* to the pipe
- After the `pipe` call returns, the process can read and write on the pipe

When `fork` makes a copy of an existing process, it also makes a copy of all open file descriptors

- After the `fork` call, both processes have read and write file descriptors on the pipe

Pipes are unidirectional

- One process writes to the pipe and another process reads from the pipe
- Need to close the file descriptors that is not used
- When the write file descriptors on the pipe are closed, a read on the pipe will return 0, indicating that there are no more data to read from the pipe

When a process exits, all of its open file descriptors are closed

25.3 Concurrency and Pipes

Producer-Consumer problems

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

Pipe is a queue data structure in the OS

- Process writing to the pipe is the producer
- Process reading from the pipe is the consumer

Pipes and Producer-Consumer problems

- Case 3 will not happen since the OS only allows one to operate at a given time
- The OS blocks the read call when the pipe is empty
- The OS blocks the write call when the pipe is full

25.4 Redirecting Input and Output with `dup2`

`dup2`: a system call that makes a copy of an open file descriptor

- Function prototype:

```c
int dup2(int oldfd, int newfd)
```

- When a child process is created, even though its file descriptor table is separate from its parent's, they can point to the same objects
- `dup2` takes 2 indices in the file descriptor table and resets one to refer to the same file object as the other

26 Signals

26.1 Intro

Each signal is identified by a number between 1 and 31, and defined constants are used to give them names

- When we type Ctrl-C, the SIGINT signal is sent, and the default action is for the process to terminate
- When we type Ctrl-Z, the SIGSTOP signal is sent, and the default action is for the proess to suspend execution
- We could use another shell window to send a signal to a program
  - To suspend the process:

    ```
    kill -STOP (PID)
    ```

  - To continue the process:

    ```
    kill -CONT (PID)
    ```

  - To terminate the process:

    ```
    kill -INT (PID)
    ```

- We could utilize the `kill` library and send signals to one C program with another

26.2 Signal Handling

The PCB contains a signal table

- Each entry in the signal table contains a pointer to code that will be executed when the OS delivers the signal to the process
  - Such code is called the *signal handling function*
- We can change the behaviour of a signal by installing a new signal handling function
- The `sigaction` system call modifies the signal table so that the desired funciton is called instead of the desired action
  - Prototype:

    ```c
    int sigaction(int signum, const struct sigaction *act,
        struct sigaction *oldact)
    ```

  - `signum`: the signal being modified
  - `act`: pointer to a struct that we need to initialize before calling `sigaction`
  - `oldact`: pointer to a struct, but its value is filled in by the system call as the current state of the signal handler before we change it

`sigaction` struct

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

- `sa_handler`: the function pointer for the signal handler we are installing

Two signals that cannot be changed: SIGKILL and SIGSTOP

- SIGKILL causes the process to terminate
- SIGSTOP suspends the process

27 Bit Manipulation and Flags

27.1 Bitwise Operations

Performs logical operations by looking at every single bit

Bitwise AND

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

Bitwise OR

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

Bitwise XOR

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

Bitwise negation/complement

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

We can store binary constants by prefixing the value with `0b`

```
char a = 0b00010011; // 0x13 in hex
```

We can store hexadecimal constants by prefixing the value with `0x`

```
unsigned char b = 0x14; // 0001 0100 in binary
```

27.2 Shift Operators

Two Shift Operators

- `<<` shift left
- `>>` shift right
- E.g. `x << y`
  - `x` is the value to be shifted
  - `y` is how many places to shift

27.3 Bit Manipulation and Flags

File Permissions

- Each file has an owner and a group
- Can be checked using `ls -l`, e.g.

```
-rwxr-xr-x 1 reid instrs 9710 Sep 30 2014 sb*
```

- First column in the output is the permission string, representing who can read, write, or execute the file
  - E.g. the owner can read, write, and execute the file; the group can read and execute the file; everyone else can read and execute the file
- Third column is the owner of the file
- Fourth column is the group
- We can set separate permission for the owner, the group, and every other user in the system
- A directory would have `d` as the first letter instead of `-`, and a link would have an `l`
- The 9 characters representing permission can be represented by bits
  - E.g.

Bit Number	8	7	6	5	4	3	2	1	0
Bit	1	1	1	1	0	1	1	0	1
Permission Character	r	w	x	r	-	x	r	-	x

The `chmod` System Call

- Modes for permissions described in *base-8* (octal)
  - Base-8 is convenient for permissions since each digit can be represented by 3 bits
  - An octal in C is written with a preceding zero
- To set the bits, we can use *bitwise or*
- To check the bits, we can use *bitwise and*

27.4 Bit Vectors

A Set Implementation

- Stores a set that contains small positive integers
- No duplicates allowed
- Can be represented by an array of bits, where the elements of the set are the indices into the array, and the value of each location tells us whether an element is present
- To add an element n into the set:

```
bit_array = bit_array | (1 << n);
```

- To remove an element n from the set:

```
bit_array = bit_array & ~(1 << n);
```

- This technique is called *bit masking*
- To have a large enough set, we can use an array of `unsigned int`s, where each element (i.e. `unsigned int`) can store 32 values

Operations on the Implementation with Array of `unsigned int`s

- To set a bit at index n to 1:
  1. Determine which element of the unsigned int array
     ```
     int index = n / 32
     ```
  2. Determine which bit to modify
     ```
     int bit = n % 32
     ```
  3. Perform the operation
     ```
     bit_array[index] = bit_array[index] | 1 << bit
     ```
- We can wrap the array in a struct
- Other operations can be implemented similarly using bitwise operations

28 Multiplexing I/O

28.1 The Problem with Blocking Reads

When the parent reads from a pipe while there is nothing on the pipe yet, the `read` call is blocked unitl the child writes to the pipe

If the parent is reading from two pipes, and its `read` calls are one after another in a loop:

- Could work fine
- If the first child does not write anything to the pipe, while the second child writes much to the pipe, the parent would wait for the first child for ever

28.2 `select`

`select` Systam Call

- Prototype:

```
int select(numfd, read_fds, write_fds, error_fds, timeout)
```

- The caller specifies a set of file descriptors (i.e. `read_fds`) to watch
- `select` blocks until one of these file descriptors has data to be read or until the resource has been closed
- A file descriptor with data to be read or with a closed resource is *ready*
- `select` modifies the descriptor set so that when it returns, the set only contain the file descriptors that are ready for reading
  - There could be more than 1 file descriptors that are ready
- `read_fds` has type `*fd_set`
- `numfd` needs to be set to the value of the highest file descriptor in the set +1
- `write_fds` and `error_fds` are also sets of file descriptors that we can use to check which file descriptors are ready for writing or have error conditions, respectively
- `timeout` is a pointer to `struct timeval` that indicates how long `select` will block before returning, even if no file descriptors are ready
- Since `read_fds` is modified by the `select` call, we cannot simply reuse the function call, instead, we need to reinitialize the set before calling `select` again

`fd_set`

- `FD_ZERO` takes the address of an `fd_set` and set all its elements to 0
- `FD_SET` adds the first parameter (file descriptor) to the second parameter (pointer to `fd_set`)
- `FD_ISSET` checks whether the first parameter (file descriptor) is in the second parameter (pointer to `fd_set`)

29 Sockets

29.1 Intro

Internet

- Each machine has an internet protocal, or *IP address*, that is used to send a message to it from any other machines connected to the internet
- A machine could be running many different processes that needs to communicate over the internet
- To specify the program, besides the IP address, we also need the *port*
- Full location of a program running on a machine connected to the internet is the machine adderss plus the port
- Messages sent from one machine is enclosed in *packets*, which contain both the address and the payload (i.e. the content)
- When the packet leaves the machine, it is received by the *router*, that facilitates transfer of packets between networks
- Routers are connected to multiple networks and know which network the packet should be sent to in order to get it closer to its final destination

Client and Server

- A *server* is a program running on a specific port of a certain machine waiting for another program to send a message
  - They usually have defined ports
  - Web pages are typically served on port 80
  - Secure web pages use port 443
  - The person running the server publishes the machine address and port
- A user runs a *client* program when they want to start interacting with a server
  - The client either sends a single message, or begins a *connection* (i.e. a conversation between the two machines that involves multiple messages)
- Once the programs have established a communation channel, then either machine can send data to the other
- To establish a communication channel, we will use **sockets**

Sockets

- Has many types
  - Datagram sockets
  - Stream sockets
  - Raw sockets

Stream Sockets

- Built on the TCP protocal
- Connection-oriented sockets

- Guarantee that message will not be lost in transit, and that messages will be delivered in the order in which they are sent

The `socket` System Call

- Prototype:

```c
int socket(int domain, int type, int protocol)
```

- Used to create an endpoint for communication
- When everything is set up, we need 1 endpoint in the client program, and 1 endpoint in the server, so both programs will independently invoke this system call
- Return value is -1 on error
- On success, return value will be the index of an entry in the file descriptor table
- `domain` sets the protocol (i.e. set of rules) used for communication
  - Usually set to `PF_INET` or `AF_INET`, which are defined to be the same
- `type`
  - For stream sockets, set this parameter to `SOCK_STREAM`
- `protocol`
  - TCP is the only available protocol for stream sockets, set this parameter to 0 (which indicates that we are using the default protocol for this type of socket)

29.2 Socket Configuration

To set the address, use the `bind` system call

- Prototype:

```c
int bind(int socket, const struct sockaddr *address, socklen_t address_len)
```

- `socket`: the socket that we want to configure
- `address`
  - Pointer to `struct sockaddr` bind works for all the different address families
  - For our particular family `AF_INET`, we set this parameter by using a `struct sockaddr_in` (where in stands for internet)
    - Definition of `sockaddr_in`:

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

  - Set `sin_family` to `AF_INET`
  - `sin_port` is where we set the port number

        * Port numbers range from 0 to 65535
        * Port numbers 0-1023 are reserved fror well-known services
        * Port number 1024-49151 are *registered ports*, i.e. we can register with IANA (Internet Assigned Numbers Authority)
            · The IANA also looks after assigning domain names at the highest level
        * Port numbers 49152-65535 are *dynamic ports*. If we are writing a server on our own machine, any port in this range is fine
        * If we are writing a server to run on a shared machine, then avoid setting up a socket on a port that some other program is already using
            · In which case we need a plan of who uses which port
        * If we use port n on a shared machine, we will set the `sin_port` to `htons(n)`
            · `htons`: host to network short
            · Since different machines store the bytes that make up an integer in different orders, we need to ensure that the two machines that are communicating can understand each other
            · The two machines must be transmitting and expecting particular data in a specific format, these agreements are called **protocols**
            · `htons` convert the integer form the byte order of the host machine to the *network order* (as defined by the protocol)
    – `in_addr` is a struct, and we only need to set its `s_addr` field
        * Set to `INADDR_ANY`, which configures the socket to accept connections from any of the addresses of the machine
            · A machine can have multiple network interface cards and can be plugged into separate networks, resulting in having different IP addresses in each network
            · The machine also has an address for itself, 127.0.0.1 (i.e. localhost)
    – `sin_zero` is extra padding, making the `sockaddr_in` struct the same length as `sockaddr` struct
        * When we `malloc` space for this struct, these bytes are not reset in any way
        * We can use `memset` to set these 8 bytes to 0 (for security purposes)
    – Since `bind` expects a pointer to `sockaddr` (not a pointer to `sockaddr_in`), we need to do 2 things
        1. Take the address of the `sockaddr_in` struct
        2. Cast to `struct sockaddr *`

- `address_len` is the length of the address we are passing
    – Set to `sizeof(struct sockaddr_in)`
- Return value is for error checking
    – Returns 0 on success
    – Returns -1 on failure
    – `bind` could fail if the port we picked is not available
- Overall setup

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

struct sockaddr_in addr;
addr.sin_family = AF_INET;
```

```c
addr.sin_port = htons(54321);
addr.sin_addr.s_addr = INADDR_ANY;
memset(&(addr.sin_zero), 0, 8);

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

The system call `listen` tells the machine to start looking for connections

- Prototype:

```c
int listen(int socket, int backlog);
```

- `socket` is the socket that we are setting up
- The return value is for error checking
- `backlog`
  - There may be a case where multiple users attempt to connect at almost the same time, bringing forth a queue for connection requests
  - `listen` sets up the data structure needed to store these partial connections
  - `backlog` is the maximum number of partial connections it can hold
  - *Not* the maximum number of connections it can hold

29.3 Setting Up a Connection

The `accept` System Call

- Prototype:

```c
int accept(int sockfd, struct sockaddr *address, socklen_t *addrlen)
```

- `sockfd` is the listening socket
- `address`
  - `accept` uses this parameter to communicate back to the caller, the address of the client
  - When accept returns, the `address` will point to a struct that holds the client's address information
  - Need to allocate memory for this struct before calling `accept`
  - We will pass in a pointer to `struct sockaddr_in`, and cast it to `struct sockaddr *`
    - The only field we need to set is the `sin_family`, which we set to `AF_INET`
- `accept` is a *blocking* system call – it waits until a connection is established
- The return value is -1 when `accept` fails
- On success, the return value is an integer representing a new socket which we will use to communicate with the client

- `addrlen` is the length of the address
  - We set the length to the size of our address, and pass in the *address* of this value
- Overall setup

```
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);
```

To initiate a connection over a socket to a server, use the `connect` system call

- Prototype:

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

- `sockfd` is the socket
- `address` is the address of the socket on the server to which we want to connect
  - Use type `struct sockaddr_in`
  - Set the field for `sin_family` to `AF_INET`
  - `memset` the field `sin_zero` to 0s
  - Set `sin_port` to the desired port, and convert to the network byte order with `htons`
  - `sin_addr` needs to refer to the IP address of the server
    - Use the system call `getaddrinfo` to look up the internet address of a machine based on its name
    - Prototype for `getaddrinfo`:

    ```
    int getaddrinfo(char *host, char *service, struct addrinfo
        *hints, struct addrinfo **result)
    ```

    - `service` and `hints` can be set to `NULL`
    - `host` is the name of the host machine, e.g. `"teach.cs.toronto.edu"`
    - `result` is the address of a pointer to a linked list of structs
      - There might be more than 1 address that satisfies the request for address information
      - Each element in the list is information about one of those valid addresses
      - We need to declare a pointer of type `struct addrinfo *` and pass its address to this field
      - The system call allocates memory for the linked list on the heap, and provides a function we call to free that memory when we are finished
      - To free that memory, use `freeaddrinfo`
    - We could look at the first address information struct from `result`
      - The first address is directly pointed to by `result`
      - It has a field `ai_addr`, whose type is `sockaddr`, that holds the information we need
      - We can cast it to type `struct sockaddr_in *`
      - From that `sockaddr_in`, we look at the `sin_addr` field, and assign that to the `sin_addr` field of the struct we are setting up for our `connect` call

- We cast the resulting `struct sockaddr_in *` to `struct sockaddr *`

- `addrlen` is the length of `address`
  - Set it to `sizeof(struct sockaddr_in)`
- Returns 0 on success, and -1 on failure
- Overall setup

```c
int soc = socket(AF_INET, SOCK_STREAM, 0);

struct sockaddr_in server;
server.sin_family = AF_INET;
memset(&server.sin_zero, 0, 8);
server.sin_port = htons(54321);

struct addrinfo *result;
int getaddrinfo("teach.cs.toronto.edu", NULL, NULL, &result);
server.sin_addr = ((struct sockaddr_in *) result->ai_addr)->sin_addr;
freeaddrinfo(result);

int return_code = connect(soc, (struct sockaddr *) &server, sizeof(struct
    sockaddr_in));
```

29.4 Socket Communication

Server side:

- Create a socket on which the server listens for connections

```c
listen_soc = socket(...);
```

- Bind that socket to a particular port and the address of the machine

```c
bind(listen_soc, ...);
```

- Tell the socket to start listening for partial connections

```c
listen(listen_soc, ...);
```

- Call `accept`, which blocks, returning only if there is an error or when a connection is made

```c
int client_socket = accept(listen_soc, ...);

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

  - When `accept` returns, it returns the descriptor for a new socket
- The listening socket is still listening – we could call `accept` on it again
  - To handle multiple clients simultaneously, use either `fork` or `select` system calls

- We can use the socket descriptor just like a file descriptor
  - Can write to the client, e.g.

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

  - `\r` and `\n` are each considered 1 character
  - `\r\n` is the *network newline*
  - Can also read from the client
- Close the socket when we are done

  ```
  close(listen_soc);
  ```

Client side:

- Create a socket to connect to the server

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

- Allocate memory to hold the values we will read

  ```
  char buf[10];
  ```

- Read from the server

  ```
  read(soc, buf, 7);
  buf[7] = '\0';
  ```

- Can also write to the server
- Close the socket when we are done

  ```
  close(soc);
  ```

Reading over the Internet

- There is no guarantee that all content can be read by a single `read` call
- Need to utilize the return value of `read` to determine how many bytes are read, and call `read` again if necessary
- The return value of `read` can also determine whether the other end of the socket is closed

30 Shell Programming

The shell is a big loop that performs the following in order

- Prints a prompt
- Reads a command
- Parses the command
- Executes the command

Varieties of Shell Programming Languages

- sh
  - Versions include Version 7 shell, ksh, ash, bash, dash, etc.
  - These are implementations of the basic sh programming language, plus additional features implemented by the author
- csh
  - Varieties include csh and tcsh

When the shell parses the command line, it performs various *command-line substitutions*

- E.g. filename wildcards are substituted with the matching list of file names by the shell before executing the command
  - E.g. `cat *.c` is substituted with `cat a.c b.c`
- The `echo` command can be used to view the subtituted command-line arguments
- To execute the commands written in a file, use `sh`
- Command-line substitutions is also used for variables
  - Example of declaring a variable:

    ```
    $ i=3
    ```

    * No space between the operators
  - When we write dollar signs in front of the variable name, then the shell substitutes it with the value of the variable, e.g.

    ```
    $ echo $i
    3
    ```

To do arithmetic in shell, use `expr`

- E.g.

  ```
  $ expr 2 + 2
  4
  ```

- To perform arithmetic on variables, use backquote `` ` ``
  - What is inside backquote is interpreted as a command and is executed

- If output is captured, the output is substituted into the command line – minus a trailing newline character at the end
- E.g.

```
$ i=`expr 4 + 1`
$ echo $i
5
$ i=`expr $i + 1`
$ echo $i
6
```

To read input, use `read`

- `read` is followed by the variable names in which the input is stored
- Each input token goes into its corresponding variable
- If there are more input tokens than variables, then all the rest goes into the last variable
- E.g.

```
$ read x y
foo bar baz
$ echo $x
foo
$ echo $y
bar baz
```

- If there are more variables than tokens, then the subsequent variables are set to empty strings

`PATH` is a special variable to the shell

- It is a colon-separated list of directories
- When a command is executed, the shell searches the PATH variable for the command

To obtain the exit status of the last command executed, use `$?`

- Can use `echo` to print this value
- 0 is a success exit status
- Anything else is a failure exit status

If Statements

- Structure

```
if condition
then
    command
else
    command
fi
```

- If the condition succeeds (i.e. has exit status 0), then the then command is executed
- Otherwise, the else command is executed

- To have multiple conditions, use `elif`, e.g.

```
if condition1
then
    command1
elif condition2
then
    command2
else
    command3
fi
```

- If we don't want to do anything for one of the branches, use `:`
  - Analogous to `pass` in Python

The `test` command can be used to compare values

- E.g.

```
$ test 2 -lt 3
$ echo $?
0
$ test 3 -lt 2
$ echo $?
1
```

- Instead of numbers, we could use variables by putting a dollar sign in front of it
- Numeric comparison operators for `test`:
  - `-eq`: equal
  - `-ne`: not equal
  - `-gt`: greater than
  - `-ge`: greater than or equal
  - `-lt`: less than
  - `-le`: less than or equal
- These numeric comparison operators are adopted from the Fortran programming language
- String comparison operators for `test`:
  - `=`: equal
  - `!=`: not equal
- Motivation of having two sets of comparison operators:
  - `test 03 = 3` is false
  - `test 03 -eq 3` is true
- File testing operators for `test`:
  - `-f file`: file exists and is a plain file
  - `-d file`: file exists and is a directory
  - `-s file`: file exists and is of nonzero size

- Etc.

While Loops

- Structure

```
while condition
do
    command
done
```

- `test` is useful for the condition
- Useful for reading a file with `read`, which stops upon end of file
  - End of file can be signalled from the terminal using Ctrl-V
  - `/dev/null` is a special file that is always empty
  - When we write into `/dev/null`, the data is discarded

Boolean Constants

- `true`: exit with 0
- `false`: exit with 1

To combine boolean statements, use `&&` and `||` as in C

- They have the short-circuit behaviour like C

Quoting in sh

- In sh, `ls` and `"ls"` are the same since they are both strings
- We need to be able to suppress the interpretation of some characters
- To suppress the meaning of a single character, use \ followed by the character
- Double quotes suppress everything except for dollar sign, backquote, backslash, or the closing double quote
- Single quotes suppress everything except for the closing single quote
- Space is preserved in double quotes and single quotes
  - E.g. `cat 'a b'` looks for the file named “a space b”
- Example:

```
$ filename='a b'
$ cat "$filename"
```

  - For the `cat`, quotes are needed to avoid interpretation of the space, and we need double quotes to still interpret the dollar sign

For Loop

- Structure

```
for variable in list
do
    command
done
```

- More like the for loop in Python
- Can loop through any number of strings, e.g.

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

- The `seq` command is similar to `range` in Python, e.g.

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

- `seq` can be used in a for loop, e.g.

```
sum=0
for i in `seq 1 100`
do
    sum=`expr $sum + $i`
done
```

- Can use a variation of for loop that loops through all command line arguments

```
for i
do
    echo an argument is $i
done
```

Case Statement

- Structure

```
case expression in
    pattern1)
        command1
        ;;
    pattern2)
        command2
        ;;
    pattern3)
        command3
        ;;
    *)
        command4
        ;;
```

```
        esac
```

- Matches `expression` to each `pattern`
- Each case is terminated with `;;`
- Cases are tested in order
- Default case: `*)`
- The patterns can be regular expressions

File Descriptors

- When a program starts, it has three files open
  - 0: standard input
  - 1: standard output
  - 2: standard error
- Standard input and output are both the terminal
- Input and output can be redirected
  - `< file`: redirect standard input to file
  - `> file`: redirect standard output to file
  - `>> file`: append standard output to file (instead of overwrite)
  - `<< file`: take input from file
    - Called "here text"
    - What's inside the here text are interepreted as if it is in double quotes
    - To have single quotes behaviour, use `<<\`
- Standard error bypasses the pipe/redirection and prints to the terminal
- We can redirect file descriptor 2 using `2> file`
- To redirect one file descriptor to another, use `>&2` where the left side of `>&` is the file descriptor to redirect and the right side is the file descriptor to redirect to
  - This particular example redirects standard output to standard error

Special characters inside a file

- `$1`, `$2` are the 1st and 2nd command-line arguments, respectively
- `$#` is the number of command-line arguments
- `$*` expands to all command-line arguments
- `$@` is same as `$*` outside of double quotes; inside double quotes, the double quotes are considered to cease to operate in between arguments while still operating within a given argument
- When we want to pass all command line arguments to another program, use `$@` inside double quotes

`shift` command moves all of the command-line arguments down one place, i.e. `$3` becomes `$2`, etc., `$1` is discarded

For a variable, we could put braces around the variable name, e.g. `${var}`

- This makes it possible to have another argument that is immediately after the variable (without spaces)

Comments in sh is `#`