SENG 265 Cheatsheet

lshatzel

October 2022

Contents

1 Best Practices:

1.1 Development Models:

- Waterfall: Requirements Analysis, System Design, Implementation, Testing, Deployment, Maintenence.
- Spiral: Determine Objectives, Identify Risks, Development and Test, Plan and Next Iteration
- Agile (best): Define Requirements, UI Design, Development, QA (Quality Assurance), UAT (User Acceptance Testing), Client Feedback, Release?, (If no, incorporate Client Requirements).

1.2 Code Locality:

- Low coupling high cohesion
  - Minimize dependencies between modules
  - Maximize related functionality witihin module (example, XML parsion in tomcat.)
- Why? Crucial information for information/data for understanding module is self contained.
- One of reasons global vars are discouraged.
- Bad: low cohesion high coupling (coupling across multiple modules (known as cross cutting)) i.e. logging in tomcat

2 The UNIX Filesystem:

- Kernel $->$ is responsible for creating this abstraction of the filesystem
- maps physical storage into logical storage
- arranged in a tree like hierarchy
- usr stands for: UNIX System Resources
- The UNIX model of interaction has the user interfacing with the shell which interfaces with the kernel which interfaces with the hardware itself.

2.1 Code:

```
.. #Parent directory in hierarchy
. #current directory (itself)
~ #Used to denote a home directory
#Equivalencies
cd /home/user = cd ~user
cd = cd ~
pwd #displays current working directory

#Examples
cd /home #brings the user all the way to root then home
ls #lists the directory
zastre keyboardcat #output
ls keyboardcat
hi-rez.mp4 tinder-stuff.txt #output
#relative pathnames
cd /home
open keyboardcat/hi-rez.mp4
open ./keyboardcat/hi-rez.mp4
open ./keyboardcat/../keyboardcat/hi-rez.mp4 #this one is weird
#but it does the same as the others.
#we start at current directory, go down one,
#then comeback up one then down one again then open hi-rez
```

2.2 File Attributes

Every file has a set of attributes

- username (file owner)
- group name (file sharing)
- file size (bytes)
- creation time, modified time
- file type (file, directory, device, link)
- permissions

```
ls -l unix.tex test
-rwxr-xr-x 1 joe users 200 Dec 29 14:39 test
#output format: permissions | group name | user name | file size (bytes) |
#creation time | file size | permissions
```

Permission Types:

- Users: owner of file/directory
- Group: Users can be given shared access to a file.
- Other: User who is not the owner and not in the same group as a sharing file.

```
user ('u'): [-rwx------]
group ('g'): [----rwx---]
other ('o'): [-------rwx]
```

files

- read (r): allow file to be read.
- write (w): allows files to be modified (edit, delete)
- execute (x): tells unix file is executable
- - owner group other have no permissions

directories

- read (r): allows directory content to be read (listed)
- write (w): allows directory contents to be modified (create, delete)
- execute (x): allows users to navigate into that directory
- - owner group other have no permissions

```
chmod o+rx
```

```
chmod u=rwx, g=rx, o=x #sets different permissions seperately
chmod u+rx, g-rx, o+x #sets pemissions seperately also
```

UNIX IS CASE SENSITIVE

```
chown #change owner
chgrp #change group
less, more #displays one file at a time
touch #update timestamp or create new file
echo $SHELL #tells us the shell that we are in
cmd [options] [arguments] #standard bash command syntax
opt #option with one letter
optname #full length option
arguments #file or c

#simple bash commands:
type #tells if cmd is built in, alias or executable
whatis #small one line desc of cmd
cat #copies files to stdout
date #displays current date and time
wc #word count output: lines | words | characters
clear #clears the terminal
```

3 Input and Output Streams:

- standard in (stdin 0)
- standard out (stdout 2)
- standard error (stderr 2)

3.1 Stream Redirection

```
< #redirect standard in
cmd < file #redirects the file into a command
> #redirect standard out (overwrites)
>> #appends standard out to current file
2> #redirect stderr
2>> #append stderr
cmd >& file #stdout and stderr at the same time
cmd 1> out 2>err.log #redirect error and stdout to two diff files
```

3.2 Script Checking Unique Server Users:

```
ps aux > temp1.txt #process status (ps)
awk '{ print $1 }' temp1.txt > temp2.txt #matches the pattern with awk
sort temp2.txt > temp3.txt  #sorts in order
uniq temp3.txt > temp4.txt #removes non unique entries
wc -l < temp4.txt > temp5.txt #counts number of lines
cat temp5.txt #outputs final file
```

Pipes vs No Pipes:

```
#with pipes
ps aux | awk '{ print $1 }' | sort | uniq | wc -l
```

3.3 Command Sequencing:

```
cmd1; cmd2; cmd3
#note that > has a higher precedence than ;
(date; who; pwd) > logfile.txt
#not the same as
date; who; pwd > logfile.txt
#will only put pwd into logfile
```

3.4 File Expansion:

```
* #substitutes for any number of characters
? #substitutes for a single character
[abc] #substitutes for a set of characters
[!abc] #substitutes for anything other than these characters
'' #strong quotes all characters are preserved
"" #weak quotes -> some characters unwrapped (\£,`)
`` #backquotes substitutes result of eval as cmd
echo `date`
vim \* #backslash escaping -> allows us to reference special chars
history #gives history
!! #repeats last command
!n #repeats command number n
!-n #repeats command typed n ago
!foo #last command started with foo
```

3.5 Shell Variable:

- Some shell vars built in
- Some are user defined

```
env #displays environment variable
set #displays values of environment and shell
somevar="value" #stores a variable
unset somevar #deletes a variable
echo $somevar #use the £ prefix to use shell var
PATH #helps shell finds commands you want to execute
#sequence of directories
#extend path
find #zastre favorite
find <directory> -name "*.c" -print
```

4 Version Control

What we want to avoid: having one person write to a file then having someone else overwrite that information. Some approaches:

4.1 Locking:

- Locks the file while someone is writing to it
- Other people wait until lock is finished, then can make their own
- Downside: If someone is one vacation or sick they cant resolve the lock

4.2 Copy Modify Merge:

- Allows users to work in parallel
- Writes are prevented until both repos are up to date
- Makes sure changes dont overlap
- git makes this happen through SHA-1 hashing to track versions

4.3 Git:

```
#git workflow
git clone <repo> #only done once
git pull
git add <files to be added>
git commit -m #message on what commit changes
git push #pushes to remote repo
#long form
git push origin master
git pull <=> git fetch + git merge #in succession
#long form
git pull origin master
git log #list of snapshots
git status #reports file relationships
#the above two are like a flashlight into the git repo
#when first setting up git
git config --global user.name "Firstname Lastname"
git config --global user.email "NETLINKID@uvic.ca"
#to check that this worked
git config --list

#To clone remote repo (exact)
git clone ssh://NETLINKID@git.seng.uvic.ca/seng265/NETLINKID
```

- Git doesnt track directories (only files) this means git wont see an empty directory only the files within it (if we want to add a directory it needs to have files in it)
- The best wat to confirm that our changes have been pushed is to reclone our remote repo. This will let us know exactly what is in remote.
- **We can have as many local repos as we'd like**
- Typically default name for remote repo is origin

- Default name for remote branch is master
- Never store executables / generated files in the git repo
- When we merge and get a conflict we have to pull and manually decide what to keep and what to delete.

5 C

5.1 Big C Program

```c
/*Function to do wordcount*/
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <stdlib>

#define MAX_LINE_LEN 256
int main (int argc, **argv){
    FILE *infile;
    char line[MAX_LINE_LEN];

    int num_chars = 0;
    int num_lines = 0;
    int num_words = 0;

    char *c;

    if (argc < 2){
        fprintf(stderr, "usage: \%s filename \n", argv[0]);
        exit(1);
    }

    infile = fopen(argv[1], "r");
    if(infile == NULL){
        fprintf(stderr, "\%s cannot open \%s", argv[0], argv[1]);
        exit(1);
    }

    while(fgets(line, MAX_LINE_LEN-1, infile) != NULL){
        num_lines += 1;
        num_chars += strlen(line);
        if(strncmp(line, "", MAX_LINE_LEN) != 0){
            num_words++;
        }
        for(c = line; *c; c++){
            if(isspace(*c)){
                num_words++; //very sketchy way to count words by spaces,
                //this is probably not a good idea
                /*Specifically this function iterates through a pointer to an array
                of chars checking if each char
                in the array is a space and calling that a word*/
            }
        }
    }
    fclose(infile);
    printf("\%s: \%d \%d \%d \n", argv[1]), num_lines, num_words, numchars);
}
```

5.2 General C Notes:

C Features

- Easy to run executables on anything.
- No native array bound checking.

- No null pointer checking.
- No checks for uninitialized variables.

5.3 How the C compilation process actually works

```
gcc -c hello.c //compiles into object file
gcc hello.c -o hello // links object file into executable
gcc hello.o -o hello -lm //links the math library in too
//more terse version of file compilation
//(does everything in one step)
gcc -Wall -std=c11 hello.c -o hello
```

1. Editor $->$ outputs the sources
2. compiler (preprocessor, parser, code generator, optimizer)
3. Linker (takes unresolved symbols and makes sure its able to run)
4. executables

```
/*Character constants in 8-bit ASCII*/
char ch = 'A';
char bell = '\a';
//note that chars are also just 8-bit (bytes)
//we can treat it as an unsigned byte
char c = 65; /*This is the same as A*/

/*Integer Literals*/
int a  = 10;
int b = 0x1CE;
int c = 0777; //octal

unsigned int x = 0xfffU;
long int y = 2L;
```

5.4 Char* and String Literals

STRING LITERALS ARE DANGEROUS

```
char *s = "unable to open file\n";
/*THIS STRING IS READ ONLY*/
/*Just an address to a variable holding a static string table*/
//an example of bad string literals

int main(){
    char s[50] = "abcdefghijklmnopqrstuvwxyz";
    char *t = "zxywvutsrqponmlkjihgfedcba";

    printf("Message s is: \%s \n", s);
    s[0] = ' ';
    s[1] = ' ';
    printf("Modified message s is: \%s \n", s);

    printf("Message t is: \%s \n", t);
    t[0] = ' ';
    t[1] = ' ';
    printf("Modified message t is: \%s \n", t);

    /*  ouput:
    message s is: 'abcdefghijklmonpqrstuvwxyz'

    modified message s is: '  cdefghijklmonpqrstuvwxyz'
    message t is ...
    Bus error 10
    results in erro
```

```
    */
}

//similar problem
//also note that \0 ends lines

char person_A[50] = "Anakin Skywalker";
char *person_B = "Darth Vader";

printf("\%s, \%s", person_A, person_B);

person_A[1] = '\0';
printf("\%s", person_A);

person_B[1] = '\0';
printf("\%s", person_B);

/*output:
Anakin Skywalker, Darth Vader
A
Seg fault
*/
```

5.5 Arrays

- Group of data elements of same type
- static or dynamic allocation -¿ static cannot grow at runtime, dynamic can.
- arrays may be multidimensional
- no bound checking
- array size must be known at compile time
- There is no array size attribute, the programmer must manage / know the array size on their own

```
//standard array declaration
<storage class> <type> <identifier>[<size>]
//size must be known at compile time
double f[3] = {0.1, 2.2, -100.51};
int freq[10] = {10, 12};
/*freq[0] = 20
freq[1] = 12
the rest of the numbers in the array are initialized to zero
*/
```

5.6 Control Flow

- C does not have a boolean type by itself (note: we can include ¡stdbool.h¿ though)
- There are certain boolean operators that are available: ¡, ¿, ¡=, ¿=, ==, !=, &&, ——, !
- **Any expression that evaluates to zero is false otherwise it is true**
- if statement note: any value at runtime can be in an if statement (just remember that any value that isnt zero is interpreted as true).

```
/*Example of how not to use equality*/
int a = 20;
if(a = 5){ //this will always evaluate to true
//because in C all statements have a value
//in this case it would be 5
    //Statement;
}

//the extreme approach? Would be to write
```

```
if(5 == a){
}

/*Switch Statements*/
switch(intexpr){
    case int_literal_1:
        statement;
        break;
    case int_literal_2:
        statement;
        break;
    case int_literal_3:
        stateement;
        break;
   .
   . /*More cases*/
   .
    default:
        statement;
        break;
}
//note on syntax:
/*
intexpr: integer expression
intlit: integer literal
(it must be computable at compile time
(i.e. we must know the cases we are looking for before hand)
if(intexpr == intlit) the statement executes
break continues execution after the switch statments closing brace
*/

//AN EXAMPLE
#include<ctype.h>
#define TRUE 1
#define FALSE 0

int isvowel(int ch){
    int res;
    switch(toupper(ch)){
        case 'A':
        case 'E':
        case 'I':
        case 'O':
        case 'U':
            res = TRUE;
            break;
        default:
            res = FALSE;
        }
    }
    return res;
```

Control Flow Cont.

- Iteration within a while loop is top tested
- keywords: continue and break are significant
- continue: starts the next loop iteration by checking the while conditional
- break: exit the loop immediately, resume at the first instruction after the loop body
- do while loop: iteration bottom-tested
- continue and break still have significance

```c
//While loop example
char buf[50];
int pos = 0;
if (fgets(buf, 50, stdin) == NULL){
    /*report an error and exit*/
}
while(buf[pos] != '\0'){
    if(isvowel(buf[pos])){
        putchar(toupper(buf[pos]));
    } else {
        putchar(buf[pos]);
    }
    pos += 1;
}

//Do while loop example
do {
    ch = getchar();
    if(ch == BLANK)
        cnt += 1; //counts blank spaces in a line
} while(ch != '\n');


//some weird possible for loops
int i, sum;
for(int i = 1, sum = 0; i <= 20; ++i, sum += i); //no body
printf("Sum of first 20 integers is: \%d\n", sum);

/*Examining some infinite loops*/
while(1){
    /*Do something*/
}

for(;;){
    /*Do something*/
}
```

The loops above represent the possible infinite loops that we can use in C.

- while: this is always evaluated on every iteration (there is an extra little bit of friction)
- for: this does nothing so there is no check. we dont have a check on each iteration so its more pro
- Note: usually we use an infinite loop when the test has to go in the middle.

5.7 Functions:

- A program is made up of one or more functions, one of which is main()
- Program execution always begins with main (means a C program needs a main to run)
- When the program sees a function name for the first time the function is invoked
- Function invocation: program control passes to function - Function is executed - control is passed back to the calling function.

```c
int main(int argc, char *argv[]){
    printf("Hello World\n");
    return 0;
}

double fmax(double x, double y){
    if(x > y){
        return x;
    }else{
        return y;
    }
}
```

- Functions are tasks we want to be able to repeat many times
- They always have a return type (needs to return void if they dont return anything)
- C wont check that every path returns a value (i.e. in if statements)
- **Must have open and close curly braces**
- Usually main has parameters (num arguments in command and text (argc, char *argv[]))

5.8 Parameter Passing

- C is call-by-value
- Means we pass a copy of the variable rather than the address of where that variable is in memory

```
/*formal parameters: m, n*/
int maxint(int m, int n){
    if(m > n){
        return m;
    } else {
        return n;
    }
}

void some_function() {
    int a = 5;
    int b = 10;
    int c;

    /*Actual parameters a,b*/
    c = maxint(a,b);
    printf("Maximum of %d and %d is %d", a, b, c);
}

int power2(double f){
    if(f > sqrt(DBL_MAX)){
        return 0;
        /*error was detected*/
    }else{
        reutrn (int) (f * f);
    }
}

void some_other_function(){
    double g = 4.0;
    int h = power2(g);
    printf("%f %d \n", g, h);
}
```

- Because we see variables we think that it is being passed in
- not true we get a copy of the variables being passed in, and we cant modify a and b from within the function
- This is a one way transfer of information from parametrs to the function
- The value is being copied into the formal parameters
- C exposes the fact that all vars are data, all data is in memory (shared memory)
- Every memory location has some address associated to it

REMEMBER THESE POINTS:

1. All variables refere to data
2. All data resides in memory
3. Every memory location has an address
4. C exposes these details for us to use in our programs
5. Some languages hide this from us (Java, C#, Python)

5.9 Pointer Variables

- Defn: holds the address of a memory location storing a value of some data type
- named variable (vars that are declared in program or local scope) int a, float c
- To obtain an address: use & symbol
- To use an address use * symbol
- Function calls are stored on stack
- Heap is available to function at all times
- Type name initial value
- BSS is really old opcodes used on mainframes
- Program is stored as text instructions
- textbfHEAP IS AVAILABLE TO ALL FUNCTIONS AT ALL TIMES
- How to read pointers
- Read right to left
- eg (int *a) a is a variable that holds an address to an int
- just remember that * has a different meaning in variable declration than in usage

```
/*When comparing the following code fragments*/
int x = 1;
int y = x;
x = 2;
printf("y is %d\n", y); /*y is 1*/

int x = 1;
int *y = &x;
x = 2;
printf("*y is %d\n", *y); /*y is now 2*/
/*y is a variable which stores the address to an int, in this case the int is x*/
/*this is the same as get the address of x and read whats at that address*/

/*just to hammer the point home*/
char *st[10];
/*this is declaring an array of pointers
(we can think of the array as storing an address to a type int*/
```

Using pointers allows us to have a function that modifies values outside of scope because we can only have one return value (without using structs)

- C doesnt check pointer validity
- The address could be a region holding garbage
- Programmers responsibility to ensure that pointers contain a valid memory address
- Also allows us to do type violations (taking an address to an int and using it as a char)

5.10 Arrays

- Arrays are pointers
- Just aggregate data types where each data element has the same type
- All elements in arrays occupy contiguous memory locations
- To get the address of any element we can use & (reference operator)
- Asking, at this address what is in the array (&grades[4])
- Usually the first array location is the most important

```c
char buffer[100];
char *cursor;

/*These lines are the same*/
cursor = &buffer[0];
cursor = buffer;

int X[4];
int *p = &X[0];
p++;
/*we can increment the pointer because C knows that it is an address to an int*/
/*it knows to go down by 4 bytes (or whatever the size of an int is in the current architechture)*/
X[n] = *(p + n);
```

- Call-by-value CAUTION
- Arrays still use call by value but what is passed is the address to the first element of the array
- C doesn't copy the value of each element

5.11 Strings

- Strings are not a datatype in C
- We are wokring directly with the memory system to make sure we have a large enough area for an array of chars
- Concatenation does not work
- Memory is not automatically allocated
- Boundaries between stings are not enforced
- strncat does work through
- If we try to copy a string (array of chars) to a space that is too small for it it will clobber the next block of memory (spill over)
- If string is in binary (text section of code) then it is immutable (unchangeable)
- Start of strings are addresses to chars
- End of string is null \0
- Size of the string doesnt necessarily the start of the array in which it is stored
- The start of a string is ALWAYS an address to a character

```c
#include <string.h>
/*String functions in C*/
strncpy()
strcmp()
strncat()
strtok()

/*Examples:*/
char words[20];
char *pw;
strncpy(words, "The quick brown fox", 20);
pw = &words[0];
pw += 4;

printf("%s \n %s \n", words, pw);

/*output*/
/*the quick brown fox*/
/*quick broun fox*/

strncpy(words, "homer simpson", 20);
printf("%s \n%s\n", words, pw);
/*output*/
```

```
/*homer simpson*/
/*r simpson*/

/*what is actually in array*/
/*homer simpson \0n fox \0*/
```

- Must have enough room in char array for all characters plus null
- Note we CANNOT modify char * (strings that are defined in the binary (text))

```
char *t = "abcdefghijk"
t[0]  = ' ';

/*results in bus error*/

/*WE CAN MANIPULATE POINTERS IN MANY WAYS*/
char *cp = buffer /*equiv*/ cp = &buffer[0]
cp + n /*equiv*/ & buffer[n]
*(cp + n) /*equiv*/ buffer[n]
cp++ /*equiv*/ cp = cp + 1
*cp++ /*equiv*/ *cp, cp++
```

5.12 Tokenization

- Tokenization allows us to break char arrays into shorter "tokens" in order to work with them
- i.e. an input line consists of individual words
- Words are seperated by whitespaces (space characters, tabs)
- We can tokenize the line by the whitespaces to get single strings of words
- strtok works by replacing the white spaces with a null terminator and and returning the pointer to that token
- It will modify the line passed into it

```
/*Examples:*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX_WORD_LEN 20
#define MAX_WORDS 100
#define MAX_LINE_LEN 100
#define MAX_LINES 10

int num_words = 0;
int num_lines = 0;
char lines[MAX_LINES][MAX_LINE_LEN];
char words[MAX_WORDS][MAX_WORD_LEN];

void dump_words(void);
void tokenize_line(char *);

int main(int argc, char *argv[]){
    int i;
    /*checks that the only thing isnt just the command itself*/
    if(argc == 1){
        exit(0);
    }

    for(i = 0; i < argc - 1; i++){
        strncpy(lines[i], argv[i+1], MAX_LINE_LEN);;
        tokenize_line(lines[i]);
    }
    dump_words();
```

```
    printf("first line: %s\n", lines[0]);
    exit(0);
}

void dump_words(){
    int i = 0;
    for(i = 0; i < num_words; i++){
        printf("%5d : %s\n", i, words[i]);
    }
    return;
}

void tokenize_line(char *input_line){
    char *t;
    t = strtok(input_line, " ");
    while(t && num_words < MAX_WORDS){
        strncpy(words[num_words], MAX_WORD_LEN);
        num_words++;
        t = strtok(NULL, " ");
    }
    /*What would the following print?*/
    /*printf(%s\n", input_line)*/
    /*It would print only the first word of the line as the rest of the words now have null terminators after
}
```

- We pass NULL into strktok the second time to tell it to tokenize further but from where you last left off.
- When strtok finds the final new line character in the string the tokenization will terminate and there will be no more strings to tokenize. strtok returns null and then the while loop stops.

5.13 Arrayness of Char Arrays:

```
 /*How to declare a 2D string array*/
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define MAX_WORDS 5
#define MAX_WORD_LEN 7

char w[MAX_WORDS][MAX_WORD_LEN];

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

    for(i = 0; i < MAX_WORDS; i++){
        w[i][0] = '\0';
    }

    /*This is the same as strncpy(w[i],"", 1)*/
    /*Putting a null terminator at the beginning of each entry of the array to clear it*/
}
```

- When we dont have enough space in the char array, and put a word that is longer than the space to hold it the null terminator can get overwritten
- When we print this will cause us to output a long string that is not what we wanted to output.
- This is because a 2D array of chars is really just a 1D array in memory with each word coming before the next. All seperated by null terminators.
- Char arrays are in Row-Major order meaning they go row by row.
- strcpy is EVIL because it will overwrite other values in a 2D array if we try to copy something that is longer than the space allocated for a word.

5.14 File IO

```
/*Files are acceses as streams using FILE objects*/
FILE *data = fopen("input.txt", "r");

/*data is a variable which stores the address to a file (not a file pointer)*/

/*File streams*/
/*Built in variables (these are streams)*/
FILE *stdin
FILE *stdout
FILE *stderr
/*Automatically opened by the O/S*/

/*open modes*/
fopen("pathname", "r"); /*reading*/
fopen("pathname", "w"); /*writing will erase a file if the file exists and ovewrite it*/
fopen("pathname", "a"); /*appending*/
/*We can add a + after any of rwa to open a file for reading and writing*/

 /*whenever we perform some file output the OS keeps enough in buffers to lay data where it should be*/
/*flushes anything intermediate in buffers*/
fclose(FILE *stream) /*close and flush any buffers. Tries to speed up file operation*/

fgetc() /*read single characters from an open file "r" mode*/

/*read at most n-1 chars from stream and copy to buf*/
/*returns NULL if EOS is encountered*/
fgets([char* (to write to)],[int size], [FILE *stream])

/*In the assignment we used stdin*/
/*if fgets returns null this is how we know we've gotten to the end of the input*/
fgets(words[num_words], MAX_WORD_LEN - 1, stdin);

/*ouputs single char*/
/*char stored in an integer*/
/*passing char as param also works*/
fputc(int c, FILE *stream) /*output a single character to the open file (openend in w mode)*/

/*rather than reading in as chars and tokenizing we can use this*/
fscanf(char *format, [..]) /*read data with a bit of a structure*/
/*reads formatted data*/
/*format specifiers encoded in format*/

/*important, string formatting int formatting*/
/*know how to write decimals*/
/*returns the number of chars printed*/
/*takes a variable number of args*/
printf(char *format, [..])

/*like printf but output goes to opened stream*/
fprintf(FILE *stream, char *format, [..])

/*outputs a single char to stream (already opened)*/
int fputc(int c, FILE *stream)



/*charbychar.c
This will echo the contents of th eifle specified as the first arg
char by char
*/

#include <stdio.h>
```

```c
#include <stdlib.h>
int main(int argc, char *argv[]){
    int ch, int num_char;
    if(argc < 2){
        /*fprintf allows us to print to a stream*/
        fprintf(stderr, "You must provide a filename\n");
        exit(1);
    }

    FILE *data_fp = fopen(argv[1], "r");

    if(data_fp == NULL){
        fprintf(stderr, "unable to open %s\n", argv[1]);
        exit(1);
    }

    num_char = 0;

    while((ch = fgetc(data_fp)) != EOF){
        num_char++;
        printf("%c", ch);
    }
    fclose(data_fp);

    fprintf(stdout, "Number of characters: %d\n", num_char);
    return 0;
}

/*linebyline.c
Echo the contents of the files specified as the first argument line by line
*/

#include <stdio.h>
#include <stdlib.h>
#define BUFLEN 100

int main(int argc, char *argv[]){
    char buffer[BUFLEN];
    int num_lines;

    if(argc < 2){
        fprintf(stderr, "You must specify a file\n");
        exit(1);
    }

   FILE *data_fp = fopen(argv[1], "r");

   if(data_fp == NULL){
    fprintf(stderr, "Unable to open file %s\n", argv[1]);
    exit(1);
   }

   num_lines = 0;
   while(fgets(buffer, sizeof(char) * BUFLEN, data_fp)){
    num_lines++;
    printf("%d: %s", num_lines, buffer);
   }
   fclose(data_fp);
   return 0;
}

/*Syntax for a format placeholder*/
%[paramter][flags][width][.precision][length]type
```

- Files are accessed as streams of FILE objects
- Streams, FILE
- A stream is a long sequence of characters which comes to an end with some sort of end of stream.
- Streams stdin, stdout, stderr are automatically opened when the program starts
- char by char oriented data
- fopen always takes a path
- fopen specifies a mode
- File addresses (file pointers) can store the value of 0, location 0 is a special case.
- if fopen is involved and something goes wrong (path doesnt correspond to something we have, dont have read permissions) 0 means it failed.
- if the file cannot be open is we are returned NULL or 0.
-

5.15 Structures

- Some languages refer to these as records
- Heterogeneous aggregate type
  - declaring a new aggregate type which is not all of the same
- very important: becomes new data type
- NO METHODS OR FUNCTIONS CAN BE ASSOCIATED WITH SUCH DATATYPE

```
 /*This type is called struct day of year*/
struct day_of_year{
 int month;
 int day;
 int year;
 float rating;
};

struct{
 int x;
 int y;
} id;
/*id is a variable(anonymous struct)*/

struct point{
 int x;
 int y;
};
/*struct point is a new type*/

 struct point{
     int x;
     int y;
 } x, y, z[10];

 /*struct point is a new type, x,y,z are variables*/

 typdef struct point{
     int x;
     int y;
 } Point;

 /*struct point is a new type Point is a synonym for that type*/
```

```c
/*to access members of a struct we use the member operator*/

struct day_of_year today;
today.day = 45;
today.month = 10;
today.year = 2014;
today.rating = -1.0;

/*we can also define arrays of structs*/
struct day_of_year calendar[365];
calendar[180].day = 27;
calendar[180].month = 9;
```

- **Important note**
- typedef struct is not defining a new type it is simply making a synonym for the struct

```c
/*Examples of using an array of structs*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MAX_NAME_LEN 20

struct body_stats_t{
    int code;
    char name[MAX_NAME_LEN];
    float weight, height;
}

int main(void){
    struct body_stats_t family[4];

    family[0].code = 10;
    family[0].weight = 220;
    family[0].height = 190;
    strncpy(family[0].name, "Michael", MAX_NAME_LEN - 1);

    print_stats(family[0]);
}

void print_stats(body_stats p){
    printf("Member with code % is named %s\n", p.code, p.name);
}

/*We can use the arrow operator with structs*/

typdef body_stats_t *p;
p = &family[0];
/*or we can say p = family*/
p->code = 10;
p->height = 191;

/*this is equivalent to*/
(*p).code = 123;
/*deference to whatever address the variable holds and set that to the value we want*/
```

5.16 Function Prototypes

- There is a problem with print stats in the above code snippet
- C is a one pass compiler it will only go through the whole file in one pass from top to bottom.
- Checks to make sure that the parameter types we are passing always matches function signatures.

- In this case the first time that the compiler sees print_stats it is when we are calling the function rather than when it is declared.
- this is where function prototypes come in (also called function declarations)
- includes optional storage class, function, function return type, function name, function parameters

```
[<storage class>] <return type> name <parameters>;
/*parameter types are necessary but names are optional, names are recommended (improves code readability)*/
/*prototype looks like a function without the body*/

int isvowel(int ch);
extern double fmax(double x, double y);
static void error_message(char *m);
```

Function prototypes don't require parameter names

5.17 Function Pointers

```
/*function call is performed to whatever function is stored at the address in fp*/
foo = (*fp)(x,y);

/*qsort uses function pointers*/
qsort(array, int num_elements, int element_size, comparison function);

int main(){
    int i;
    int numbers[MAX_NUMBERS] = {3, 14, 15, 9, 26, 58. 53};

    qsort(numbers, MAX_NUMBERS, sizeof(int), compare_int);

}

int compare_int(const void *a, const void *b){
    int ia = *(int *)a;
    int ib = *(int *)b;

    return ia-ib;
}
```

- Will not be asked to write a function pointer on an exam.
- but we can parameterize functions to other functions
- a function is not a variable but we can assign the address of functions into pointers and pass them to functions and return them from functions.

5.18 Typedef

- Using typedef allows us to create synonyms for already defined types

```
typdef datatype synonym;

typedef unsigned long int ulong;
typdef unsigned char byte;

ulong x, y, z[10];
```

5.19 Programming Idioms

- Programming idiom
- Recurring construction in programming languages
- imply terseness
- i.e. non string example: infinite for loop
- Terseness can imply speed

- Idiom is for(;;)

Example of an idiomatic strlen function

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

#define MAX_STRING_LEN 100

int stringlength_4(char a_string[]){
    char *c;
    int len;

    for(len = 0, c = a_string; *c; c++, len++){
        retunr len;
    }
}

int main(int argc, char *argv[]){
    char buffer[MAX_STRING_LEN];

    if (argc == 1){
        exit(0);
    }

    strncpy(buffer, argv[1], MAX_STRING_LEN);
    printf("%d\n", stringlength_4(buffer));

    exit(0);
}

//non idiomatic strlen function:

int stringlength_1(char a_string[]){ //this is the same as writing char *a_string
    int len = 0;

    while (a_string[len] != '\0'){ //each char is compared against null, in while loop
        len = len + 1;
    }
    return len;
}

int main(int argc, char *argv[]){
    char buffer[MAX_STRING_LEN];

    if(argc == 1){
        exit(0);
    }

    strncpy(buffer, argv[1], MAX_STRING_LEN);
    printf("%d\n", stringlength_1(buffer));

    exit(0);
}

//most idiomatic version

int stringlength_5(char a_string[]){
    char *c;

    for(c = a_string; *c; c++);
```

```
    return c - a_string;
}

//this works because a char is 1 byte
//therefore taking the difference between pointers will give the correct length
```

- For the non idiomatic version:
  - Usually for loops are used for strings instead of while loops
  - while statements are better for things that have a changing or dynamic termination property that we have to keep checking for.
  - in other words: for loops are for lists where the elements do not change
  - and while loops are meant for lists with more complex conditions
- For the idiomatic version:
  - depends on the meaning of true and false in C
  - we can initialize multiple variables in the loop header
  - in each iteration we check that *c is not 0, which would return a false value.
  - dereferencing the char value
  - combines pointer arithmetic

5.20 Scope

- Four main types of scope in C:
- Program Scope:
- File Scope
- Function Scope
- Block Scope
- Each falls into their own category of scope:
- Global:
  - File
  - Program
- Local:
  - Function
  - Block

5.20.1 Program Scope

- Variable exists for the lifetime of the program and can be accessed from ANY file within the program

```
/*
 * file 1
 */

 int ticks = 1;
 void tick_tock(){
    ticks += 1;
 }

/*
 * file 2
 */

extern int ticks;

int read_clock(){
    return ticks * TICKS_PER_SECOND;
}
```

- Since we didnt put anything in front of ticks we are able to access it from anywhere inside of the program.
- but we need to tell the compiler that we want to access ticks from another file so we use the extern keyword

5.20.2 File Scope

```
/*
 * file 3
 */

 static long long int boot_time = 0;

 void at_boot(void){
    boot_time = get_clock();
 }

 int main(void){
    printf("%i\n", boot_time);
 }

/*
 * file 4
 */

 //THE LINE BELOW WILL FAIL

 extern long long int boot_time = 0;
```

- In this case the variable is hidden from outside of the file
- So it is hidden from the other files in the program which is why we cant use it.

5.20.3 Function Scope

```
/*
 * file 5
 */

 void function_f(int x){
    ... = x + ...;
 }

/*
 * file 6
 */

 //The variable declaration within the function
 //will cause a compiler error

 void function_g(int x){
    int x;  /*Cant do this*/
 }

 //cant redeclare because x is already in the scope
 //this is called shadowing


/*
 * file 7
 */

 int sum = 0;

 void function_h(int x){
```

```
    int sum = init_sum();
}

//on the other hand we can shadow global variables
```

- Scope of function arguments is the same as the scope of the variables defined at the outermost scope of a function, we are not allowed to shadow arguments of a function
- We can however shadow global variables as seen in file 7

5.20.4 Block Scope

```
/*
 * file 7
 */

int sum;

void function_y(int X[], int n){
    {
        //start of new nested scope
        int j;
        for(j = 0, sum = 0; j < n; j += 1){
            sum += X[j];
        }
    } //end of nested scope
}
```

- Variable is visible from declaration to the end of the block
- begin block by open and close curly braces
- if we were to put static infront of a variable within block scope it will preserve the local value between function calls therefore the function will start off at the value from the previous function call.

5.21 Abstract Data Types (ADTs)

- We've used basic data types so far
- but to have modularization we need abstract data types
- ADT was precursor to objects
- formal defn: set of operatoins which access a collection of stored data
- known as encapsulation in OOP languages
- Dont care how the structures are implemented just care about them doing what we want them to do.
- We can use ADTs and function prototypes to simulate modules
- more of a convinience thing
- A C compiler doesn't force file seperation
- Allows us to implement the one declaration one definition rule
- This is a rule that a program should have only one definition for one thing.
- For module mod there are two files
- interface named mod.h
  - Function prototypes
  - public type defs
  - constants
  - and global vars
  - Note: also called header files
- Implementation module named mod.c:
  - Actual function implementations

5.21.1 Example: Bitstring

- bitstring.h: contains declarations for data structures and operations required to support bitstring manipulations, must be visible to user
- bitstring.c: implementation of bitstring operations

5.21.2 bitstring.h

```
#ifndef BITSTRING_H
#define BITSTRING_H

typedef unsigned int Uint;
typedef enum _bool {false = 0, true = 1} bool;

#define BITSPERBYTE 8
#define ALLOCSIZE (sizeof(Uint) * BITSPERBYTE)
#define BYTESPERAUNIT (sizeof(Uint))

 extern void clear_bits(Uint[], Uint);
 extern void set_bits(Uint[], Uint);
 extern void reset-bit(Uint[], Uint);
 extern void test-bit(Uint[], Uint);

 #endif
```

5.21.3 bitstring.c

```
 #include "bitstring.h"

 void clear_bits(Uint bstr[], Uint naunits){
     Uint i;
     for(i = 0; i < nauints; i++{
         bstr[i] = 0;
     }
 }

 void set_bit(Uint bstr[], Uint bit){
     Uint b_index = (bit - 1) / ALLOCSIZE;
     Uint b_offset = (bit - 1) % ALLOCSIZE;

     bstr[b_index] |= (1 << b_offset); //bitwise or and rightshift
 }

 void reset_bit(Uint bstr[], Uint bit){
     Uint b_index = (bit - 1) / ALLOCSIZE;
     Uint b_offset = (bit - 1) % ALLOCSIZE;

     bstr[b_index] &= (~(1 << b_offset));
 }

 //determines the state of a bit in a bitstring
 bool test_bit(Uint bstr[], Uint bit){
     Uint b_index = (bit - 1) / ALLOCSIZE;
     Uint b_offset = (bit - 1) % ALLOCSIZE;

     return(( bstr[b_index] & (1 << b_offset)) ? true : false);
     //1 returns true otherwise false
 }
```

5.21.4 Bitstring: Usage

```
 #include "bitstring.h"

 #define NUINTS 4
```

```c
int main(int argc, char *argv[]){
    Uint set[NUINTS];

    clear_bits(set, NUINTS);
    set_bit(set, 8);
    set_bit(set, 12);

    if(test_bit(set, 12) == true){
        reset_bit(set, 12);
    }

    return 0;
}
```

5.22 Dynamic Memory

- memory divided into five segments:
  - uninitialized data: data
  - initialized data: Bss
  - heap
  - stack
  - code: text
- stack: as the function call depth increases so does the stack (which increases downward toward heap)
- heap: as memory is requested the heap grows upwards (malloc)
- Each function call creates a stack frame which allocates memory for variables to be used by function
- if stack and heap grow too much all memory available will be used and an out of memory condition will occur
- when they collide it will generate a signal but doesnt necissarily terminate program or cause a segfault.
- we want dynamic memory so that we can work with input whose size might not be known at run time
- heap: dynamic allocation, when size is now known at runtime
- stack: size is known at compile time.
- working with heap is harder because we have to manually deal with memory
- heap memory addresses are sometimes called anonymous variables
- they don't have names

```c
#include<stdlib.h> //contains function prototypes for malloc and related functions.

malloc() //allocates dynamic memory

//malloc takes a single parameter representing number of bytes of heap memory to be allocated
//returns a memory address to beginning of block of memory
//if allocation fails malloc() returns null

sizeof() //operator
//returns number of bytes to store an instance of that type
//we use size of in conjunction with malloc to determine block size
//always check value returned by malloc

int *a = malloc(sizeof(int));
if(a == NULL){
    /*Error*/
}

struct datetype *dt = malloc (sizeof(struct Datetype));
```

```
if(dt == NULL){
    //error
}

char *buffer = malloc(sizeof(char) * 100));
if(buffer == NULL){
    //error
}

//malloc and casting
//function prototype for malloc()

extern void *malloc(size_t n);
typedef unsigned int size_t;

//function returns a generic pointer
//have to typecast returned pointer

//denoted by:
<sometype> *

//example:
double *f = (double *) malloc(sizeof(double));
char *buf = (char *) malloc(100);
//takes void * and makes sure its treated right

//sometimes there is a warning if malloc isnt typecast

//unless explicitly deallocated heap memory stays allocated

//malloc family:

calloc() //function allocates and initializes block of heap memory
realloc() //adjust heap structure to change size of a block/chunk
valloc() //force allocation (pretty much obsolete)

free() //returns heap memory to pool (when no longer needed)
//input type is a pointer to allocated block of memory

void very_polite_function(int n){
    int *array = (int *) malloc (sizeof(int) *n));
    /*code using array*/

    free(array);
}
```

- A memory leak happens when memory is continuously allocated but not freed
- Can be hard to seperate memory leak from a bug, as can cause random behavior
- systems with automatic garbage collection almost never have memory leaks
- Redundant memory is returned for heap to reuse

5.22.1 Dynamic Arrays:

- All of C programs using arrays up to now have been static in size
- Never need to manage this memory
- When having dynamic arrays though we need to manage the memory ourselves.

```
typedef struct Nameval Nameval;

struct Nameval{
    char *name;
```

```
     int value;
};

struct Nvtab{
     int nval;
     intn max;
     Nameval *nameval;

} nvtab;

enum {NVINT = 1, NVGROW = 2};

//creating a new nameval

Nameval *new_nameval(char *name, int value){
     Nameval *temp;

     temp = (Nameval *)malloc(sizeof(Nameval));
     if(temp == NULL){
         fprintf(stderr, "Error mallocing a Nameval");
         exit(1);
     }
     //temp->name === (*temp).name
    temp->name = (char *)malloc((strlen(name) + 1) * sizeof(char);

    if(temp->name == NULL){
     fprintf(stderr, "Error mallocing memory for string");
     exit(1);
    }
    strncpy(temp->name, name, strlen(name)+1);

    temp->value = value;
    return temp;
}

//addname

int addname(Naveval newname){
     Nameval *nvp;

     if(nvtab.nameval == NULL){
         nvtab.nameval = (Nameval *)malloc(NVINT * sizeof(Nameval));
         if(nvtab.nameval == NULL){
             return -1;
         }
     } else if (nvtab >= nvtab.max){
         nvp = (Nameval *)realloc(nvtab.nameval, NVGROW *nvtab.max)* sizeof(Nameval));
         if(nvp == NULL){
             return -1;
         }
         nvtab.max = NVGROW * nvtab.max;
         nvtab.nameval = nvp;
     }

     nvtab.nameval[nvtab.nval] = newname;
     return nvtab.nval++;
}

//delete a name
//this can be tricky because we have to decide what to do with the gap
//if element order doesnt matter we can just swap and shrink
//if it does then we have to shift
int delname(char *name){
int i;
```

```
    for(i = 0; i < nvtab.nval; i++){
        if(strcmp(nvrab.nameval[i].name, name) == 0){
            memmove(nvtab.nameval + i, nvtab.nameval + i + 1,
                (nvtab.nval - (i+1) * sizeof(Nameval));
                nvtab.nval--;
                return 1;
        }
    }
    return 0;
   }
```

- We have to write flexible code to take in many input lines
- We may not know the number of lines that are input into the program
- this means we have to use malloc, realloc, and free appropriately
- or we could just use getline()
- keeps scanning until it hits the end of file
- allocates memory on heap for us
- we always have to use free with getline though

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

int main(void){
    FILE *fp;
    char *line = NULL;
    size_t len = 0; //just an unsigned int
    ssize_t read; //ssize_t is when a func may return a size or negative, s = signed

    fp = fopen("/etc/motd", "r");
    if(fp == NULL){
        exit(1);
    }

    while((read = getline(&line, &len, fp)) != -1){
        printf("retrieved line of length %zu : \n", read);
        printf("%s", line);

    }
    if(line){
        free(line);
    }
    exit(0);
}
```

5.22.2 Wrapper Function: emalloc

We can make a wrapper function for malloc which checks for allocation errors automatically

```
    void *emalloc(size_t n){
        void *p;

        p = malloc(n);
        if(p == NULL){
            fprintf(stderr, "malloc of %u bytes failed", n);
            exit(1);
        }
        return p;
    }
```

5.22.3 Lists

- Arrays are not always a suitable choice
- lists can be rearranged by changing a few pointers. Cheaper than a block move like memmove.
- When items are inserted or deleted, other items are not moved in memory
- if we store addresses to list elements in other data structures, the list elements themeselves wont necissarily be invalidated by changes to the list.
- If the set of items to maintain changes frequently (esp if number of items is unpredictable).
- A list is a good way to store them

```
//definition for a node in the linked list

typedef struct Nameval Nameval;
struct Nameval{
    char *name;
    int value;
    Nameval *next; //in list
}

//constructing an item
Nameval *newitem(char *name, int value){
    Nameval *newp;

    newp = (Nameval *)emalloc(sizeof(Nameval));
    newp->name = name;
    newp->value = value;
    newp->next = NULL;
    return newp;
}

//add items to front
Nameval *addfront(Nameval *listp, Nameval *newp){
    newp->next = listp;
    return newp;
}

//add items to the end
Nameval *addend(Nameval *listp, Nameval *newp){
    Nameval *p;
    if(listp == NULL){
        return newp;
    }
    for(p = listp; p->next != NULL; p = p->next);
    p->next = newp;
    return listp;
}

//find an item
Nameval *lookup(Nameval *listp, char *name){
    for(; listp != NULL; listp = listp->next){
        if(strcmp(name, listp->name) == 0){
            return listp;
        }
    }
    return NULL;
}

//Note: many operations require list traversal
//we can make this easier with a general purpose function
//called apply
```

```
void apply(Nameval *listp, void (*fn)(Nameval*, void*), void *arg){
    for(; listp != NULL, listp = listp->next){
        (*fn)(listp, arg);
    }
}

//examples:

//print all elements in list
void printnv(Nameval *p, void *arg){
    char *fmt;
    fmt = (char *)arg;
    printf(fmt, p->name, p->value);
}

//count all elements in list
void inccounter(Nameval *p, void *arg){
    int *ip;
    ip = (int *)arg;
    (*ip)++;
}


//properly freeing a list

void freeall(Nameval *listp){
    Nameval *next;
    for(; listp != NULL; listp = next){
        next = listp->next;
        free(listp);
    }
}
```

5.22.4 Trees:

- Binary Search Tree
- straightforward to implement
- Node has value and left and right pointers
- Pointers lead to children nodes
- All children to left have values lower than the node
- all children to the right have values greater than the node

```
//struct for treenode
typedef struct Nameval Nameval;
struct Nameval{
    char *name;
    int value;
    Naveval *left;
    Nameval *right;
}

//tree insert function
//treep is the pointer to teh root of the tree
//newp is the node to insert
Nameval *insert(Nameval *treep, Nameval *newp){
    int cmp;

    if(treep == NULL){
        return newp;
    }
```

```
    cmp = strcmp(newp->name, treep->name);

    //recursively find
    if(cmp == 0){
        fprintf(stderr, "ignoring duplicate entry %s\n", newp->name);
    } else if(cmp < 0){
        treep->left = insert(treep->left, newp);
    }else{
        treep->right = insert(treep->right, newp);
    }

    return treep;
}


//lookup (find function)
Nameval *lookup(Nameval *treep, char *name){
    int cmp;

    if(treep == NULL){
        return NULL;
    }

    cmp = strcmp(name, treep->name);
    if(cmp == 0){
        return treep;
    }else if(cmp < 0){
        return lookup(treep->left, name);
    }else{
        return lookup(treep->right, name);
    }
}

void applyinorder(Nameval *treep, void (*fn)(Nameval*, void*), void *arg){
    if(treep == NULL){
        return;
    }
    applyinorder(treep->left, fn, arg);
    *(fn)(treep, arg);
    applyinorder(treep->right, fn, arg);
}

//this works for printnv from list implementation
```

- Routines dont have to be recursive, we can translate them into iterative ones for more efficiency.

5.23 Make

- A software build is a constructed executable version of the program.
- To build software more efficiently, we shouldn't have to reprocess every file in our program if a change was made to only a small part of the program.
- Make is a programming utility.
- make uses makefiles to describe dependencies.
- not reprocessing all files in a program can save compilation time
- but processing can also mean re-generating files, relinking objects, re-running tests/
- most code remains unchanged from compilation to compilation.
- good programming practice suggests we break programs into smaller modules.
- each module corresponds to a separate file.

5.23.1 Dependencies:

- Each generated file depends on others to be created.
- typically each created file depends on at least one input file.
- Drawn as a graph called a dependency graph.
- just points to which files depend upon which other files.

5.23.2 Makefiles:

```
#makefile example

SHELL=/usr/bin/bash

CC=gcc

newlinker: newlinker.o newsets.o
    $(CC) -o newlinker newlinked.o newsets.o

newlinker.o: newlinker.c newsets.h
    $(CC) -c -g -Wall -st=c99 newlinker.c

newsets.o: newsets.c newsets.c
    $(CC) -c -g -Wall -std=c99 newsets.c
clean:
    -rm newlinker.exe newlinker.o newsets.o

#generic rule structure

target: prerequisites
    recipe
```

- Make files contain:
- rules: which consist of 3 parts
  - The targets come before the colon; these are usually the name of the file generated by a program. (i.e. executables or object programs).
  - some targets have no prerequisites, i.e. clean, these are called phony targets.
  - The prerequisites come after the colon. These are the files that the target requires in order to be created (dependencies)
  - recipe is what is within the rule (indented by a tab. This is the part that make actually carries out. (command)
  -
- Variables:
  - Makes things easier to read
  - makes things easier to modify
  - defined on their own line

```
#example of a variable
OBJECTS=data.o main.o io.o

    project1: $(OBJECTS)$
        gcc $(OBJECTS)$ -o project1

    data.o: data.c data.h
        gcc -c data.c

    main.o: data.h io.h main.c
        gcc -c main.c

    io.o: io.h io.c
```

```
        gcc -c io.c

#example of implicit rules

default: single

single: single.o teams.o input.o

single.o: teams.h single.c $(includes)

clean:
    -rm -f *.o
```

5.23.3 Implicit Compilation:

- Certain ways of remaking target files are often used.
- Implicit Rules: tell make how to use customary techniques so that you dont have to specify them in detail when you want to use them
- i.e. C compilation usually takes a .c file and makes a .o file
- we can just write the target and dependencies of the rule and not specify a recipe, this will still work.

5.23.4 Automatic Variables

```
#specify the target of the rule
$@

#all the dependencies of the rule
$^

#the first dependency of the rule
$<

#all dependencies "newer" than the target
#?

#These go in the recipe section of the rule and simply allow us to avoid syntax
#errors

#example
#(OLD RULE)
newlinker: newlinker.o newsets.o
    $(CC) -o newlinker newliker.o newsets.o

#(NEW RULE USING VARIABLES)
newlinker: newlinker.o newsets.o
    $(CC) -o $@ $^
```

5.23.5 Additional Features

- Phony targets: like clean as mentioned above (have no dependencies but have a recipe)
- recursive makefiles: makefiles kept in subdirectories, makefile at the top controlling all of the makfiles
- include files: we can "include" variables in different make files

6 Python:

6.1 Why Python?

- Object oriented (but doesnt have to be)
- Less verbose than Java and cleaner than Perl.

- Built in datatypes (strings, lists)
- Machine Learning
- Powerful regex

6.2 Basic Datatypes:

```
x = 34 - 23 #A Comment
y = "Hello" # another comment
if z == 3.45 or y == "Hello": #booleans end in :
 x = x + 1 #addition
 y = y + " World!" #string contatenation
 w = x // 4 #integer division
 z = x / 4 #Floating point division

 print(x)
 print(format(y, ".2f")) #Two digits after decimal

 = #assignment
 == #comparison
 + - * / % #behave as expected for numbers
 #can also concatenate strings with +
 #can use % for string formatting (like printf in C)
 and or not #logical operators unline Java or C
 print() #basic printing function

 z = 5 // 2 #integer
 x = 3.456 #float
 y = 4 / 3 #float

 "abc" 'abc' #we can use single or double quotes for strings
 "matt's" #can use quotation marks within strings, dont need to match
 """ab'c""" #can use triple double quotes for multi line
 #strings or to contain both single and double quotes within (but only one)
```

- First assignment creates a variable
- Variables dont need to be declared
- python figures out types on its own
- whitespace is meaningful in python
- Esp indentation and newlines
- Newlines must end code
- no braces in python just indentation
- colon appears at start of new block (functions, classes)

6.3 Comments

```
#this is a comment
#but we can also include documentation strings
"""This tells us about the function or class we define. Good style"""
```

6.4 Assignment

```
a, b = 10,20
a,b = b,a

#output: a = 20, b = 10
#why? python reads from right to left
#always read assignment from right to left
```

- Binding a vairable in python means setting a name to hold a **REFERENCE** to some object.

- assignment creates references not copies
- Names in python do not have intrinsic types but objects do have types.
- Python determines the type of the reference automatically based on data assigned to it
- A name is created the first time it appears on the left of the assignment expression
- A reference is **deleted** via garbage collection after any names bound to it have passed out of scope.
- Upon accessing nonexistent names we will get a traceback error.
- python has reserved words which we cannot use for variable names.
- **All types are objects in python there is no notion of primitive types**

```
and, assert, break, class, continue, def, del, elif, else
except, exec, finally, for, from, global, if, import
in, is, lambda, not, or, pass,
print, raise, return, try, while
```

6.5 Sequence Types

1. Tuple
    - **Immutable**
    - Can be mixed types
2. Strings
    - **Immutable**
    - Conceptually like a tuple
3. List
    - **Mutable**
    - *ordered* sequence of items, mixed types

```
#all operations can be applied to all sequence types

#TUPLE
t = (23, 'abc', 4.56, (2,3), 'def')

#LIST
li = ["abc", 34, 4.34, 23]

#STRING
st = "Hello World"
st = 'Hello World'
st = """This is a multiline string"""

#We can access individual members using array notation
t[1] #output 'abc'

li[1] #output 34

st[1] #output e

#we can also positive and negative index

t[1] #output 'abc'

#negative lookup: count from right starting with -1
t[-3] #output 4.56

#return a COPY of a tuple

#doesnt include the last element
```

```
t[1:4] #('abc', 4.56, (2,3))

#negative indices can also be used when slicing
t[1:-1]
#output ('abc', 4.56, (2,3))

t[:2] #copy starting from beginning
t[2:] #copy starting from the first index and going to the end

li[:] copying a whole sequence
```

Will always return a copy not the original

6.6 Operators

```
#The in operator
#boolean to test whether a value is inside a container
t = [1, 2, 3, 4]
5 in t
#returns false

#can also use not
4 not in t

#returns false

#the + operator
#produces a NEW tuple list or string
#concatenation of whatever we used + with
(1, 2, 3) + (4, 5, 6)
#output: (1, 2, 3, 4, 5, 6)

"Hello" + " " + "World" + " "
#output: Hello World

#The * operator
#Produces a NEW tuple, list or string
#repeats original content

(1, 2, 3) * 3
#output (1, 2, 3, 1, 2, 3, 1, 2, 3)

#List only operators

li = [1, 11, 3, 4, 5]

li.append('a')

#output [1, 11, 3, 4, 5, 'a']

li.insert(2, 'i')

#output [1, 11, 'i', 3, 4,  5, 'a']

li.extend([9, 8, 7])

#output: [1, 2, ... , 9, 8, 7]

#difference between extend and append!!
#extend takes lists as an argument, whereas append takes singletons

#whereas
li.append([10, 11, 12])
#output: [1, 2, ... [10, 11, 12]]
```

- Tuples Vs. Lists
- Lists are slower at runtime but more flexible than tuples.
- Lists can be modified and we can perform operations on them
- Tuples are immutable and have fewer features
- we can type cast between lists and tuples by using list(tu) tuple(li)

6.7 Reference Semantics

- Assignment manipulates references
- x = y does **NOT** make copy of the object y's reference
- x = y makes x REFERENCE the object y references
- i.e. what happens when we type x = 3
  - An int 3 is created and stored in mem (right side is evaluated first)
  - A name x is created
  - Reference to the memory location storing 3 is assigned to x
  - x now refers to the integer 3
- datatypes such as int float string and tuple are immutable
- This means we cannot change the value that x stores we can only change what x is storing
- x = x + 1 makes a new object of whatever was in x and adds one to it
- old data is garbage collected

6.8 Assignment

```
x = 3
y = x
y = 4

#in this case the value of x will remain 3
#the value of y will change to 4
#there are new objects created
#and y is assigned to that object
```

- For mutable types assignment works differently
- if we make a change through a reference to a mutable object then then the object itself will change and this will be reflected in the reference.

6.9 Dictionaries

- Dictonaries are mappings between keys and values.
- Keys HAVE TO BE any immutable type values can be any type
- A single dictionary can store values of different types, define, modify, view, and lookup and delete the key-value pairs.
- Dictionary gotcha if the key doesnt exist or reference value it throws an error

```
d = {'user':'bozo', 'pswd':1234}

d['user']

#output: bozo

d['bozo']

#error is thrown
#we can change key value mappings on the fly
```

```
#constructing a key value pair with a new key doesnt cause an error

d['id'] = 45

#note: dictionaries work by hashing

#remove the key and value associated with user

del d['user']

#removes all key value pairs from the dictionary
d.clear()


#useful methods
d.keys() #returns a list of keys (iterable)
d.values() #returns a list of values

d.items() #returns a list of item tuples
```

6.10 Functions

```
#function definition always begins with def

def get_final_answer(filename):
    """docstring"""
    line1
    line2
    return total_counter

#function call
myfun(3, 4)

#default values
def myfun(b, c=3, d="hello"):
    return b + c
#we can override the default values or leave them out
#argument order can also be changed as long as we specify the argument
myfun(c=43, b=1, a=2)

#main function
def main():
    #stuff goes here

if __name__ == "__main__":
    main()
#tells python to call the main function at the start of the program

#lambda notation

def applier(q,x):
    return q(x)

applier(lambda z: z * 4, 7)

#output: 28

#gives applier an unnamed function that takes input
#multiplies it by 4
```

- Functions are call by assignment
- Mutable datatypes: behave like call by reference

- Immutable datatypes: behave like call by value
- ALL functions in python have a return value
- functions without a return actually return None (like null)
- there is no function overloading in python
- functions can be used as any other data type (we can return them, assign them, put them in lists)
- lambda notation
  - define functions without names
  - useful when passing a short function as an argument to another function
  - lambda expressions can only take single expression single parameter functions

6.11 Logical Expressions

:

```
#Comparison
x == y #x and y have the same value
x is y #x and y are variables which refer to the exact same object

 #boolean logical expressions
 a and b
 a or b
 not a
 #use parenthesis to break down complex expressions
 x and y and z
 #if all are true value of Z gets returned
 #otherwise the value of the first false subexpression is returned

 x or y or z
 #if all are false the value of z gets returned
 #otherwise the first true sub-expression gets returned
 #and/or use short circuit evaluation so no further expressions are eval

 x = true_value if condition else false_value

 #uses short circuit evaluation
 #true value is eval and returned if true
 #false value is eval and returned if false

#suggested use
 x = (true_value if condition else false_value)
```

- True and False are constants in python
- Equivalents to true and false (non-zero nums, non empty obj), (zero, none empty container or object) respectively
- and and or DONT return true or false
- they return value of one of the sub expressions which may be true or false

6.12 Control Flow

:

- if
- while
- assert

```
 # if statements
if x == 3:
    print("X equals 3.")
elif x == 2:
    print("X equals 2.")
 else:
    print("X equals something else")
 print("This is outside the if")

 #while loops
 x = 3
 while x < 10:
    x = x + 1
    print("still in the loop")
 print("Outside the loop")

 break #leaves while loop
 continue #stops the current iteration and starts the next

 assert #checks if a condition is true during execution
 #if it is false program stops and gives a line number
 #a reminder to ourselves about some value
 #should never be shipped in user code
```

6.13 For loops and List Comprehensions

```
for <item> in <collection>:
 <statement>
#item can be more than just a var name
#AS LONG AS SEQUENCE ITEMS HAVE SAME TYPE
for (x,y) in [('a',1),('b',2)...]
 print(x)
#called pattern matching notation

range() #returns list of numbers (non inclusive)
for x in range(5):
 print(x)
 #prints list of numbers from 0 to 4

 enumerate() #allows index numbers and items

 for i, item in enumerate(["moo", "woof", "meow", "quack"]):
    print(i, item)
#list comprehensions

#generates new list by applying function to every member of og list

 li = [3, 6, 2, 7]

 [elem*2 for elem in li]

 #output: [6,12,4,14]

 #general form:
 [expression for name in list]
 #expression acts on name
 #calculates result of name and expression and returns a new list

 #NEEDS TO ALL BE OF SAME TYPE
 #expressions can also contain user defined functions
 def subtract(a, b):
    return a - b
 oplist = [(6,3), (1,7), (5,5)]
```

```
[subtract(y,x) for (x, y) in oplist]

#output: [-3, 6, 0]

#filtered list comprehension

[expr for name in list if filter]

#nested list comprehensions
li = [3,2, 4, 1]
[elem*2 for elem in
    [item + 1 for item in li]]
#output: [8, 6, 10, 4]
```

6.14 String Conversions and Operations

```
#join turns a list of strings into one string
<separator_string>.join(<some_list>)

":".join("abc", "def", "ghi")
#ouput: abc:def:ghi

#split splits one string into a list of strings
<some_string>.split(<separator_string>)
"abc:def:ghi".split(":")

#output: ["abc", "def", "ghi"]

#note how the syntax is reversed between join and split

#string conversions
"Hello" + str(2)

#output: Hello 2

#cant depend on python knowing how to concatenate a string
#have to cast whatever we want to concatenate

#ANY data type can be converted to a string

#upper and lower

"hello".upper()
#output: HELLO

"HELLO".lower()
#output: hello

#string format operator %

#similar to sprintf in C
x = "abc"
y = 34

"%s xyz %d" % (x,y)
#output: abc xyz 34

#print outputs to stdout
print("%s xyz %d" % ("abc", 34))

#automatically adds new line to the end of strings

#to print a list of strings
print("abc", "xyz")
```

```
#output: abc xyz
#python automatically includes a space between

#if we want to print without new line and just a space

print("abc", end = " ")

#complex formatting
print('Course unit: {};  Number {}'.format('SENG', '265'))

#output: Course Unit: SENG; Number 265

print('Course unit: {1}; Number {0}'.format('265', 'SENG'))

#output: Course unit: SENG; Number: 265

#center:
{:0^4}
#left align:
{:0>4}
#right align:
{:0<4}
```

6.15 File Processing and Error Handling:

```
#different ways to write a word count function


def mywc():
    num_chars = 0
    num_words = 0
    num_lines = 0

    for line in sys.stdin:
        num_lines = num_lines + 1
        num_chars = num_chars + len(line)
        line = line.strip()
        words = line.split()
        num_words = num_words + len(words)

    print(num_lines, num_words, num_chars)

def mywc():
    num_chars = 0
    num_words = 0
    num_lines = 0

    #note the change here
    #if file names provided the loop will iterate through line in file
    #otherwise it will loop through stdin
    for line in fileinput.input():
        num_lines = num_lines + 1
        num_chars = num_chars + len(line)
        line = line.strip()
        words = line.split()
        num_words = num_words + len(words)

    print(num_lines, num_words, num_chars)

#bad while loop example
#its better to use a for loop
def mywc():
    num_chars = 0
    num_words = 0
```

```
        num_lines = 0

        lines = sys.stdin.readlines()

        while(lines):
            a_line = lines[0]
            num_lines = num_lines + 1
            num_chars = num_chars + len(a_line)
            line = a_line.strip()
            words = a_line.split()
            num_words = num_words + len(words)
            lines = lines[1:]

        print(num_lines, num_words, num_chars)
```

- Play with error handling in python by accessing files without permissions

```
def main():
    import sys
    fileptr = open('filename', 'r')
    if fileptr == None:
        print("Something bad happened")
        sys.exit(1)
    somestring = fileptr.read() #read first line
    for line in fileptr: #continue reading
        print(line)
    fileptr.close()

    #when we open a file we can specify how the file should be opened

    file = open("somefile", "r") #read strings
    file = open("somefile", "rb") #read binary arrays
    file = open("somefile", "w") #write strings
    file = open("somefile", "wb") #write binary string

    #more on working with files
    file = open("fubar", "w"(
    file.write("hamberders\n") #note: file.write returns number of chars written

    file.close()

    file = open("fubar.bin", "wb")
    file.write("hamberders")
    #throws an error
    #cant write strings to binary output

    #hexdump -C fubar
    #00000000 68 61 6d 62 65 72 64 65 72 73 0a

    #what we can do

    file = open("oops.txt", "w")
    file.write("\xab\xba\xfe\xed")

    file.close()
    #output
    #00000000 c2 ab c2 ba c3 be c3 ad

    file = open("okay.txt", encoding=("genau.txt", encoding="utf-8", mode="w")
    file.write("\xab\xba\xfe\xed Gebürstag")
    file.close()

    #ouput (after hexdump)
```

```
#00000000 c2 ab c2 ba c3 be c3 ad 20 547 65 62 c3 bc 72 74
```

- Python supports both ASCII (latin-1) encodings and UTF-8, sometimes we need to specify the encoding we want
- do not depend on platform default

Exceptions and Handlers

```
while True:
    try: x = int(input("Number, please! "))
    print("The number was: ", x)
except ValueError:
    print("Oops! That was not a valid number.")
    print("Try again.")
    print()

try:
    f = open("gift.txt", "r")
    #do file processing things
    f.close()
except FileNotFoundError:
    print("Cannot open gift.txt")
except PermissionError:
    print("Whatcha trying to do here with gift.txt eh?")
    #do failure-due-to perms stuff


def loud_kaboom():
    x = 1/0

def fireworks_factory():
    raise ZeroDivisionError("Gasoline near bone dry christmas trees!")

def playing_with_fire():
    try:
        loud_kaboom()
    except ZeroDivisionError as exc:
        print("Handling run-time error: ", exc)
    try:
        fireworks_factory()
    except ZeroDivsionError:
        print("Gotta stop this from happening")
#try and except is the same as raise and throw in java
```

- File IO has lots of possibilities in terms of things going wrong whenever we open a file, it is important that we want to make sure we handle the possible exceptions.
- try and except are passed all the way to the caller and handler.

6.16 Scope Rules

- Scope depends on location of variable definition, and the global modifier
- **LEGB**
  - First search LOCAL defn
  - not found: Enclosing defn
  - not found: Global defn
  - not found: Built in defn
  - not found: built in
  - not found: despair

```
#we cant use global vars unless we explicitly intend to
    X = 22
    def func():
        X = 33
    func()
    print(X) #unchanged out: 22


"""--------------------"""
#falls through to global scope
X = 99
def func(Y):
    Z = X + Y
    return Z

func(1) #result: 100


"""--------------------"""

X = 88

def func():
    #telling function we explicitly want to refer to global X
    global X
    X = 99
func()
print(X) #prints 99

"""--------------------"""
#x as global will introduce x into the global scope
y,z = 1, 2

def all_global():
    global x
    x = y + z

#nested functions
X = 99

def f1():
    X = 88
    def f2():
        print(X) #LEGB rule falls through to E, where it gets X
    f2()
f1() #output: 88
```

- Global scope is actually module scope
- no real notion of private in python
- always good to minimize the use of global variables
- Sometimes it makes sense to have global variables though.

6.17 Importing and Modules

Different import types:

```
    import somefile #if we want to use class or function we must append somefile.

    from  somefile import * #has downside of overwriting our own classes and functions with same name

    from somefile import className #has the downside of overwriting functions and vars with the same name.
```

Useful/Commonly used Modules:

- sys (lots of handy stuff)
- os (os specific code and directory processing)
- math (mathematical code)
- Random (random number code)

How to add our own modules:

- Directories that Python looks for files in is sys.path
- To add a new directory of our own, we append it to the list via:
- sys.path.append('/my/new/path')

6.18 Object Oriented Programming

- **Everything** in python is an object.
- Classes:
  - A class is a special data type which defines how to build certain objects.
  - Instances are objects that are created which follow the definition inside a given class
  - Python doesnt use seperate class interfaces (unlike Java)
  - We define a method by including a function within the class.
  - Each method must contain the first argument self
  - usually has dunter init dunter as the constructor.

```
class Student:
"""A class representing a student"""
 def __init__(self, n, a):
     self.full_name = n
     self.age = a
 def get_age(self):
     return self.age
```

  - line 2: class docstring (describes the class)
  - constructor for the object
  - anything we assign self will become an instance variable, dont have to declare it elsewhere.
  - Note how all methods have self as a parameter.

6.18.1 Instantiating Objects

- No new keyword in python, simply assignment
- Just use the class name with () parenthesis
- The init function works as the constructor for the object
- b = Student("Bob", 21)
- init can take any number of arguments.

6.18.2 self

- Self is the very first argument of every class method
- In general self refers to the instance whose method was called.
- eg. in init self refers to the object currently being created.
- similar to this in Java
- self is never included when calling the method, only when defining the method.

6.18.3 Garbage Collection

Python has automatic garbage collection no need to free or delete manually.

6.19 Access to Attributes and Methods

```
class Student:
    """A class representing a student"""
    def __init__(self, n, a):
        self.full_name = n
        self.age = a
    def get_age(self):
        return self.age

   #traditionally we would access as so:
   f = Student("Bob Smith", 23)
   f.full_name  #get an attribute
   f.get_age() #getter method

    #sometimes in large code bases we are only given the name of an
    #attribute or method at runtime
    #we can use getattr instead:

    getattr(object_instance, string)

    #string is a string which contains the name of an attribute or method of a class
    #the method itself returns a reference to that attribute or method

    hasattr(object_instance, string)
    #does the same but returns a boolean
```

6.19.1 Attributes

- Two main kinds of attributes:
- Data Attributes:
  - Each variable is owned by a particular instance of a class
  - Each instance has its own value for it
  - These are the most common kind of attribute
- Class attributes:
  - Owned by the class as a whole
  - all instances of the class share the same value for it
  - called ”static” variables in some languages
  - Good for: class wide constants or counters

6.19.2 Data Attributes

Data attributes are created and initailized by init (assignment creates the attribute)

```
   class Teacher:
    "A class representing teachers"
    def __init__(self,n):
        self.full_name = n
    def print_name(self):
        print(self.full_name)
```

6.19.3 Class Attributes

- All instances share an copy of a class attribute
- When any instance changes it changes for all of the instances
- notice how class attributes are accessed with self.dunterclassdunter.name

```
class Sample:
    x = 23
    def increment(self):
        self.__class__.x += 1
```

6.20 Inheritance

6.20.1 Subclasses

- Subclasses can extend the definition of superclasses
- We put the name of the superclass into the declaration for the subclass
- Method redefinition: just include a new definition of the method.

```
class AI_Student(Student):

#executing a parent class method
ParentClass.methodName(self, a, b, c)


#class extending student class
class Student:
    def __init__(self, n, a):
        self.full_name = n
        self.age = a
    def get_age(self):
        return self.age
class AI_Student(Student):
    def__init__(self, n, a, s): #redfines init
        Student.__init__(self,n,a) #old constructor
        self.section_num = s
    def get_age(self): #redefines get age method
        print("Age: " + str(self.age))
```

6.21 Special Built-In Methods

```
#All built in members have double underscores around their names
__init__
__doc__ #stores the documentation string for the class
__repr__ #how the object is represented as a string
__cmp__ #how objects are compared
__len__ #how len(obj) works
__copy__ #how to copy a class
dir(x) #returns list of all methods and attributes defined for object x
```

6.22 Private Data and Methods:

- Any attribute or method with two leading underscores is meant to be treated as private. It is not meant to be accessed outside of that class.

6.23 Python Data Model

6.23.1 Card Deck Collection

```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit']

class FrenchDeck:
    #NOTE THE LACK OF CONSTRUCTOR

    def __len__(self):
        return 52
```

```
    def __getitem__(self, position):
        if position >= 52:
            raise IndexError
        suit_index = position // 13
        rank_index = (position % 13) + 2

        if (suit_index == 0): suit = "spades"
        elif (suit_index == 1): suit = "hearts"
        elif (suit_index == 2): suit = "diamonds"
        else:                   suit = "clubs"

        if rank_index >= 2 and rank_index <= 10:
            rank = rank_index
        elif rank_index == 11: rank = 'J'
        elif rank_index == 12: rank = 'Q'
        elif rank_index == 13: rank = 'K'
        elif rank_index == 14: rank = 'A'

        return Card(str(rank), str(suit))
```

- The use of len and get item allows us to index into the french deck in this code
- we can also construct a certain card Card('4', 'clubs')
- len and getitem also allow for the use of

6.24 Decorators:

- Decorators perform actions around functions, which are then called the decorated functions.
- Tricky to figure out how a function is called within decorator
- And how to make sure that the decorator receives its needed arguments.

```
#if we want to use decorators to time functions
import time

#this is the decorator name
def duration(func):

    #nested function but it will not be called by name
    #*args are the positional arguments passed to stopwatch
    def stopwatch(*args):
        t0 = time.perf_counter()

        #calls the func that has been generated passing args from stopwatch
        result = func(*args)
        elapsed = time.perf_counter() - t0
        print('[%0.8fs]' % elapsed))

        #returns the function (closure)
        return result
    return stopwatch

    #This is the decorator
    #it is the same as calling obtain_name = duration(obtain_name())
    @duration
    def obtain_name(greeting):
        n = input(greeting + " What is your name? ")
        return n

    @duration
    def obtain_age():
        while (True):
            try:
```

```
            age = input("What is your age (in years)? ")
            age = int(age)
            return age
        except ValueError:
            print("Sorry, we need a whole number. Try again.")

def main():
    n = obtain_name("Welcome to Bluefish Insurance Brokers.")
    a = obtain_age()
    print(n, "is", a, "years of age. Woot, I say.")

main()
```

- The decorated functions are called through the decorator now.
- The function itself gets passed as func to duration
- The function arguments are passed to stopwatch
- finally the function is called on its arguments in stopwatch and its result is stored in result
- result is returned and then stopwatch is done

6.25 Generators

- All sequences in python are iterable
- They implement the __iter__ function
- this is either as a method, or through the python interpreter
- python will call iter(x) on an object and the interpreter will look for the iter method if it has been implemented, otherwise it can fall back on getitem, otherwise it will throw a type error
- **Difference** between *iterable* and *iterators*
- iterable is a peoperty/characteristic possesed by some object
- iterator is a method/code that implements two methods:
- we can use a generator to make an object that has a next operation rather than using __iter__
  - __next__: returns the next available item
  - __iter__: returns self, when the object is used in a for loop
- Deep Dive into yield
- yield produces a generator object
- when code calls next on a generator object, python resumes the generator from the statement after the previous yield and returns the value of the next yield with next()
- The yield statement yields a value
- Generators are also iterators
- Because of the use of the next() function the generator resumes from the statement following the yield.

7 Regex

- In a nutshell regex (regular expressions) are a language-independent approach to expressing patterns.
- Type of regular grammar. (Mathematical object with alphabet rules and so on.)
- grep stands for global regular expression print

7.1 Python Regex

```
#Metasymbols

. #match any character except \n
a* #zero or more repetitions of a
a+ #one or more repetitions of a
a? #zero or one repetition of a
a{5} #exactly 5 repetitions of a
a{3,7} #3 to 7 repetitions of a
[abc] #matches any one of a, b, c (any char in set)
[^abc] #matches any char not in set
a|b #matches a or b
(...) #groups matches
\ #escapes any meta symbol

#special pattern elements
\d #any decimal digit character
\w #any alphanumeric character [A-Za-z0-9_]
\s #any whitespace character
\b #empty string at word boundary
#(specifically it can match before a word char \w,
#after a word char and before the next \w \W \w after the last word char
^ #match 0 characters at the start of the string
$ #match 0 chars at the end of the string
\D #match any non digit character
\W #match any non alphanumeric character
\S #match any non whitespace character
\B #empty string, not a at a word boundary
\number #matches text of a group with number 'number'
?: #specifies non-matching group
#(i.e. we wont be able to rereference a match after this with a number)

#simple regex example

 import re

 text1 = 'Hello spam... World'
 text2 = 'Hello...spam'
 matchobj = re.match('Hello.*World', text2)
 print(matchobj)

 #result None

#why? Looking for Hello with some character followed by 0 to many characters then World
#text2 doesnt contain world


matchobj = re.match('Hello(.*)World', text1)
print(matchobj)

#result: <_sre.SRE_Match object at 0x10043b80a>

print(matchobj.group(1))

#result: spam...

#why? matchgroup1 is in brackets, this is what the Hello World is surrounding in text1

#pattern compilation

pattObj = re.compile('Hello(.*)World')
matchobj = pattobj.match(text1)
```

```
#same result as above

#more complicated patteerns

#matches ical datetime objects
datetime1 = "20220211T110000"

matchobj = re.match("(\d{4})(\d{2})(\d{2})T.*", datetime1)
if matchobj:
    (year, month, day) = matchobj.groups()

 #output 2022 02 11

 #matches dates of the form xx/xx/xxxx
 #or of the form x/x/xx
 #the question mark quantifier makes the preceding group optional
 pattobj = re.compile("(\d\d?)/(\d\d?)(\d\d(\d\d)?)")


 (day, month, year, _) = matchobj.groups()
 #leave the last blank because we only want the last two digits of the year

 #sometimes we put an r in front of a string in python in order to not
 #interpret any special python characters
 pattobj = re.compile(r"\b(\d\d?)/(\d\d?)(\d\d(\d\d)?)")
 #\b is backspace in python, whereas it means a word boundary in regex

 line1 = "Michael Zastre"

 line1 = re.sub("Michael", "Mike", line1)

 #output Mike Zastre

price = 3.141500002

p = re.sub(r"(\.\d\d[1-9]?)\d*", r"\1", p)
#the r"\1" refers to the first match group (back reference)

#usually substitutions are done linearly

letter = re.sub(r"=LOCATION=", place, blank_letter)
letter = re.sub(r"=TITLE=", title, letter)
...
```

- a match object is an interface whihc we can used to extract matched substrings
- Match groups: allow us to extract substrings of a match
- Groups can be nested but they can never overlap
- we can reduce overhead by compiling regexes, this prevents them from having to be reconstructed again (DFA construction is expensive but running through DFA is O(N))
- Match vs search, match has automatic anchors, search allows us to look anywhere in a string without anchors.
- Metasymbols/metacharacters match one or more character, but some are meant to match a specific position
- substitutions replace matched words, and will replace all matches
- re.sub

7.2 C Regex

```
//regex in C is much like regex in python other than metasymbols
//\d is [[:digit]] \w is [[:alnum]]

//regex ingredients
```

```
regex_t //regular expression iteself
regmatch_t //indicate where match patterns begin and end
regcomp //all regexes must be compiled in C (thus also malloced)
regfree //releases memory from compiling regexes

int status;
regex_t re;
regmatch_t match[4]; #match is an array which can identify 4 groups

char *pattern = "([[:digit]]+)";
char *search_string = "abc def 123 hij";

#REG_EXTEND uses extended regular expressions
if(regcomp(&re, pattern, REG_EXTEND) != 0){
    return 0;
}

status = regexec(&re, search_string, 2, match, 0){
    if(status != 0){
        printf("No match\n");
        return 0;
    }
}

char match_text[100];
#rm_eo = regex match ending offset
#rm_so = regex match starting offset

#have to precisely extract needed string from regex in string

stnrcpy(match_text, search_string+match[1].rm_so, match[1].rm_eo - match[1].rm_so);

match_text[match[1].rm_eo - match[1].rm_so + 1] = '\0';
```

- recomp takes three parameters:
  - address to a regext variable
  - actual pattern to search for (in POSIX form)
  - Flags
- regexec takes five parameters
  - Address to a regext variable
  - The string to be searched
  - The max number of groupings in the pattern
  - the match array itself len must = prev parameter
  - flags

8 Debugging

- What defines a bug:
- failure, fault, or error
- failure: something visible to the programs users
- fault: the state of the program which leads to a failure
- error: incorrect code fragment which leads to the fault and failure

Observable Failures

- Infinite loop

- deadlock
- incorrect output
- memory leak
- Run-Time Exception

Run Time Failure Examples:
Blue Screen of Death: Usually caused by kernel panic.
Segmentation Fault: Usually happens when we try to write an address we arent supposed to. Program tries to access a memory location outside its accessible range.
Bus Error: Occurs when we have tried to access a misaligned data item. Normally we have an address to an int, and we are adding to it, going off word boundary.
Invalid Instruction: Can sometimes happen when accessing data at invalid array locations. The program has tried to execute an unassigned or privileged opcode.
more runtime errors include:

- ArithmeticException
- ArrayIndexOutOfBoundsException
- NullPointerException
- ClassCastException
- OutOfMemoryException

8.1 The Knuth/Barr Bug Categories:

A: Algorithm: The code follows the intent of the programmer but the intent was wrong Subcategories:

- Off-by-one: miscounting by one
- Validation: Failure to check whether an algorithm is valid or not.
- logic: simply wrong algorithm.
- performance: Inappropriate or coding error

D: Data: The code read/s writes incorrect data or access a wrong storage location

- index: wrong array index.
- number: issues to how numbers are represented in the computer.
- limit: incorrect handling for the first or last elements in a sequence.
- memory: mismanagement of memory

F: Forgotten: A missing action statement or maybe wrong place in the program

- init: forgetting ot initialize a variable or data structure
- location: code appears in wrong place or is out of order
- missing: forgetting to provide code for a certain case

B: Blunder: The algorithm is correct but was implemented wrong or mistyped

- variable: writing the wrong variable name
- language: mistake encouraged by the language syntax (= vs ==)
- expression: incorrectly written expression

8.2 Bug Avoidance Categories:

- Use a high level language
- Think before you code
- Write robust code
  - Validate: every function/subroutine should validate inputs
  - Assertions: Use assertions to verify that pointers are non-null and array indices in range
  - reuse: Library functions and classes wherever possible
  - check results: Make sure to check return values
  - KISS: Keep it simple
  - Watchout for gotchas: (a = true) vs (a == true)

9 Random Special Notes:

- Break vs Exit in C - break transfers control to the next level of the program - exit kills the program
- Contents of #include are expanded in place within the C file upon compilation (i.e. stdio (mostly function prototypes to tell program that they exist))
- Difference between i++ and ++i, i++ will give back i and then increment ++i will increment then give back i

```
//ie. i = 314
printf("\%d \n", ++i);
//output = 315

printf("\%d \n", i++);
//output = 314
```

- Python is dynamically typed whereas C is statically typed.
- **Void means nothing whereas void * means everything, (void *) is a pointer which can point to anything**
- In Zastre terms void * is a variable which holds an address to anything

SIZEOF:

- int: 4 bytes
- char: 1 byte
- float: 4 bytes
- double: 8 bytes

Linking:

- Certain .h files require linking (math.h) which we can do with the