COM S 229 (1) Advanced Programming Techniques Spring 2014

Course Outline

Introductory Things

1. Brief intro to Unix
2. Commands and other info

C Programming Language

1. Philosophy of C
2. Comparison to Java
3. Stages of Compiler, Source Code Organization
4. Using Libraries
5. Make
6. File I/O
7. Types, Operators
8. Arrays and Memory Management
9. Variables, Globals, Modifiers.
10. Function parameters. Flow control.
11. Strings.
12. Pointers.
13. Preprocessor.

C++ Programming Language

1. consts and references
2. inlined functions and overloading.
3. classes.
4. inheritance
5. polymorphism
6. exceptions, nested classes, namespaces

7. templates
8. memory management
9. OO design basics

1 Lecture 1: January 13, 2014 : Introduction to Unix

Welcome.

Course Syllabus, Policies, Programming Skills Builder.

1.1 Brief Intro to Unix

Unix has a GUI (multiple!) just like Windows. (Mac OS is built on Unix!)

But we can do almost everything through the shell (and better!)

Getting a Pyrite account(?)

Logging in to Pyrite.

Mac OS : Terminal - ssh

Linux : Shell - ssh

Windows : Putty

1.2 What is a shell?

A program to parse and execute commands. (You can write your own!)

A/K/A the "Command Line"

Runs interactively:

- Get a prompt
- Type a command, type enter.
- Command executes.
- Get another prompt.

Commands look like:

```
cmd arg1 arg2 ...  argn
```

The command is either a built-in utility or a program living somewhere in the filesystem.

Arguments are separated by whitespace, represent information to assist the command in its execution.

1.3 Man

`man` — Display the manual pages.

First argument includes the command to show the page for. (Standard commands only)

Show "man man"

Switches: Change the behavior very slightly. Learn by using `man`

Sections:

Sec. 1 - General commands

Sec. 3 - C library functions

Sec. 2 - system calls (i.e. forking, threading, semaphores)

1.4 File Structure

`pwd` – Present Working Directory

`cd` – Change directory

`mkdir` – Create a directory

`ls` – Show files

`cp` – Copy file

`mv` – Move files

`rm` – Remove files. (WARNING: No "Trash" or "Recycle Bin")

1.5 Text Files

`cat` – Catenate files (or just list them)

`less` – View (less of) a fie.

`view` – runs `vi` in a read-only mode. Allows for `vi` commands for searching and moving.

1.6 Text Editing

`vi` – What I will be using.

`emacs`

`pico`

1.7 Redirection

Many of these commands will send output to the terminal. Instead of sending it to the screen, you can send it to a file!

```
cat file1.txt file2.txt file3.txt > allfiles.txt
```

Or, perhaps you would rather have a command read from a file, instead!

```
sort < infile.txt
```

Or both!

```
sort < in.txt > out.txt
```

We can *append* to an existing file:

```
cat file4.txt >> allfiles.txt
```

1.8 Pipes

Take the output of one command, and send it to another.

```
cat file1.txt file2.txt | sort
```

```
cat file1.txt file2.txt | sort | uniq | wc -l
```

1.9 Other Helpful Commands

`time` – Followed by a command, times how long it takes!

`sort` – Sort the lines of the input.

`uniq` – Filter out non-unique lines (only checks consecutive lines for uniqueness)

`wc` – Count words and lines. (`wc -l` counts the number of lines!)

`scp` – Secure copy. Allows for copying files across ssh.

```
scp myfile.txt user@pyrite.cs.iastate.edu:mydir/
```

`tail` – Read the last few lines of a file.

`head` – read the first few lines of a file.

2 Lecture 2: January 15, 2014 : Getting the basics of C

(via Skype)

A quick introduction to getting started in C.

NO OBJECTS!

Procedural: Every procedure/function is "global".

Variables can be global, too!

ANSI comments are only `/* */`, not `//`

2.1 Hello, World!

```
/* hello.c
 * A standard hello world program!
 */

#include <stdio.h>

int main(void)
{
    printf("Hello, World!\n"); /* for effect, leave out \n */

    return 0; /* A non-zero return value is an error signal. */
}
```

Variables and flow control work almost exactly the same as in Java.

Fibonacci

Demonstrate for loops, basic math, and formatted output. Also, long types.

```
/* fibonacci.c
 * List the first few fibonacci numbers
 */

#include <stdio.h>

int main(void)
{
    int n = 130; /* Start with n = 30 */

    /* the fibonacci numbers start with 0, 1, and then every other term is the
     * sum of the two previous terms. */

    int i;
    long long int a = 0;
    long long int b = 1;

    printf("%20lld\n", a); /* start with no digit specification, then increase it */
    printf("%20lld\n", b);

    for ( i = 2; i <= n; i++ )
    {
        long long int c = a + b;
        a = b;
        b = c;

        printf("%20lld\n", c);
    }

    return 0;
}
```

Structs.

3 Lecture 3: January 17, 2014 : C Basics

They already know: basic compiling commands, basic use of printf/scanf for numbers.

Today's goals: learn basic string IO, basic structure creation, functions.

Below is a code file that grows iteratively.

3.1 meet

```
/* meet.c
 * Get to know a person
 */

#include <stdio.h>

const int current_year = 2014;

int main(void)
{
    printf("Hello! How old are you?\n>");

    int age;
    scanf("%d", &age);

    int birthyear = current_year - age;
    printf("You were born in either %d or %d!\n", birthyear, birthyear-1);

    return 0;
}
```

Add a string buffer, input the name and output it.

First use just a first name, but then try a First & Last name, showing that we need something else if we want a whole line or more than one word.

```
/* meet.c
 * Get to know a person
 */

#include <stdio.h>

const int current_year = 2014;

int main(void)
{
    printf("Hello! How old are you?\n>");

    int age;
    scanf("%d", &age);

    int birthyear = current_year - age;
    printf("You were born in either %d or %d!\n", birthyear, birthyear-1);

    char buffer[1000];

    printf("What is your name?\n>");
    scanf("%s", buffer);
    printf("Well, hello, %s. Nice to meet you!\n", buffer);

    return 0;
}
```

Replace use of scanf with getchar()

```
/* meet.c
 * Get to know a person
 */

#include <stdio.h>

const int current_year = 2014;

int main(void)
{
    printf("Hello! How old are you?\n>");

    int age;
    scanf("%d", &age);

    int birthyear = current_year - age;
    printf("You were born in either %d or %d!\n", birthyear, birthyear-1);

    char buffer[1000];

    printf("What is your name?\n>");

    int i = 0;
    char in = getchar(); /* gets rid of the \n from the last prompt! */
    while ( i < 1000 && (in = getchar()) != '\n' )
    {
        buffer[i] = in;
        i++;
    }
    buffer[i] = 0;

    printf("Well, hello, %s. Nice to meet you!\n", buffer);

    return 0;
}
```

Now let's create a struct to store information about a person.

```
/* meet.c
 * Get to know a person
 */

#include <stdio.h>

const int current_year = 2014;

struct person
{
    int age;
    char name[1000];
};


int main(void)
{
    printf("Hello! How old are you?\n>");

    struct person p;
    scanf("%d", &(p.age));

    int birthyear = current_year - p.age;
    printf("You were born in either %d or %d!\n", birthyear, birthyear-1);

    printf("What is your name?\n>");

    int i = 0;
    char in = getchar(); /* gets rid of the \n from the last prompt! */
    while ( i < 1000 && (in = getchar()) != '\n' )
    {
        p.name[i] = in;
        i++;
    }
    p.name[i] = 0;

    printf("Well, hello, %s. Nice to meet you!\n", p.name);

    return 0;
}
```

Now let's create a function to perform the greeting.

```
/* meet.c
 * Get to know a person
 */

#include <stdio.h>

const int current_year = 2014;

struct person
{
    int age;
    char name[1000];
};

struct person getToKnow()
{
    printf("Hello! How old are you?\n>");

    struct person p;
    scanf("%d", &(p.age));

    int birthyear = current_year - p.age;
    printf("You were born in either %d or %d!\n", birthyear, birthyear-1);

    printf("What is your name?\n>");

    int i = 0;
    char in = getchar(); /* gets rid of the \n from the last prompt! */
    while ( i < 1000 && (in = getchar()) != '\n' )
    {
        p.name[i] = in;
        i++;
    }
    p.name[i] = 0;

    return p;
}

void sayHi(struct person p)
{
    printf("Hello, %s. Nice to meet you!\n", p.name);
}

int main(void)
{
    struct person p = getToKnow();
    struct person q = getToKnow();

    sayHi(p);
    sayHi(q);

    return 0;
}
```

4 Lecture 4 : Jan 22, 2014

4.1 Why Learn C? C++?

1. Jobs.

2. Legacy Code.

The same is for Cobol and Fortran, but not nearly as much. C and C++ were used to build the personal computing explosion in the 90's and into the early 2000's.

3. Still used for many things today, especially for:

   1. Operating Systems: Windows/Linux/Mac Kernel

   2. Device drivers

   3. 3D Video Games!!!! (Few layers of abstraction means that more performance can be extracted from hardware.)

4. Concepts and syntax from C and C++ have been inherited to many other languages: Java, C#, Javascript.

5. Techniques are still useful. C/C++ require you to know the details of how the computer works, Java does not.

6. C/C++ are very good choices for new projects, especially when speed is important, or you want to run something on a large number of devices that you do not control, such as a supercomputer.

KID GLOVES ARE OFF

You will need to learn memory management for C/C++. Java, C#, other languages usually include this, but they are HIDING the truth from you! In fact: there are memory errors in this language *precisely because* you have no control over the memory.

4.2 C History

- 1969 – Ken Thompson wrote first UNIX system.
  - In assembly.
  - On a PDP-7 Computer, DEC, 1965. 18-bit architecture, about 144K of memory.
- Thompson wanted a *language* to abstract some of the common patterns of assembly.
  - Started with BCPL : Basic Combined Programming Language.
  - Stripped it down, called it B
  - B was not quite right.
- Dennis Richie came in, added types and other features to B. Named it C (it's one better!)
- UNIX was ported to a PDP-11 computer, using C to implement the kernel!
  First language to be written in a "high-level" language.
- C and UNIX were henceforth developed together.

4.3 C Philosophy

Computers at the time: not very powerful, limited memory.

Programmers at the time: Very concerned with efficiency for computation time AND memory.

System Programmers: Need to access things "directly", and be able to set individual bits of memory.

Overal C Philosophy: No feature should be added if it causes a speed or memory penalty for programs that do not use the feature.

The C Language is very *minimalist* according to today's standards.

At the time, it was revolutionary!

- Flow Control (branching, loops)
- User-defined procedures (functions)
- Economy of expression (another philosophy) given by a rich set of operators.

$$+, -, *, /, \hat{}, \&, |, \&\&, |||, ...$$

- Easy to compile into efficient code.

In C — **The Programmer is always right!**

This means: very little error-checking. If a user CAN do something, then that means the programmer meant for that to happen!

```
int x[10];
x[54] = 350;
```

Q: What happens in Java?

In C:

- No Compile Error.
- No Exception Thrown.
- One of 3 possible things will happen:
  - Program runs fine, no errors.
  - Program does something strange, much later.
  - Segmentation fault immediately.

  Which one occurs depends on: input to the program, which compiler you use, which system you run it on!!!

4.4 Versions of C

- 1978 : Brian Kernighan and Dennis Richie publish "The C Programming Language". Is a standard, mainly written for people making compilers.
- 1989: C language standard completed by ANSI (American National Standards Institute). Used in Second Edition of K& R.
- 1999: C99 – Standard was updated, adopted in 2000.
- 2011: C11 – Standard updated, adopted in 2011.

Different compilers may have "non-standard" features, or may treat "undefined behavior" differently.

(This is why your code may fail to compile or run correctly when porting between systems.)

4.5 Comparing C and Java

Operators and syntax are very very similar, but **be careful of differences!**

Both: Programs are stored in text files and can be divided into several files.

Java: Compiles into Byte Code, which is then interpreted by the Java Runtime Environment.

- Allows Java to be platform-independent.
- JRE is implemented natively.

C: Compiles into machine language (usually with assembly as an in-between step!)

- Runs "directly" on processor.
- BUT: machine language code is *not* portable across different processors.

Some Java Features you should forget about in C:

- Type Safety.

  C has type checking, but you can arbitrarily cast data into different types, and not always strict (i.e. printf), and there is no "private" or "protected".

- Bounds Checking.

  C will take any array length or memory location and assume the programmer is doing it on purpose, including `x[-1]`.

- Garbage Collecting.

  In C, you are in charge of creating and deleting stores of memory. (MUCH more on this later!)

- Objects!

  All functions in C are "global" and have no "base object". We have structs to group data, but everything is public and no inheritance or polymorphism.

  We can "fake" these things in C, but it requires some difficulty.

4.6 Stages of the Compiler

When we did `gcc -o fibonacci fibonacci.c`, the compiler did 4 things.

- **(Stage 1) Preprocessor.**

  Lines that start with # are called "preprocessor directives". You have seen some of these:

  1. `#include <file>` or `#include "file"` loads a file and "pastes" it where the `#include` sits.

     Also interprets all code that is pasted there, for other preprocessor directives.

  2. `#define symbol value` takes all instances of "symbol" in the remaining code and replaces it with "value".

     By convention: symbols are all upper case, but not required. Example (in `fibonacci.c`)

     ```
     #define FIB0 0
     #define FIB1 1

     ...

     int a = FIB0;
     int b = FIB1;

     ...
     ```

  Fun Trick: `gcc -E code.c` will stop after the preprocessor! **Try It:** `gcc -E fibonacci.c`

- **(Stage 2) Compilation.**

  C code is converted into assembly code.

  Fun Trick: `gcc -S code.c` will stop after the compilation, and will output the assembly code, into a `.s` file (a text file!)

  Important: C is a *single-pass* compiler! It goes through the code ONCE! Java is two-pass: it looks through the code for all possible variable and function definitions (inside an object) and uses those in the second pass while encoding the internals of a function. This is why C requires function prototypes and header files at the BEGINNING of a file!

- **(Stage 3) Assembler.**

  Converts assembly code (a text file) into machine language (a binary file).

  Not yet an executable! Just an "object file". (NOTHING TO DO WITH JAVA OBJECTS)

  It contains a list of *callable procedures*

  `gcc -c` will stop at this point and output a `.o` file.

  You *will* use this for your projects! (More about this when talking about Makefiles)

- **(Stage 4) Linker.**

  Input: One or more object files.

  Output: A full executable (or shared library).

  IDEA: Pulls all the necessary bits together into a single program that can run when called!

  `gcc -o name` will make sure this executable has the given name, and not `a.out`.

5 Lecture 5 : January 24th, 2014

Let's recap the stages of a compiler

- **(Stage 1) Preprocessor.**
- **(Stage 2) Compilation.**
- **(Stage 3) Assembler.**

  Converts assembly code (a text file) into machine language (a binary file).

  Not yet an executable! Just an "object file". (NOTHING TO DO WITH JAVA OBJECTS)

  It contains a list of *callable procedures*

  `gcc -c` will stop at this point and output a `.o` file.

  You *will* use this for your projects! (More about this when talking about Makefiles)
- **(Stage 4) Linker.**

  Input: One or more object files.

  Output: A full executable (or shared library).

  IDEA: Pulls all the necessary bits together into a single program that can run when called!

  `gcc -o name` will make sure this executable has the given name, and not `a.out`.

Let's see an example! Download `ttt.c`, `tictactoe.h`, `tictactoe.c`, and `ttttboard.h`.

The following Makefile contains the proper compiling commands, in addition to some extra info! (We'll do makefiles in a bit)

```
# Makefile for Tic-Tac-Toe

all: ttt tictactoe.o


ttt: tictactoe.o tictactoe.h tttboard.h
    gcc -ansi -o ttt ttt.c tictactoe.o

tictactoe.o: tictactoe.c tictactoe.h tttboard.h
    gcc -ansi -c tictactoe.c

clean:
    rm ttt tictactoe.o
```

TODO: Draw the file structure for the commands.

TODO: Compile, run, and briefly test the program.

5.1 Source Code Organization

Option 1: One huge file, "program.c" and compile with `gcc -o program program.c`

This will be VERY SLOW! Editing becomes tediuous and frustrating!

Idea: Any source file longer than 1000 lines is probably too long!

Similar Idea: Any single procedure longer than 300 lines is probably too long!

Option 2: Several `.c` files containing different, interrelated functions.

Editing problem goes away.

How do we compile these files? Use `gcc -c`

Use the linker to combine these object files with `gcc -o program objectfile1.o objectfile2.o objectfile3.o [program.c]`

If one `.c` file wants to use another, then we need to have **prototypes** in a `.h` file!

If you put *implementations* in a `.h` file, and multiple object files include that file, then there are *competing* implementations when trying to link! THIS IS A PROBLEM!

Best Practices: Always have one file `program.c` that contains the `main()` method for `program`.

WARNING: In C, the file names have *no significance!* But you should be putting reasonable names on things!

5.2 Splitting Your Source – A "How To" Guide

1. Divide source into separate `.c` files.
2. Function prototypes and other declarations that must be "visible" go into `.h` files.
3. Any source file (`.c` or `.h`) that needs to "see" a declaration, should #include the `.h` file.
4. Compile `.c` files with `gcc -c file.c` (produces `file.o`)
5. Do NOT compile the `.h` files (no implementations!)
6. Link object files with `gcc -o name file.o ...`
7. If you modify a `.c` file, rebuild its object file and relink.
8. If you modify a `.h` file, rebuild object files for all `.c` files that include it (even "down the chain")
9. The linker will fail in strange ways for lots of reasons, so be careful!

5.3 UNIX Trick: `grep`

Suppose I forgot which header file declares a function prototype.

Use `grep` to find any lines containing a certain string.

Example: To check all `.h` files in the current directory for `player1`, use:

```
grep player1 *.h
```

Bonus Example: `grep player1 -n *.h`

5.4 Using Libraries

If you want to use the methods of a library, use `#include "libcool.h"` to include the definitions.

Compile object files as necessary.

When coming to linking mode, you need to tell the linker where these libraries are located!

Standard Libraries such as `stdio`, `stdlib` and `string` are automatically linked.

On some machines, you need to use `-lm` flag to link the `math.h` library!

If you are using a non-standard library, you must share the compiled library files. These could be `.o` files or `.so` files, or many others.

SEE LIBRARY DOCUMENTATION FOR MORE LINKING INFORMATION!

5.5 Makefiles

It is frustrating to keep all of this source code organization up-to-date.

Q: How do I know which object files are out of date?

Q: Can I keep myself from running command after command?

Q: What if I have trouble remembering all my object files when coming to linking time?

`make` is here to save us!

`make` is a system for automatically building/rebuilding a file or files *when other files change.*

We can manage complex chains of dependencies!

All managed by a plain-text file called a "makefile" (and usually is called `Makefile`)

See `man make` for all the gritty details, flags, and switches.

Main syntax:

```
# This is a comment!

target : List of dependencies
<tab> command to build target
```

Create and investigate the makefile for `ttt`

Special targets:

`all` (what to do when just `make` is given with no target)

```
all : program file1.o file2.o file3.o
```

`clean` (Usually reserved to delete all binary files (all targets))

```
clean :
    rm program file1.o file2.o file3.o
```

`tarball` (Compression of all source code. Helpful for project!)

```
tarball :
    tar czf source.tar.gz foo.c bar.c bar.h makefile
```

6 Lecture 6 : January 27, 2014

Start by talking about TA Office Hours and Project 1.A.

6.1 Command-Line Arguments

A complicated subject in C is the idea of a "pointer". (The `*` symbol means "pointer".)

For Now: Assume that a `*` means "array". Multiple stars means multiple dimensions!

Also, recall that `char*` is a *string*.

```
int main(int argc, char** argv)
{
    /* argc counts the number of command-line arguments (including the executable's name) */

    /* argv is an array of char arrays */
    /* i.e. an array of strings */
    int i = 0;
    for ( i = 0; i < argc; i++ )
    {
        printf("%s\n", argv[i]);
    }

    return 0;
}
```

7 Lecture 7 : January 29th, 2014

7.1 File I/O

You've already been doing this, with `stdin` and `stdout`.

Now, we need to do it with actual files in the filesystem!

All I/O is in `stdio.h`

1. Declare a variable of type `FILE*`. (What is that `*` for? We'll get to it later)

```
FILE* infile;
```

2. Open the file for reading, writing, or both! (and possibly binary mode!)

```
FILE* infile = fopen( "file.txt", "r");
FILE* outfile = fopen( "file2.txt", "w");
FILE* iofile = fopen("file3.txt", "rw");
FILE* bfile = fopen("muzak.mp3", "rb");
```

3. Test that the file opening succeeded!

```
if ( infile )
{
    /* This executes if the open succeeds! */
}

if ( !outfile )
{
    /* This executes if the open fails! */
}
```

4. To read/write as in printf/scanf, use `fprintf` and `fscanf`:

```
fscanf( infile, "%d", &num_tacos);
fprintf( outfile, "There are %d tacos!\n", num_tacos);
```

5. Close the file when you are done!

```
fclose(outfile);
```

6. Optional: "flush" the file buffer to be sure things are updated correctly:

```
fflush(outfile);
```

7.2 Summary of File Input Methods

1. `fscanf( FILE*, formatString, args )` – Like scanf, but for a file!
2. `fgetc( FILE* )` – Get a single character
3. `fgets( char array, size, FILE* )` – Read a line of text from a file, store it in the array, up until the given size.
4. `fread( data pointer, wordsize, length, FILE*)` – Read binary data from a binary file. **IMPORTANT FOR PROJECT!**

REALLY COOL TRICK!

`ungetc(char, FILE*)` — Put a character back onto a file's input stream (does NOT modify the file, it's just a way to "go back").

7.3 Summary of File Output Methods

1. `fprintf( FILE*, formatString, args )` – Like printf, but for a file!
2. `fputc( FILE* , char )` – Write a single character to file.
3. `fputs( char array, FILE* )` – Write a given string to a file.
4. `fwrite( data pointer, wordsize, length, FILE*)` – Write binary data to a binary file. **IMPORTANT FOR PROJECT!**

7.4 Simple Things To Do

File Copy (i.e. replace `cp`)

```
#include <stdio.h>

int main(int argc, char** argv)
{
    FILE* infile = fopen(argv[1], "r");
    FILE* outfile = fopen(argv[2], "w");

    char c = 0;

    while ( ( c = fgetc( infile ) ) != EOF )
    {
        fputc( outfile, c );
    }

    fclose(infile);
```

```
    fclose(outfile);

    return 0;
}
```

7.5 Standard Files

Three standard files:

- `stdin` — Standard input, read only

  `scanf(...)` is equivalent to `fscanf(stdin, ...)`

- `stdout` — Standard output, write only

  `printf(...)` is equivalent to `fprintf(stdout, ...)`

- `stderr` — Standard error, write only, used for error messages (so they do not corrupt output redirection to file or pipes)

7.6 Types in C

Lots of the primitive types are similar to those in Java. Differences: some can be modified with `short`, `long`, `signed` or `unsigned`. For most types, `signed` is default!

In the table below, the ranges are the *minimum* ranges based on the ANSI standard. Actual implementations could have larger sizes!

Type	**Description**	`short`	(normal)	`long`	`long long`
`int`	A signed integer, using twos-complement encoding, with given ranges.	2 bytes, 16 bits	2 bytes, 16 bits	4 bytes, 32 bits	8 bytes, 64 bits
`char`	A single byte, frequently used for storing an encoding of a letter using the ASCII table.	N/A	1 byte	N/A	N/A
`float`	A floating-point representation of a real number (to finite accuracy)	N/A	4 bytes, 32 bits	N/A	N/A
`double`	A floating-point representation of a real number (to finite accuracy)	N/A	8 bytes, 64 bits	16 bytes, 128 bits[1]	N/A

For more information, see http://en.wikipedia.org/wiki/C_data_types

C99 added a `_Bool` type if you include `stdbool.h`. (NOT for `-ansi`)

Since no `bool`, there is no `true` or `false`!!! `if`, `for`, and `while` statements all evaluate any non-zero value as "true" and only something equal to zero as "false".

Download the code file http://orion.math.iastate.edu/dstolee/229/code/types.c and compile it to an executable. This will output more data on what pyrite uses for its type lengths.

7.7 Characters

The type `char` is a very simple data type: just one byte. However, it's encoding more complicated information! Each letter, punctuation mark, and other things are encoded by the following table:

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(nul)	0	0000	0x00	(sp)	32	0040	0x20	@	64	0100	0x40	’	96	0140	0x60
(soh)	1	0001	0x01	!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
(stx)	2	0002	0x02	"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
(etx)	3	0003	0x03	#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
(eot)	4	0004	0x04	$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
(enq)	5	0005	0x05	%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
(ack)	6	0006	0x06	&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
(bel)	7	0007	0x07	’	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(bs)	8	0010	0x08	(	40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
(ht)	9	0011	0x09	)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
(nl)	10	0012	0x0a	*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
(vt)	11	0013	0x0b	+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
(np)	12	0014	0x0c	,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
(cr)	13	0015	0x0d	-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
(so)	14	0016	0x0e	.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
(si)	15	0017	0x0f	/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
(dle)	16	0020	0x10	0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
(dc1)	17	0021	0x11	1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
(dc2)	18	0022	0x12	2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
(dc3)	19	0023	0x13	3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
(dc4)	20	0024	0x14	4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
(nak)	21	0025	0x15	5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
(syn)	22	0026	0x16	6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
(etb)	23	0027	0x17	7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
(can)	24	0030	0x18	8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
(em)	25	0031	0x19	9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
(sub)	26	0032	0x1a	:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
(esc)	27	0033	0x1b	;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b
(fs)	28	0034	0x1c	<	60	0074	0x3c	\	92	0134	0x5c	|	124	0174	0x7c
(gs)	29	0035	0x1d	=	61	0075	0x3d	]	93	0135	0x5d	}	125	0175	0x7d
(rs)	30	0036	0x1e	>	62	0076	0x3e	^	94	0136	0x5e	~	126	0176	0x7e
(us)	31	0037	0x1f	?	63	0077	0x3f	_	95	0137	0x5f	(del)	127	0177	0x7f

Using apostrophes, as in `’a’` or `’
n’`, you can access the "number" for the letter without remembering what the ASCII table says.

```
char c = getchar();
if ( c >= ’a’ && c <= ’z’ )
{
    printf("%c (%d) is lower-case\n", c, (int)c);
}
else if ( c >= ’A’ && c <= ’Z’ )
{
    printf("%c (%d) is upper-case\n", c, (int)c);
}
else if ( c >= ’’0’ && c <= ’9’ )
{
    printf("%c (%d) is a number\n", c, (int)c);
}
else
{
    printf("%c (%d) is something else!\n", c, (int)c);
}
```

8 Lecture 8 : January 31st, 2014

Discuss what happens when creating an array with `char buffer[1024];` and why you cannot do `char buffer[n];` where `n` is a variable.

The `*` symbol means "pointer". (Generally, this means that a variable with type "`type*`" is a number that stores a memory location where something of type "`type`" lives.)

For arrays, these *pointers* point to the start of an array.

```
#include <stlib.h>

int main(void)
{
    char s1[100];

    int length = 100;
    char* s2 = (char*)malloc(length);

    scanf("%s", s1);
    scanf("%s", s2);

    printf("You typed \"%s\" then \"%s\"!\n", s1, s2);

    free(s2);
    return 0;
}
```

Q: What does this code do?
Q: What happens if someone types a word longer than 100 characters? Exactly 100 characters?

For safety: use `char * fgets ( char * str, int num, FILE * stream );`

Key: Create an array using `malloc` (and cast it into the type) and delete the array using `free`.

```
    int num = 0;
    printf("How many numbers?\n");
    sscanf("%d", &num); /* the & symbol gives the _address_ of num, turning it into a pointer! */

    float* list = (float*)malloc( num * sizeof(float) );

    int i = 0;
    for ( i = 0; i < num; i++ )
    {
        printf("Please enter number %d: ", i);
        float f; /* double d; */
        sscanf("%f", &f); /* sscanf("%lf", &d); */
        list[i] = f; /* = d; */
    }

    printf("Here is the full list:\n\t");
    for ( i = 0; i < num; i++ )
    {
        printf("%02.4f\t", list[i]);
        if ( (i+1) % 5  == 0 )
        {
            printf("\n\t");
        }
    }
    printf("\n");

    free(list);
    return 0;
```

8.1 Doubly-Indexed Arrays

In order to have multiple dimensions in an array, we actually store an *array of arrays*.

```
#include <stdlib.h>

typedef struct matrix_s
{
    int rows;
    int cols;
    double** data;
} matrix;

int main(void)
{
    matrix mtx;

    mtx.rows = 3;
    mtx.cols = 5;

    mtx.data = (double**)malloc(mtx.rows*sizeof(double*));

    int i = 0;
    for ( i = 0; i < mtx.rows; i++ )
    {
        mtx.data[i] = (double*)malloc(mtx.cols*sizeof(double));

        int j;
        for ( j = 0; j < mtx.cols; j++ )
        {
            /* special casting to be careful of int-division rounding! */
            mtx.data[i][j] = ((double)(i+1)) / ((double)(j+1));
        }
    }

    /* TODO: do something with the matrix! */
    if ( mtx.data != 0 )
    {
        for ( i = 0; i < mtx.rows; i++ )
        {
            /* BEST PRACTICE: test for non-zero pointer before freeing */
            if ( mtx.data[i] != 0 )
            {
                free(mtx.data[i]);
                mtx.data[i] = 0;
                /* BEST PRACTICE: Set pointer to zero after freeing */
            }
        }

        free(mtx.data);
        mtx.data = 0;
    }

    return 0;
}
```

9 Lecture 9 : February 3rd, 2014

9.1 Realloc

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

int main(void)
{
    int size = 10;
    int count = 0;
    int i;

    int* numbers = (int*)malloc( size * sizeof(int) );
    bzero( numbers, size * sizeof(int) );

    while ( 1 )
    {
        int in = 0;

        int result = scanf("%d", &in);

        if ( result == 0 )
        {
            /* scanf did not fill it... perhaps end-of-file? */
            break;
        }

        if ( in < 0 )
        {
            /* negative numbers mean stop! */
            break;
        }

        if ( count >= size )
        {
            /* We are out of room! Let's make more room! */
            size += 10;
            numbers = (int*) realloc( numbers, size * sizeof(int));
        }

        numbers[count] = in;
        count++;
    }

    for ( i = 0; i < count; i++ )
    {
        printf("%10d ", numbers[i]);

        if ( (i+1) % 5 == 0 )
        {
            printf("\n");
        }
    }
    printf("\n");

    free(numbers);
    numbers = 0;

    return 0;
}
```

9.2 Memory Management for Structs

Consider a simple structure:

```
typedef struct string_struct
{
    int lengthOfString;
    int lengthOfValue;
    char* value;
} string;
```

Discuss the memory of a variable of type `string`.

What happens in the following code? (Start with s1 and s2, build up s3 and s4.)

```
int main(void)
{
        int n = 1024;

        string s1, s2;

        s1.lengthOfValue = n;
        s1.value = (char*)malloc( s1.lengthOfValue * sizeof(char) );
        scanf("%s", s1.value);
        s1.lengthOfString = strlen( s1.value ); /* needs #include <string.h> */

        s2 = s1;



        int i = 0;

        string* s4 = (string*)malloc(sizeof(string));
        string* s3 = &s2;

        s4->lengthOfValue = s2.lengthOfValue;
        s4->lengthOfString = s2.lengthOfString;
        s4->value = (char*) malloc( s4->lenghtOfValue );
        for ( i = 0; i < s4->lengthOfString; i++ )
        {
                s4->value[i] = s2->value[i];
        }
        s4->value[i] = 0; /* terminate the string! necessary for printf, strlen, etc. */

        for ( i = 0; i < s2.lengthOfString; i++ )
        {
                char c = s2.value[i];

                if ( c >= 'a' && c <= 'z' )
                {
                        s2.value[i] = c + ('A' - 'a');
                }
        }


        s2.lengthOfString = s2.lengthOfString - 1;

        printf("s1 : ");
        for ( i = 0; i < s1.lengthOfString; i++ )
        {
```

```
                fputc( s1.value[i], stdout );
        }

        printf("\ns2 : ");
        for ( i = 0; i < s2.lengthOfString; i++ )
        {
                fputc( s2.value[i], stdout );
        }
        /* What happens if we used printf("%s\n", s2.value) ? */

        printf("\ns3 : ");
        for ( i = 0; i < s3->lengthOfString; i++ )
        {
                fputc( s3->value[i], stdout );
        }
        printf("\ns4 : ");
        for ( i = 0; i < s4->lengthOfString; i++ )
        {
                fputc( s4->value[i], stdout );
        }

        free(s1->value);
        s1->value = 0;

        /* What happens if we do free(s2->value) at this point? */

        free(s4->value);
        s4->value = 0;
        free(s4);
        s4 = 0;
}
```

WARNING!!!!!

If you use `typename variable` inside of a method to create a struct (or primitive) then you should absolutely NOT pass that address as a return to the calling method.

Variables defined directly in the method (without `malloc`) are destroyed automatically when a method returns!

9.3 Using struct pointers in Functions

Consider the following function prototypes using the `string` structure from earlier.

```
/** 1. Create a string structure from a given char array */
string createString( char* s );

/** 2. Create a string structure from a given char array */
string* createString( char* s );
```

Q: What are the benefits of each of these prototypes? How would you use them in code?

A: (1) will *copy* the returned `string`, including the memory address for the `char*` array. (2) will return a pointer to a struct object that must have been allocated with `malloc`. Still, the second is safer and acts more like Java. However, (2) will allocate memory that must be deallocated later with `free`!

```
/** 1. Populate a string structure from a given char array */
void fillString( string str, char* s );

/** 2. Populate a string structure from a given char array */
void fillString( string &str, char* s );

/** 3. Populate a string structure from a given char array */
void fillString( string* str, char* s );
```

Q: What are the benefits of each of these prototypes? How would you use them in code?

A: (1) will *copy* the passed `string`, including the memory address for the `char*` array. If anything in the `string` is changed (such as `lengthOfString` or `lengthOfValue`, or if `value` is reset with `malloc` or `realloc`) then those changes are NOT reflected in the original structure. (2) takes the passed structure as *pass by reference* and the changes to `str` will be reflected in the original structure. (3) takes the passed structure as a pointer, and changes will be reflected. This requires the user to have allocated the `string` in their own way, which is sometimes valuable!

```
/** 1. Read a line from the given file and return a string containing that line. */
string inputLine( FILE* file );

/** 2. Read a line from the given file and return a string containing that line. */
string* inputLine( FILE* file );
```

Q: What are the benefits of each of these prototypes? How would you use them in code?

10 Lecture 10, February 5th, 2014

Q& A for project and C programming basics.

11 Lecture 11, February 7th, 2014 : Debugging!

Sometimes, you write lots of code and it doesn't work! There are *bugs!* Let's debug!

To allow for debuggers to access your code, add the `-g` flag to all of your compilations:

```
gcc -ansi -pedantic -g -c file.c gcc -ansi -pedantic -g -o program program.c
                                    file.o
```

For the examples below, we will use the program found at http://orion.math.iastate.edu/dstolee/229/code/0207/debugex.tar.gz and http://orion.math.iastate.edu/dstolee/229/code/0207/knapsack.tar.gz

Unpack and compile using `make`.

11.1 `valgrind` : Checking for memory errors

To run `valgrind` to check for memory errors, use the command

```
valgrind --leak-check=full --dsymutil=yes --show-reachable=yes cmd arg1 arg2 arg3
                                    ...  argc
```

This will run `valgrind`, surrounding the program `cmd` and the given arguments in a wrapper that keeps track of all memory allocations, deallocations, reads, and writes. For example, after unpacking, type the command

```
valgrind --leak-check=full --dsymutil=yes --show-reachable=yes ./knapsack 100 19 52 34 28}
```

Examples of erroneous output:

```
==6440== Invalid write of size 4
==6440==    at 0x4008DC: main (distance_cliquer.c:94)
==6440==  Address 0x4c3c040 is 0 bytes inside a block of size 3 alloc'd
==6440==    at 0x4A0881C: malloc (vg_replace_malloc.c:270)
==6440==    by 0x400835: main (distance_cliquer.c:86)

==7465== Conditional jump or move depends on uninitialised value(s)
```

```
==7465==    at 0x400828: max_packing (knapsack.c:96)
==7465==    by 0x400611: main (knapsack.c:22)


==6440== Invalid read of size 4
==6440==    at 0x4015CE: unblock (distance_cliquer.c:459)
==6440==    by 0x401B91: cliquer (distance_cliquer.c:649)
==6440==    by 0x401B81: cliquer (distance_cliquer.c:648)
==6440==    by 0x401FC6: getMaxIndepSizeCyclicUsing (distance_cliquer.c:757)
==6440==    by 0x400B29: main (distance_cliquer.c:142)
==6440==  Address 0x4c3c040 is 0 bytes inside a block of size 3 alloc'd
==6440==    at 0x4A0881C: malloc (vg_replace_malloc.c:270)
==6440==    by 0x400835: main (distance_cliquer.c:86)


==6440== 32,032 bytes in 1 blocks are still reachable in loss record 1 of 2
==6440==    at 0x4A0881C: malloc (vg_replace_malloc.c:270)
==6440==    by 0x400FCC: initCliquerData (distance_cliquer.c:277)
==6440==    by 0x400A9D: main (distance_cliquer.c:117)
```

A good strategy: Use this before anything else, ESPECIALLY before turning in an assignment!

Tackle the memory errors from top-to-bottom.

The *very* bottom will mention memory that was not deallocated.

This is particularly important, as it will be where memory leaks are discovered!

11.2 `gdb` : The GNU DeBugger

Use: `gdb program`

Helpful commands:

- `help [command]` — Get some help on how to use the `gdb` command requested.
- `run arg1 arg2 ... argk` — Run the program with the given command-line arguments. During the run: `CTRL+C` and `CTRL+Z` will stop your program from running, but will not kill `gdb`.
- `quit` — Leave the debugger.
- `backtrace` or `bt` — Print the "backtrace" : the sequence of methods that call each other to the current point in code. The numbers on the side refer to the "frames".
- `frame #` or `f #` — Change the "frame" : the method of the current backtrace to be thinking about. All references to variables or methods will be based on the perspective of this frame. (Particularly important for variable names in a recursive call!)

- `break id` — Set a breakpoint at the given identifier. This identifier could be a label (such as `this_spot:` in the code) or it could be a specific line in a specific file (such as `file.c:230`). When execution reaches this point, it will pause and give control to the user.
- `clear id` — Remove a breakpoint at the given identifier.
- `continue` — Continue execution.
- `next` — Perform one more step.

For example, the following commands will help diagnose a problem with `knapsack.c` (line numbers may be wrong...)

```
(gdb) break knapsack.c:100
(gdb) run 100 19 52 34 28
Breakpoint 1, max_packing (N=100, m=4, numbers=0x601010) at knapsack.c:100
100             sum += packing[i];
(gdb) continue
(gdb) print i
$3 = 1
(gdb) print packing[i]
$4 = 34
(gdb) print sum
$5 = 52
(gdb) watch i
Hardware watchpoint 2: i
(gdb) condition 2 i >= 4
(gdb) clear knapsack.c:100
Deleted breakpoints 1 3
(gdb) continue
Hardware watchpoint 2: i

Old value = 3
New value = 4
0x0000000000400877 in max_packing (N=100, m=4, numbers=0x601010) at knapsack.c:98
98        for ( i = 0; packing[i] >= 0; i++ )
(gdb) print i
$6 = 4
(gdb) print packing[i]
$7 = 0
```

12 Lecture 12, February 10th, 2014

12.1 Interesting Things from Piazza

Preprocessor Guards for Header Files. Each of your header files may be included multiple times in a "chain" of includes! To stop your header file from being pasted multiple times, use the following best practice:

```
#ifndef _FILE_H_
#define _FILE_H_

[ #include statements ]

[ FILE CONTENTS ]

#endif
```

(Perform this on `stringstruct.h`)

This will only include the file contents for the first time it is included, and other times it will have no information.

GDB and malloc_error_break. If you are having failures that appear to be memory related, specifically with calls to malloc, realloc, or free, then use the following:

```
gdb ./mypgrogram
(gdb) break malloc_error_break
(gdb) run arg1 arg2 ... argc
```

(Test this on `freeerror.c`)

This will create a breakpoint that will pause the program whenever malloc, realloc, or free "throw" and error.

12.2 Operators in C

Comparison Operators:

```
<  >  <=  >= ==  !=
```

Q: How to compare strings?

A: `int strcmp(char* s1, char* s2)` — If 0, then they are equal. If greater than 0, then str1 is lexicographically bigger than str2. If less than 0, then str1 is lexicographically less than str2.

lexicographic order $\equiv$ *dictionary order* except it is case sensitive!

Arithmetic Operators:

```
+  -  *  /  %
```

(Add, Subtract, Multiply, Divide, Modulus)

If a and b are positive, then `a % b` returns the value r such that $a = qb + r$ for an integer q and $0 \leq r < b$.

EXCEPT: If `a` or `b` is negative, then you may get a *negative* value for r! See http://stackoverflow.com/a/4003287/127088 for more information.

Logic Operators:

```
||  &&  !
```

13 Lecture 13 : February 12th, 2014

Bit-Shift Operators:

`<<  >>`

Move all bits a certain amount to the "left" or "right."

To test: input some numbers (in hexadecimal) and shift them some amount to the right or left:

```
while ( 1 )
{
    int value;
    int shift;
    int result;

    result = scanf("%X %d", &value, & shift);

    if ( result < 2 )
    {
        break;
    }

    if ( shift > 0 )
    {
        value = value << shift;
    }
    if ( shift < 0 )
    {
        value = value >> (-shift);
    }
    printf("%X\n", value);
}
```

Bit-Wise Operators:

`&  |  ^ ~`

(AND, OR, XOR, BIT-WISE NEGATION) Try the following:

```
void printBinary(int a)
{
    int i;
    for ( i = 0; i < 8*sizeof(int); i++ )
    {
        if ( a & (1 << i) != 0 )
        {
            printf("1");
        }
```

```
        else
        {
            printf("0");
        }
    }
}
```

Then, read in two integers, and test the action of these operations:

```
printf("     a : "); printBinary(a); printf("\n");
printf("     b : "); printBinary(b); printf("\n");
printf(" a | b : "); printBinary(a|b); printf("\n");
printf(" a & b : "); printBinary(a&b); printf("\n");
printf(" a ^ b : "); printBinary(a^b); printf("\n");
printf("    ~a : "); printBinary(~a); printf("\n");
printf("    ~b : "); printBinary(~b); printf("\n");
printf("a << 4 : "); printBinary(a << 4);printf("\n");
printf("b >> 8 : "); printBinary(b >> 8);printf("\n");
```

```
Please enter a and b (in hexadecimal):
00F343E17 FF0A37034
     a : 0000 1111 0011 0100 0011 1110 0001 0111
     b : 1111 0000 1010 0011 0111 0000 0011 0100
 a | b : 1111 1111 1011 0111 0111 1110 0011 0111
 a & b : 0000 0000 0010 0000 0011 0000 0001 0100
 a ^ b : 1111 1111 1001 0111 0100 1110 0010 0011
    ~a : 1111 0000 1100 1011 1100 0001 1110 1000
    ~b : 0000 1111 0101 1100 1000 1111 1100 1011
a << 4 : 1111 0011 0100 0011 1110 0001 0111 0000
b >> 8 : 1111 1111 1111 0000 1010 0011 0111 0000
```

Unary Operators: (let `int a;` be defined)

```
-a  !a  ~a  ++a  a++  --a  a--
```

(Integer Negation, Logical Negation, Bitwise Negation, Increments, and Decrements)

Difference between `++a` and `a++` : prefix/postfix. Try the following:

```
int a, b, i;
a = 0;
b = 0;
for ( i =0; i < 10; i++ )
{
    printf("++a=%2d b++=%2d\n", ++a, b++);
}
```

13.1 The C String Library

Methods in `string.h`.

There are usually two versions: one with 'n' and one without.

The 'n' version includes a `num` parameter, saying how many characters to use. (This does NOT count the terminating zero!)

String Manipulation Methods

```
size_t strlen ( const char * str );

char * strcpy ( char * destination, const char * source );
char * strncpy ( char * destination, const char * source, size_t num );

char * strcat ( char * destination, const char * source );
char * strncat ( char * destination, const char * source, size_t num );

int strcmp ( const char * str1, const char * str2 );
int strncmp ( const char * str1, const char * str2, size_t num );
```

String Searching Methods

```
const char * strchr ( const char * str, int character );
      char * strchr (       char * str, int character );

const char * strrchr ( const char * str, int character );
      char * strrchr (       char * str, int character )

const char * strpbrk ( const char * str1, const char * str2 );
      char * strpbrk (       char * str1, const char * str2 );

size_t strspn ( const char * str1, const char * str2 );
size_t strcspn ( const char * str1, const char * str2 );

const char * strstr ( const char * str1, const char * str2 );
      char * strstr (       char * str1, const char * str2 );

    char * strtok ( char * str, const char * delimiters );
```

14 Lecture 14 : February 17th, 2014

Discuss Project 1B.

14.1 String Manipulation

Warning: I do not condone pointer arithmetic, EXCEPT in the case of string manipulation (or `char` arrays in general). This is more due to how the C string library works than an endorsement of pointer arithmetic. If you use pointer arithmetic for any other data type, then I *will not help you* resolve your problems!

Recall that a C string is just an array of `char`s with a terminating zero. When performing string manipulation, we can exploit that fact!

Suppose we have the string "Hello! It is good to see you."

'H'	'e'	'l'	'l'	'o'	'!'	' '	'I'	't'	' '	'i'	's'	' '	'g'	'o'
'o'	'd'	' '	't'	'o'	' '	's'	'e'	'e'	' '	'y'	'o'	'u'	'.'	0

However, if we replace all instances of the space character (`' '`) with 0, then we have several strings in a row, each being null-terminated!

'H'	'e'	'l'	'l'	'o'	'!'	0	'I'	't'	0	'i'	's'	0	'g'	'o'
'o'	'd'	0	't'	'o'	0	's'	'e'	'e'	0	'y'	'o'	'u'	'.'	0

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

int main(int argc, char** argv)
{
    int i, len1, len2;
    char* str1 = (char*)malloc(50);
    strcpy(str1, "Hello! It is good to see you."); /* necessary for modifying the string */

    char* str2;

    len1 = strlen(str1);

    /* Iterate through all letters of str */
    for ( i = 0; str1[i] != 0; i++ )
    {
        if ( str1[i] == ' ' )
        {
            str1[i] = 0;
        }
    }

    /* Now, iterate through all strings and print them, one-by-one */
```

```
    str2 = str1;
    while ( str2 < str1 + len1 )
    {
        len2 = strlen(str2);
        printf("%s %d\n", str2, len2);

        /* skip to the next string */
        str2 += len2 + 1;
    }

    return 0;
}
```

Q: What happens if there is a double-space after the period?

Now, observe that the following loops are equivalent (but the second is hard to parse!):

```
/* Iterate through all letters of str */
int i;
for ( i = 0; str1[i] != 0; i++ )
{
    if ( str1[i] == ' ' )
    {
        str1[i] = 0;
    }
}
```

```
/* Iterate through all letters of str */
char *p;
for ( p = str1; *p != 0; p++ )
{
    if ( *p == ' ' )
    {
        *p = 0;
    }
}
```

However, we normally want a list of strings given to us! Let's write a method that does that.

```
/**
 * split(s) takes a string s and returns a list of all the "words" in that string.
 *
 * Returns a list of pointers to strings, but only the first should be free'd!
 * At end, the integer pointed at by num_words stores the number of words!
 */
char** split(const char* s, int* num_words)
{
    int i = 0;
    int s_len = 0;
    int list_size = 100;
    char** list = (char**)malloc(list_size * sizeof(char*));

    s_len = strlen(s);

    /* a deep copy of s in to list[0], and we will modify this copy */
    list[0] = (char*)malloc(s_len + 1);
    strcpy( list[0], s );
```

```
    *num_words = 1;

    /* For all positions of the array... */
    for ( i = 0; i < s_len; i++ )
    {
        /* If we find a space... */
        if ( list[0][i] == ' ' )
        {
            /* Then continue until end-of-string or we are done with spaces */
            while ( i < s_len && list[0][i] == ' ' )
            {
                list[0][i] = 0;
                i++;
            }

            if ( i < s_len )
            {
                if ( *num_words >= list_size )
                {
                    list_size += 100;
                    list = (char**)realloc(list, list_size*sizeof(char*));
                }

                list[*num_words] = list[0] + i;
                *num_words = *num_words + 1;
            }
        }
    }

    return list;
}
```

Here is code to use the method:

```
char* s;
int i, num_words;
char** list = split(s, &num_words);

for ( i = 0; i < num_words; i++ )
{
    printf("%s\n", list[i]);
}

/* to deallocate: */
free(list[0]);
free(list);
```

Q: What happens if a string starts with a space?

Exercise: Fix the method to return a list of non-empty words even if the string starts with a space!

15 Lecture 15 : February 19th, 2014

Project announcements, Exams returned.

15.1 Exercise: parens

Task: Create C a program parens that reads lines of text from standard in (use parens.c as a starting point). As part of the code, there should be a method called `parenthesize(char* s)` that first prints the string s (with a newline at the end) and then identifies all substrings inside one level of parentheses. It then calls parenthesize on these substrings.

Example Input:

```
This is just a line.
This is (a bit) more than a line.
Sometimes (such as (now)) there can (may) be multiple (many (many (many))) parentheses.
```

Example Output:

```
This is just a line.
This is (a bit) more than a line.
a bit
Sometimes (such as (now)) there can (may) be multiple (many (many (many))) parentheses.
such as (now)
now
may
many (many (many))
many (many)
many
```

Q: What (should) happen if the closing parentheses do not match the opening parentheses?

Exercise: Create a program to verify that all parentheses, brackets, and braces are properly nested.

16 Lecture 16: February 21st, 2014

Goal for today: Wrap up strange C keywords.

16.1 const

`const` is "constant". Means: never change after first assignment.

```
const int G = 9.81;
```

Doesn't make a lot of sense for global variables. Preprocessor directives would put the constants into the code. The compiler may already do this!

However, for function parameters it DOES make sense!

```
void strcpy( char* to, const char* from );
```

The `const` in the parameter definition is a *promise* that the code will not change the values in that parameter. The compiler checks this, to a point.

```
void test(const int* a)
{
    a[0] = 1; /* gives a compiler error */

    int* b = (int*)a;
    b[3] = 2; /* works just fine */
}
```

There is no true data safety!

16.2 register

The `register` keyword signifies that the given variable will be used a lot, so it may be useful to place that value in a "register" instead of a variable on the memory stack.

```
int doSomething(register int a)
{
    while ( a < (1 << 30) )
    {
        a++;
```

```
    }

    return a;
}
```

The compiler is free to ignore this, or to do this anyway. Probably a useless keyword in modern compilers with optimization flags.

16.3 Optimization Flags

`gcc` has several levels of optimization available, given by adding a flag to the compilation argument (at every stage). Here are some important ones:

- `-Og` – Optimize for the debugger. It will make the instructions fast, but still you will be able to see which values come from what variables in your code.
- `-O1`, `-O2`, `-O3` – Three levels of optimization.
- `-Ofast` – The highest level of "safe" optimization.
- `-O0` – Be sure to do no optimization, even some perhaps default optimization. (Good for `valgrind`)

Do not enable optimizations until after all bugs are fixed! (And don't do it for the project!) Trying to debug optimized code is horrendous. Also, behavior *may* change so **be careful.**

See http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html for the full list.

16.4 static

The `static` keyword means something very different in C than it does in Java. It essentially means: "Remember my value between calls to this function."

Example:

```
#include <stdio.h>

int getNextValue()
{
    static int a = 0;
    a++;
    return a;
}
```

```
int main(void)
{
    int i;
    for ( i = 0; i < 100; i++ )
    {
        printf("next: %d\n", getNextValue() );
    }
    return 0;
}
```

Again: This can lead to some unexpected behavior, so only use this for very special occasions, such as tracking the number of times the function is called (for SCIENCE!).

16.5 extern

Variables and functions can be *defined* exactly once. They can be *declared* multiple times.

For functions, we can declare them using function prototypes. We do this in header files all the time. So, there is no difference between the two lines below:

```
int myFunction(int arg1, char* arg2);
extern int myFunction(int arg1, char* arg2);
```

Well, the difference is subtle and not worth discussing. Essentially, you can have any number of prototypes of either type with no problem.

For variables, this has a very important distinction, particularly with global variables. If two .c files want to interact with a common global variable, then they cannot both define it in their code. The two definitions would be in conflict at the link stage.

Instead, one .c file defines the global variable as normal, but the other .c files must use `extern` to be sure that they do not duplicate the definition. Thus, `extern` allows the *declaration* of a variable without duplicating its *definition*.

See this StackOverflow question for a detailed description of how to organize code to use an extern variable.

Also, recall that the `ttt` project from 01/24/2014 had an `extern` variable in a header file, so all .c files that include that header file have access to the variable.

Warning: It is probably best to NOT use global variables if you can avoid it. Instead, use a structure to store all the "important" data, and pass a pointer to that struct as a parameter to the methods that require it.

16.6 volatile

When using `extern` variables, the compiler does not "see" all of the behavior for that variable all the time. So, this can cause some confusion when performing optimizations. You can alert the compiler that a variable *may change its value* in code that is not in the current file by labeling it with `volatile`.

16.7 union

The `struct` type gives a way of sequentially grouping data types together into a single collection. If you define

```
struct stuff
{
    int a;
    float b;
    double c;
    char* d;
}
```

then a variable of type `struct stuff` corresponds to 4 bytes for `a`, 4 bytes for `b`, 8 bytes for `c`, and 8 bytes for `d`. Each of the data types have specific uses, and this is valuable.

17 Lecture 17: February 24th, 2014

Goals: Discuss self-referential structures, `void*` data type, and function pointers while implementing a doubly-linked list.

17.1 Doubly-Linked List

Q: What is a linked list?

Q: What do you store in a linked list? (Key/Value pairs? For now: assume a string key.)

Let's take our implementation step-by-step. Before implementing our structs, let's just declare a structure and start making method prototypes.

```
typedef struct list_struct linkedlist;
```

Q: What operations would you expect out of a linked list?

- InsertFront or InsertBack
- GetFirst/Front or GetLast/Back, GetItem
- Initialize, Free
- Contains Key?, Contains Value?
- Delete by Key, by Value?

Here are some possible method prototypes:

```
linkedlist* initList();
void freeList(linkedlist* list);
char* getFrontKey(linkedlist* list);
char* getLastKey(linkedlist* list);
char* getKey(linkedlist* list, int i);
void pushFront(linkedlist* list, char* key);
void pushBack(linkedlist* list, char* key);
void popFront(linkedlist* list);
void popBack(linkedlist* list);
int containsKey(linkedlist* list, char* key);
void deleteKey(linkedlist* list, char* key);
void delete(linkedlist* list, int i);
```

Our first necessity is to create the actual nodes of the linked list.

17.2 Self-referential structures

```
/* declaration of the struct and the typedef */
typedef struct node_struct listnode;

/* definition of the struct USING the typedef */
struct node_struct
{
    /* This is the doubly-linked part */
    listnode* next; /* pointers to same type! */
    listnode* prev; /* could also be struct node_struct* */
    char* key;
    void* value; /* discuss in next section */
};

struct list_struct
{
    listnode* front;
    listnode* back;
    int size;
};
```

18 Lecture 18: February 26th, 2014

We continue our discussion of creating a linked list structure in C. What happens if we want key/value pairs? How can we say the value can store "any kind of data"?

18.1 void* data type

If a variable has type `void*`, then that means it contains arbitrary binary data. It should be cast back into that type before use.

For example, the `value` inside the `node_struct` above.

A difficult part about using `void*` is that you need to be "told" how to do basic things with the data: make deep copies, deallocate, or even how to compare them!

Function pointers can allow us to do that:

18.2 Function Pointers

```
returntype (*name) (argtype1, argtype2, ..., argtypek );
```

Example:

```
void printToStdErr(char* err);
void printToStdOut(char* err);
void saveToFile(char* err);

void (*errorLogger) (char*);

int main(int argc, char** argv)
{
    errorLogger = &printToStdErr;


    if ( argc > 1 && strcmp(argv[1], "stdout") == 0 )
    {
        errorLogger = &printToStdOut;
    }

    (*errorLogger)("This is an error!\n");

    return 0;
}
```

Function pointers can be used the same way any type is: as a variable, as a function parameter, as a structure member.

In fact, you can "fake" polymorphism to some extent using function pointers.

```
typedef struct
{
    char* name;
    char* species;
    void (*makeNoise)();
} pet;

pet dog, cat, fox;
dog.makeNoise = &bark;
cat.makeNoise = &meow;
fox.makeNoise = &dingdingdingredingdingding;
```

19 Lecture 19: February 28th, 2014

More uses of Function Pointers:

19.1 Function Pointer Use: Iterators

Suppose we want to iterate along the items in our linked list, and perform some action on the key/value pair for each node.

```
/**
 * iterate : given a list l and a function pointer func,
 * call *func(key, value) for each node in the list, in order.
 */
void iterate(list* l, void (*func)(list*, char*, void*) )
{
    node* cur = l->front;

    while ( cur != 0 )
    {
        *func(l, cur->key, cur->value);

        cur = cur->next;
    }
}
```

This can be used to help with the management of the list.

```
void freeKeyValuePair(list* l, char* key, void* value)
{
    free(key);
    *(l->deallocate)(value);
}

void freeList(list* l)
{
    iterate(l, &freeKeyValuePair);
}
```

19.2 Function Pointer Use: Return List, Item-by-Item

It is possible to return a list of items by using an array, but then the size of the list must be "returned" using a pass-by-reference parameter. This is a clunky way to get a list of things.

Further, perhaps the list is very large and is not good to store in memory all at once! Instead, we can "return" each item as it is found, by using a function pointer.

For instance, perhaps we only want to output the Fibonacci numbers that are prime. We could compute each one individually, but the *cost* of generating them grows with n (unless you use the exponentiation way of computing it).

Instead, we can do the following:

```
void fibonacci(int N, void (*msg)(int,int))
{
    int i = 0;
    int a = 0;
    int b = 1;

    *msg(0, a);
    *msg(1, b);

    for ( i = 2; i <= N; i++ )
    {
        int c = a+b;
        *msg(i, c);

        a = b;
        b = c;
    }
}
```

Then, do the following:

```
void printPrime(int n, int p)
{
    int q;
    int is_prime = 1;
    int maxfactor = (int)sqrt(p);

    for ( q = 2; is_prime && q <= maxfactor; q++ )
    {
        int r = p/q;

        if ( p == (q * r) )
        {
            is_prime = 0;
        }
    }
```

```
    if ( is_prime )
    {
        printf("%d %d\n", n, p);
    }
}

int main(void)
{
    fibonacci(1000, &printPrime);
    return 0;
}
```

19.3 Function Pointer Use: Error Logging

You can make a generic method for sending error messages by using a function pointer. The following function pointer can be used to send a string to the "error log". The actual method it points to can be used to just make a `printf`, or a `fprintf(stderr,...)`, write to a specific log file, or even make a network connection and send a message to another computer!

```
/* global error message function-pointer */
void (*printError)(char*);
```

19.4 Generic Makefiles

And now for something completely different: Advanced Makefiles

Makefiles are much more advanced than you have seen so far (unless you went and found someone else's Makefile).

The most important way to generalize your Makefile is to take common things and turn them into *variables*:

```
VARIABLENAME = Variable Value Until A New Line
ANOTHERVAR = You \
        can \
        also \
        extend \
        variables \
        across \
        lines
```

To use a variable, use `$(VARIABLENAME)`.

The most common use is to take all compiler commands and flags and make them separate.

I have included a magic makefile online that does awesome awesome things!

20 Lecture 20: March 3rd, 2014

C++ was created by Bjarne Stroustrup in the early 80's, intended to be the *next iteration of C*.

20.1 Philosophy of C++

Goal: Retain as much of C as possible, while also adding modern features.

- Efficiency is still key! Don't suffer for features you don't use!
- Strengthen type checking.
- Clean up some syntax.

20.2 Features of C++

Added some features to make programming easier:

- References
- Classes — Basically structs with methods... but also inheritance, polymorphism, etc.
- Templates — Similar to Java Generics
- Exceptions — Basically the same as in Java
- Namespaces — Similar to Java packages.
- Function Overloading — Same name, different action based on parameters.
- Operator Overloading — Make operators do different things based on the surrounding types!
- Object-oriented style I/O — TERRIBLE AND I HATE THIS. But, it's just a library. You can choose to use it or not!

Immediately valuable things:

1. `bool` type, and `true`, `false`.
2. Single-line comments: `// THIS IS A COMMENT UNTIL END OF LINE`
3. Define variables anywhere! `for ( int i = 0; i < n; i++ )`
4. Alternate casting:

```
float f;
int i = (int)f;
i = int(f);
```

5. We can refer to structs by their identifier only:

```
struct foo { stuff };

struct foo old; // old version
foo figher; // valid! Equivalent to previous
```

6. Pointers must be explicitly cast!

```
int* array = (int*)malloc(n * sizeof(int));
```

7. New ways to create arrays, structs, and objects.

```
struct foo { int target; char* hero; };
foo *fighter = new foo(); // create a struct (requires a constructor... more on this later
delete fighter; // delete the struct

int* i = new int[n]; // Instantiate, even if n is a variable
delete[] i; // Delete the array at i
i = 0; // nullify pointer

int** mtx = new int*[n];
for ( int i = 0; i < n; i++ )
{
    mtx[i] = new int[m];
}

for ( int i = 0; i < n; i++ )
{
    delete[] mtx[i];
}
delete[] mtx;
mtx = 0;
```

20.3 Non-Features of C++

- No standard string type (other than `char*`)
- No array bound checking
- No garbage collecting.

20.4 Hello, World!

```
#include <iostream>
```

```
using namespace std;

int main(int argc, char** argv)
{
    cout << "Hello, World!" << endl;

    return 0;
}
```

A warning about `using namespace std` :
http://stackoverflow.com/questions/1452721/why-is-using-namespace-std-considered-bad-practice

21 Lecture 21: March 5th, 2014

21.1 Compiling in C++

Instead of `gcc`, use `g++`.

Everything else is the same!

We still get our typical flags:

- `-c` — stop at compile time and output an object file (*.o)
- `-o file` — Output binary to `file`
- `-g` — Keep debugger flags.
- `-O*` — Optimization flags
- `-I*` — Look for `#include` files in *
- `-D *` — `#define` *
- `-Wall` — Make all warnings possible! (You should use this while practicing!)

More Information about `g++`

21.2 Input/Output with String Streams

There are three main *streams* in `iostream`:

- `cout` — The output stream to `stdout`
- `cin` — The input stream to `stdin`
- `cerr` – The output stream to `stderr`

To output strings, ints, floats, whatever using `cout`, we use the shift operator: <<

Think about the symbols as "arrows" sending data from one place to another.

Technical Bit: The << operator works when the types to the left is `ostream` and *returns* an `ostream`. This allows stringing the operator around:

```
int age = 29;

cout << "You are " << age << " years old!" << endl;
```

This last line is read left-to-right, so we have the proper order of operations is equivalent to:

```
(((cout << "You are ") << age) << " years old!") << endl;
```

Thus, you can see that every << operator has `ostream` to the left of it.

The `endl` variable is a special handler that does several things:
1. adds an endline character,and
2. flushes the stream!

We can also take input from `cin` using the >> operator. (this expects an `istream` to the left, and returns an `istream`)

```
cout << "What is your name?";

char buffer[1024];
cin >> buffer;

cout << "What is your age?";
int age;
cin >> age;

cout << "Hello, " << buffer << ". You are " << age << " years old!" << endl;
```

We can also chain inputs together:

```
cout << "What is your name and age?";

char buffer[1024];
int age;
cin >> buffer >> age;

cout << "Hello, " << buffer << ". You are " << age << " years old!" << endl;
```

WARNING: The line "`cin >> buffer;`" is NOT OVERFLOW SAFE, and also only gets the first "word".

To get a line of text, use:

```
cin.getline(buffer, buflen);
```

To test end of file: use `cin.eof()`, or simply `cin`.

```
while ( cin ) // while ( !cin.eof() )
{
    char c;
    cin >> c;
    cout << c;
}
```

PROBLEM: `cin >> c;` reads the next *non-whitespace* character!

Fix:

```
while ( cin ) // while ( !cin.eof() )
{
    char c;
    c = cin.get();
    cout << c;
}
```

21.3 About `using namespace std`

The variables `cout`, `cin`, `cerr`, and `endl` are all contained in the `std` namespace.

If you do not use `using namespace std`, then you can refer to these variables using: `std::cout`, `std::cin`, `std::cerr`, `std::endl`.

This is a bit unwieldy, but it gets the job done.

21.4 Pass by Reference

In C++, we get a handy way of passing values "by reference". This means that the variables will not just be copies of the value, but if you change the variable in the function, the original function will also be updated!

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

void swap(int& x, int& y)
{
    int t = x;
    x = y;
    y = t;
}

void sort(int* list, int size)
{
    for ( int i = 0; i < size; i++ )
    {
        for ( j = i; j < size; j++ )
        {
            if ( list[i] > list[j] )
            {
                swap(list[i], list[j]);
            }
        }
    }
}

int main(int argc, char** argv)
{
    int size = atoi(argv[1]);
    int* list = (int*)malloc(size*sizeof(int));
    int range = size*size;
    srand(time(NULL)); // Initialize the randomness with a seed
    printf("Original List: ");
    for ( int i = 0; i < size; i++ )
    {
        list[i] = (int)( rand() % range );
        printf("%d ", list[i]);
    }
    printf("\nSorting...\n");
    fflush(stdout);
    sort(list, size);
    printf("Sorted List: ");
    for ( int i = 0; i < size; i++ )
    {
        list[i] = (int)( rand() % range );
        printf("%d ", list[i]);
    }
    printf("\n");
    free(list);
    return 0;
}
```

Observe that we don't need the `-lm` to link the math library when using `g++`.

To do the same kind of `swap` method in C, we would need to use pointers:

```
void swapInC(int* x, int* y)
{
    int t = *x;
    *x = *y;
    *y = t;
}

void sort(int* list, int size)
{
    for ( int i = 0; i < size; i++ )
    {
        for ( j = i; j < size; j++ )
        {
            if ( list[i] > list[j] )
            {
                swap( &(list[i]), &(list[j]));
            }
        }
    }
}
```

Big differences:

1. Pointers can be NULL, References can't.
2. Pointers must be dereferenced "manually"
3. Pointers may be reassigned
4. Pointer arithmetic :(

We can have references in `structs` and as function return types. THIS IS COMPLICATED. Maybe we will get to it later.

A reference is just a type: Use `type&` to have a reference to a variable of type "`type`"

We can have references of pointers!

```
void resizeAndZeroArray(int* &A, int n)
{
    if ( A == 0 )
    {
        A = (int*)malloc(n * sizeof(int));
```

```
    }
    else
    {
        A = (int*)realloc(A, n* sizeof(int));
    }
    bzero(A, n * sizeof(int));
}
```

However, we cannot have pointers to references (`int& * ref`) or references of references (`int&& ref`). That is, *references are top-level-only modifiers of types.*

QUESTION: What does the following code do?

```
int i = 3;
int j = 4;
int& r = i;
int* p = &r;
*p = j;
```

22 Lecture 22: March 7th, 2014

Today's topic: C++ CLASSES!

Basic Syntax:

```
class ClassName
{
private:
    // Place private members and methods here

protected:
    // Place protected members and methods here

public:
    // Place public members and methods here

}; // This semicolon is important!
```

Best Practice: Create a `.hpp` file containing the class definition, including prototypes for methods inside the class. Create a `.cpp` file containing the implementations of the class methods.

22.1 Example: A String Class

```
/**
 * The String class is intended to contain a string, and act very similar to a C-string,
 * except we will check for array bounds and things you would expect out of a
 * modern String object.
 */
class String
{
protected:
    char* str;
    int length;
    int size;

    void resize(int newSize);

public:
    String(); // Blank Constructor
    String(char* s); // Constructor, given an existing C-string.

    virtual ~String(); // Destructor

    char get(int i);
```

```
    const char* getCString(); // Get a string that should NOT be modified outside!

    void append(char* s); // Add a new string

    int indexOf(char c); // Get the first position of a character
    int indexOf(char c, int start); // Get the first position of a character, starting at some
    int indexOf(char* s); // Get the first position of a substring


}; // don't forget!
```

23 Lecture 23: March 10th, 2014

Today: How to use classes in regular code!

Things to fix from last time: Implement getLength, Realloc for concatenate,

```
#include <stdio.h>
#include "String.hpp"

int main(int argc, char** argv)
{
    String* s = new String();

    for ( int i = 0; i < argc; i++ )
    {
        s->concatenate(argv[i]);
        if ( i < argc - 1)
        {
            s->concatenate(" ");
        }
    }

    printf("%s\n", s->getCString() );

    delete s;

    return 0;
}
```

Also, ask what happens with the following lines of code:

```
String* s = new String(argv[0]);
String t;
String q("Hello!");

String a = *s;
String* r = &t;
```

For the line `String a = *s;`, we need a *copy constructor*:

```
String::String(String& s)
{
    /* The string s is pass-by-reference, so no copy is done YET */
    /* Now: Make a deep copy of s! */
}
```

Best Practice: When having multiple constructors, it may be helpful to have a common `init` method that sets up the basic values. Then, each constructor only needs to call the init method with the appropriate input.

Question: What happens when we pass-by-value but don't have a copy constructor?

23.1 Operator Overloading

Here are some interesting reads about operator overloading:

http://courses.cms.caltech.edu/cs11/material/cpp/donnie/cpp-ops.html

http://stackoverflow.com/questions/4421706/operator-overloading

There is a difference between a *member function* and a *non-member function*. The brackets `[ ]` *must* be implemented as a member function, because of the way things are organized.

```
#include <stdio.h>
#include "String.hpp"

int main(int argc, char** argv)
{
    String* s = new String(argv[0]);

    for ( int i = 1; i < argc; i++ )
    {
        s->concatenate(" ");
        s->concatenate(argv[i]);
    }

    printf("%s\n", s->getCString());

    for ( int i = 0; i < s->getLength(); i++ )
    {
        printf("%c", (*s)[i]);
    }
    printf("\n");

    for ( int i = 0; i < s->getLength(); i++ )
    {
        printf("%c", s->operator[](i));
    }
    printf("\n");


    delete s;

    return 0;
}
```

24 Lecture 24: March 24th, 2014

Project 2 is assigned.

Today, we discuss some **math** that will help!

24.1 Graph Theory

A *graph* is given by a *vertex set* V (which contains elements called *vertices*; each one is a *vertex*) and an *edge set* E (which contains elements called *edges*) and each edge is a pair of vertices. Graphs can be drawn as dots and lines: A dot for every vertex, and a line between dots for each edge.

Graphs can model relationships: Consider people on Facebook to be vertices, and place an edge between two people if they are friends. *Friendship* is either all-or-nothing. There are no one-directional friendships on Facebook.

However, on Twitter there is a form of directional friendship: *following*. Let each Twitter profile be a vertex, and place a *directed edge* from one profile to another if the first "follows" the second. This can be drawn as an arrow from the first to the second.

A *directed graph* is given by a *vertex set* V and an *edge set* E where each edge is an *ordered* pair.

We can represent a graph using a very simple data structure, called an *adjacency list*.

For each vertex v, we store a list $N^+(v)$ called the *out-neighborhood* of v. $N^+(v)$ stores a list of the vertices u such that $v \to u$ is a directed edge. The *out-degree* of v is $d^+(v)$, which is the size of $N^+(v)$.

We will be thinking of a directed graph where the vertices are the positions of the game board and the directed edges are the possible moves between positions. So, given a vertex (x, y) the out-neighborhood $N^+((x, y))$ contains the possible moves that an actor at that position could use.

24.2 The `GraphMap` class

In the Project 2 Codebase, there is a class called `GraphMap`. It stores the 2D map of the game as a graph. Thus, there are two coordinate systems: One is the standard (x, y) position of a vertex. The second is the *index* of a vertex in a list.

So, each position is identified by a triple, $(i; x, y)$. The index i ranges from 0 to `getNumVertices()`-1. The position (x, y) has $0 \leq x < w$ and $0 \leq y < h$, where w and h are the width and height of the board (which you can get by using the `getWidth()` and `getHeight()` methods).

To translate between the index and the position, use these methods on the `GraphMap`:

1. `getVertex(int x, int y)` — Given (x, y), it returns the index i.

2. `getPosition(int v, int& x, int& y)` — Placing the index i in the first parameter, the variables given for x and y will be set to the corresponding (x, y) position. (Recall *pass-by-reference*.)

24.3 Path Finding

When you implement the `selectNeighbor` method, you are given a pointer to the current `GraphMap` and your current (x, y) position.

Your goal is to identify which of the out-neighbors of the current position you should move to.

1. `int getNumNeighbors( int x, int y )` — Given an (x, y) position, determine the out-degree of that vertex.
2. `void getNeighbor( int x, int y, int i, int& a, int& b )` — Given an (x, y) position and a value i between 0 and $d^+((x, y)) - 1$, place the ith neighbor of (x, y) into (a, b).

In this way, you can "navigate" the graph.

A *path* is a sequence of vertices $v_1, \ldots, v_k$ such that $v_i v_{i+1}$ is a (directed) edge for each i where $0 \leq i < k$. We discuss an algorithm to find a path from v to u in a directed graph. It is given in detail in Algorithm 1 on Page 74.

The basic idea is this: Keep a list of where you have been (this is what S is doing), how you got there (this is what p is doing), and which vertices should be "expanded" (this is what Q is doing). By using a queue, we are visiting the vertices that are closest to v *first*, and so we will always be traveling along shortest paths. This will not find *all* shortest paths!

24.3.1 Suggested Modifications

Here are a few ways you should consider about modifying the BFS algorithm to suit your needs:

1. Instead of returning a length, make it return the *first vertex*, helping you select your move.
2. Use BFS to test which actors have paths to other actors. This will help you select which actor to visit first. (See the map `maps/stronglyconnected.txt` for an example of why this is important. It is only important for Hard cases.)
3. Use BFS to test which positions your enemies can visit in the next few moves.
4. Modify BFS to avoid certain positions (such as the ones you identified in the previous step) while finding a path.

Algorithm 1 Breadth-First Search – Given a directed graph and vertices v, u, return the length of a shortest path from v to u. Returns ∞ if no path exists.

```
// Create data structures
n ← |V|
S ← new int[n]
p ← new int[n]
Q ← new Queue();

// Initialize data structures
p[v] ← v
S[v] ← 1
Q.push_front(v)

// Perform BFS Loop
while Q.size() > 0 do
  x ← Q.front()
  Q.pop_front()
  for all y ∈ N+(x) do
    if S[y] == 0 then
      S[y] ← 1
      p[y] ← x
      Q.push_front(y)
    end if
  end for
end while

// Determine the result
if S[u] == 0 then
  return ∞
else
  count ← 0
  x ← u
  while x ≠ v do
    x ← p[x]
    count ← count +1
  end while
  return count
end if
```

24.4 Other Actors

You can also determine the position and types of the other actors using the following methods in the `GraphMap`:

```
int getNumActors();
int getActorType( int i );
void getActorPosition( int i, int\& x, int\& y );
```

25 Lecture 25: March 26th, 2014

Today, we talk about inheritance, polymorphism, and other related things using C++ classes.

25.1 Inheritance

Example: Cats and Dogs are both Pets. All pets have common names, but they have different behavior!

```
class Pet
{
    protected:
         const char* name;

    public:
        Pet(const char* name);
        virtual ~Pet();

        const char* getName();

        virtual const char* speak();
};

class Dog : public Pet
{
    public:
        Dog(const char* name);
        virtual ~Dog();

        virtual const char* speak();
};

class Cat : public Pet
{
    protected:
        int num_lives;

    public:
        Cat(const char* name);
        virtual ~Cat();

        virtual const char* speak();

        int getNumLivesLeft();
};
```

Here is how the behavior is defined:

```
const char* Pet::speak()
{
    // Say nothing
    return "";
}

const char* Dog::speak()
{
    return "bark";
}

const char* Cat::speak()
{
    return "Death to all humans!";
}
```

You can now define different objects in the following way:

```
Pet* p = new Pet("Iggy the Iguana");
Dog* d = new Dog("Cici");
Cat* c = new Cat("Scratch Fury, Destroyer of Worlds");

printf("%s says, \"%s\"\n", p->getName(), p->speak() );
printf("%s says, \"%s\"\n", d->getName(), d->speak() );
printf("%s says, \"%s\"\n", c->getName(), c->speak() );
```

25.2 Polymorphism

When using pointers, we don't need to know the actual type, just what type we actually care about!

```
Pet* p = new Pet("Iggy the Iguana");
Pet* d = new Dog("Cici");
Pet* c = new Cat("Scratch Fury, Destroyer of Worlds");

printf("%s says, \"%s\"\n", p->getName(), p->speak() );
printf("%s says, \"%s\"\n", d->getName(), d->speak() );
printf("%s says, \"%s\"\n", c->getName(), c->speak() );
```

Q: What happens if we make the `speak` method non-virtual?

25.3 Multiple Inheritance

In C++ you can inherit from multiple types!

```
class CatDog : public Dog , Cat
{
    public:
        CatDog(const char* name);
        virtual ~CatDog();
}
```

Q: What happens when we call `speak` on a `CatDog`? Does it matter what order we inherit?

WARNING: Multiple inheritance can act very strangely! Do so at your own risk!

26 Lecture 26: March 31, 2014

(Friday was a group discussion about the project while handing out exams.)

We discussed (briefly) the Standard Template Library, including `stack`, `queue`, `pair`, and `string`. We did not get to all of the details, but you can find ALL of the important information in your STL textbook or at the following web pages:

http://www.cplusplus.com/reference/stl/

http://www.cplusplus.com/reference/stack/stack/

http://www.cplusplus.com/reference/queue/queue/

http://www.cplusplus.com/reference/queue/priority_queue/

http://www.cplusplus.com/reference/list/list/

http://www.cplusplus.com/reference/vector/vector/

http://www.cplusplus.com/reference/utility/

http://www.cplusplus.com/reference/utility/pair/

http://www.cplusplus.com/reference/string/

27 Lecture 27: April 2, 2014

FIRST: Apologize publicly to Zach for pointing out a STUPID issue with templates inside of templates in C++ (but not C++11).

To compile with C++11, use `-std-c++11`

See http://en.wikipedia.org/wiki/C++11 for more information.

Today, we will discuss how to mimic some Java behavior, such as abstract classes and interfaces.

27.1 Interfaces

In Java, an *interface* is a collection of member functions that must be implemented by any class that implements that interface. There are some good features here:

1. Polymorphism works, so variables can store pointers to instances of objects that implement those methods.
2. One class can implement multiple interfaces.
3. There are no assumptions to how the methods are implemented.
4. No common member variables!

Interfaces CANNOT BE FAKED using C++ inheritance, because despite multiple inheritance, it will NOT work to call methods on pointers to the not-first base!

Example.

Let's create two interfaces: `Comparable` and `Printable`.

The `Comparable` interface allows for a way to compare two objects of the same type using a `compare` method.

```
class Comparable
{
public:
    int compareTo(Comparable* c1);
};
```

The `Printable` interface allows a way to print the data of the class using a `print` method.

```
class Printable
{
```

```
public:
    void print();
};
```

Now, create a data type (say `ComplexData`) that inherits both of these, then you have a problem!

BUT NOW, if you call `print()` on a pointer of type `Printable*`, you will actually call the `compareTo()` method with a null parameter!

THIS IS NOT HOW IT WORKS IN JAVA.

What is going on?

Well, in C++, each class is essentially a struct with some function pointers.

The virtual function pointers can be overwritten by subclasses, but the *positions* of the function pointers in the struct remains fixed!

So, when you inherit, the superclasses are just concatenated in the subclass's struct, and the new stuff is added afterward.

There is no "dynamic typing" that allows us to figure out *where* the method `print()` lives in the subclass if the `Printable` interface appears after the `Comparable` interface.

MAJOR CORRECTION!!!

A few students who know what's going on pointed out that there are ways of doing this! It just requires using some complicated casting keywords. You can read about them here:

http://www.cplusplus.com/doc/tutorial/typecasting/

The code on the web site will be updated accordingly.

27.2 Abstract Classes

In Java, an *abstract class* is a class that cannot be instantiated on its own, but must be extended in order to be instantiated. There are a few good features.

1. *Requires* "pure virtual" methods to be implemented before instantiation.
2. Allows for inherited member variables.
3. Allows for inherited behavior with member functions.
4. Allows for inherited destructor for member variables.

I almost made `Actor` be an abstract class for Project 2, but decided it would be helpful to instantiate an `Actor` that did not move.

Example.

```
class AbstractIntegerContainer
{
private:
    int size_array;
    int* array;
    int num_accesses;

protected:
    int get(int i);
    vlid set(int i, int value);

public:
    AbstractIntegerContainer(int N);
    virtual ~AbstractIntegerContainer();

    int size();
    int getNumAccesses();

    // TODO: Implement these methods to work as a stack or a queue, your pick!
    virtual void push(int i) = 0;
    virtual int peek() = 0;
};
```

We can extend this abstract class into a Stack or a Queue, and make the push and peek methods work as expected!

27.3 Const Functions

We have discussed how we can use `const` to return data that should not be modified. However, if we apply standard OO principles to our classes, so all members must be accessed through a method, then we cannot call those methods!

We can modify our `AbstractIntegerContainer` to have some `const` methods, including: `get`, `size`, `getNumAccesses`, `peek`.

For more info, see http://stackoverflow.com/questions/3141087/what-is-meant-with-const-at-end-of-f

HOWEVER, whenever we call the method `get`, we ARE changing something! We are changing `num_accesses`! This is not terribly important to the behavior of the container, but instead is important to some statistics tracking, perhaps.

To allow these methods to still be `const`, but to modify this variable, we can claim `num_accesses` to be `mutable`. This allows the variable to change, even inside `const` functions.

28 Lecture 28: April 4, 2014

FIRST: When having a template inside of a template, you must separate the two > symbols:

```
std::stack<std::pair<int,int> > pairstack;
```

SECOND: A few students who know what's going on pointed out that there are ways of doing this! It just requires using some complicated casting keywords. You can read about them here:

http://www.cplusplus.com/doc/tutorial/typecasting/

The code on the web site will be updated accordingly.

28.1 The Standard Template Library

I will not lecture on this. You have a book! Use it.

I will not make exam questions on how to use anything but `std::stack` and `std::queue`.

28.2 Templates

We have seen how to use existing templates. Let's make our own template!

One important thing: Templates are NOT implementations! They must be defined entirely in a header file!

```
// template <typename Type>
template <class Type>
class AbstractArray
{
private:
    int array_size;
    Type* array;

protected:
    Type get(int i)
    {
        if ( i < 0 || i >= this->size() )
        {
            // Q: what happens if this cannot cast to Type?
            return 0;
        }
        return this->array[i];
    }
```

```
    void set(int i, Type value)
    {
        if ( i >= this->array_size )
        {
            this->array_size = i + 100;
this->array = (Type*)realloc(this->array, this->array_size * sizeof(Type));
        }

        this->array[i] = value;
    }

public:
    AbstractArray(int N)
    {
        this->array_size = N;
        this->array = (Type*) malloc(this->array_size * sizeof(Type));
    }

    virtual ~AbstractArray()
    {
        free(this->array);
        this->array = 0;
    }

    virtual int size()
    {
        return this->array_size;
    }

    virtual Type peek() = 0;
    virtual void push(Type value) = 0;
    virtual void pop() = 0;
};
```

We can also use inheritance with our template classes:

```
template <class Type>
class Stack : public AbstractArray<Type>
{
private:
    int top;

public:
    Stack(int N) : AbstractArray<Type>(N)
    {
```

```
        this->top = 0;
    }

    virtual ~Stack()
    {
    }

    virtual int size()
    {
        return this->top;
    }

    virtual Type peek()
    {
        return this->get(this->top - 1);
    }

    virtual void push(Type value)
    {
        this->set(this->top, value);
        (this->top)++;
    }

    virtual void pop()
    {
        if ( this->top > 0 )
        {
            (this->top)--;
        }
    }
};
```

Then, to use it, we can create a main method.

```
#include <stdio.h>
#include "Stack.hpp"


int main(void)
{
    AbstractArray<char>* aia = new Stack<char>(100);

    aia->push('H');
    aia->push('i');
    aia->push('!');
    aia->pop();
    aia->push('?');
```

```
    aia->print();

    delete aia;
    return 0;
}
```

28.3 Specialization

Once we have a generic template, we can also create specific implementations for specific data types!

```
template <>
class AbstractArray<char>
{
private:
    int array_size;
    char* array;

protected:
    char get(int i )
    {
        return this->array[i];
    }

    void set(int i, char c)
    {
        this->array[i] = c;
    }

public:
    AbstractArray(int N)
    {
        this->array_size = N;
        this->array = (char*)malloc(N);
    }

    virtual ~AbstractArray()
    {
        free(this->array);
        this->array = 0;
    }

    virtual int size()
    {
        return this->array_size;
```

```
    }

    virtual char peek() = 0;
    virtual void push(char c) = 0;
    virtual void pop() = 0;


    void print()
    {
        for ( int i = 0; i < this->size(); i++ )
        {
            printf("%c", this->get(i));
        }
        printf("\n");
    }

};
```

29 Lecture 29: April 4, 2014

Announce the CS Retention Survey.

Today, we discuss templates some more, except we will use template *functions*.

Our goal: Use templates instead of interfaces in order to have specific implementations for comparing and sorting.

```
template <class T>
int compare(T& c1, T& c2)
{
    if ( c1 < c2 )
    {
        return -1;
    }
    else if ( c2 < c1 )
    {
        return 1;
    }

    return 0;
}

// A specialization to int
template <>
int compare<int>(int& c1, int& c2)
{
    return c1-c2;
}

template <class T>
void testAndSwap(T& a, T& b)
{
    if ( compare(a,b) < 0 )
    {
        T t = a;
        a = b;
        b = t;
    }
}

template <class T>
void sort(T* data, int length)
{
    for ( int i = 0; i < length; i++ )
    {
        for ( int j = i+1; j < length; j++ )
        {
            testAndSwap( data[i], data[j] );
        }
    }
```

```
}


#include <stdio.h>
#include <stdlib.h>
#include "sort.hpp"

int main(void)
{
    printf("Enter a list of numbers. Make one negative to end the list.\n");

    int cur_num = 0;
    int buflen = 100;
    int* buffer = (int*)malloc(buflen* sizeof(int));

    while ( cur_num < buflen )
    {
        scanf("%d", &(buffer[cur_num]));

        if( buffer[cur_num] < 0 )
        {
            break;
        }
        cur_num++;
    }

    sort<int>(buffer,cur_num);

    printf("Sorted:");
    for ( int i = 0; i < cur_num; i++ )
    {
        printf("%d ", buffer[i]);
    }

    free(buffer);
    return 0;
}
```

29.1 Specialization

We can again specialize our implementation. If we are sorting `char`s, then there are very few bits to consider! This means that radix sort is likely the fastest!

See http://en.wikipedia.org/wiki/Radix_sort for more information.

```
// in the case of char, we can do radix sort!
template <>
void sort<char>(char* data, int length)
{
    char* temp = (char*)malloc(length);
```

```
    int max = 0;
    for ( int i = 0; i < length; i++ )
    {
        if ( data[i] > max )
        {
            max = data[i];
        }
    }

    int bit = 1;

    while ( bit <= max && bit < 256 )
    {
        int num_zero_bits = 0;

        for ( int i = 0; i < length; i++ )
        {
            if ( (data[i] & bit) == 0 )
            {
                num_zero_bits++;
            }
        }

        int cur_zero = num_zero_bits - 1;
        int cur_one = length - 1;

        for ( int i = length - 1; i >= 0; i-- )
        {
            if ( (data[i] & bit) == 0 )
            {
                temp[cur_zero] = data[i];
                cur_zero--;
            }
            else
            {
                temp[cur_one] = data[i];
                cur_one--;
            }
        }

        for ( int i = 0; i < length; i++ )
        {
            data[i] = temp[i];
        }


        bit = bit << 1;
    }

    free(temp);
}


#include <string.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include "sort.hpp"


int main(int argc, char** argv)
{
    size_t buflen = 1024;
    char* buffer = (char*)malloc(buflen);

    getline(&buffer, &buflen, stdin);

    printf("Original:%s\n", buffer);
    sort<char>( buffer, strlen(buffer) - 1 );
    printf("Sorted:%s\n", buffer);

    free(buffer);
    return 0;
}
```

30 Lecture 30: April 7, 2014

Today, we discuss template functions and how to specialize them. In particular, we discuss a "comparison" function and a sorting function.

We first create a template function `compare` that can compare any two objects. Using the greater-than operator (we could have overloaded the operator) we can test these things.

```
#include <stdlib.h>

template <class T>
int compare(T& c1, T& c2)
{
    if ( c1 > c2 )
    {
        return 1;
    }
    if ( c2 > c1 )
    {
        return -1;
    }

    return 0;
}
```

However, for integers, we can simply subtract to get the same behavior. Thus, we specialize the template.

```
template <>
int compare<int>(int& c1, int& c2)
{
    return c1 - c2;
}
```

We can now use both instances of compare to test if two values should be swapped, and then swap them!

```
template <class T>
void testAndSwap(T& a, T& b)
{
    if ( compare<T>(a,b) > 0 )
    {
        T t = a;
        a = b;
        b = t;
    }
}
```

This enables a very simple sorting algorithm to work in general!

```
template <class T>
void sort(T* data, int length)
{
    for ( int i = 0; i < length; i++ )
    {
        for ( int j = i+1; j < length; j++ )
        {
            testAndSwap(data[i], data[j]);
        }
    }
}
```

However, we can specialize to different types! For instance, when dealing with type `char`, there are very few bits and we can use Radix sort. Below is an implementation based on the general algorithm at http://en.wikipedia.org/wiki/Radix_sort.

```
template <>
void sort<char>(char* data, int length)
{
    char* temp = (char*)malloc(length);

    int max = 0;
    for ( int i = 0; i < length; i++ )
    {
        if ( data[i] > max)
        {
            max = data[i];
        }
    }

    int bit = 1;

    while ( bit < max )
    {
        int num_zeroes = 0;

        for ( int i = 0; i < length; i++ )
        {
            if ( (data[i] & bit) == 0 )
            {
                num_zeroes++;
            }
        }

        int cur_zero = num_zeroes - 1;
        int cur_one = length - 1;

        for ( int i = length - 1; i>= 0; i-- )
        {
            if ( (data[i] & bit) == 0 )
            {
                temp[ cur_zero ] = data[i];
```

```
                cur_zero--;
            }
            else
            {
                temp[ cur_one ] = data[i];
                cur_one--;
            }
        }

        for ( int i = 0; i < length; i++ )
        {
            data[i] = temp[i];
        }

        bit <<= 1;
    }

    free(temp); temp = 0;
}
```

We can now use this common file to sort `int`s and `char`s:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sort.hpp"

int main(void)
{
    int buflen = 1000;
    int* buffer = (int*)malloc(buflen*sizeof(int));

    int count = 0;

    printf("Enter a list of numbers, ended with a negative number:\n");

    while ( count < buflen )
    {
        scanf("%d", &(buffer[count]));

        if ( buffer[count] < 0 )
        {
            break;
        }

        count++;
    }

    sort<int>(buffer, count);
```

```
    printf("Sorted:");

    for ( int i = 0; i < count; i++ )
    {
        printf("%d ", buffer[i]);
    }
    printf("\n");


    free(buffer); buffer = 0;


    return 0;
}


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sort.hpp"

int main(void)
{
    size_t buflen = 1000;
    char* buffer = (char*)malloc(buflen);

    getline(&buffer, &buflen, stdin);

    printf("Original:%s\n", buffer);


    sort<char>(buffer, strlen(buffer)-1);

    printf("Sorted:%s\n", buffer);

    free(buffer);
    return 0;
}
```

30.1 Preprocessor Macros

We had a brief discussion of using `#define` to define functional macros. We created the following macros:

```
#define CHECK_AND_FREE(A)  \
```

```
    if ( A != 0 ) \
    { \
        free(A);\
        A = 0;\
    }

#define CHECK_DELETE_AND_FREE(A, length ) \
    if ( A != 0 )\
    {\
        for ( int i = 0; i < length; i++ )\
        {\
            if ( A[i] != 0 ) \
            {\
                delete A[i];\
            }\
        }\
        delete[] A;\
        A = 0;\
    }
```

and we used them as follows:

```
#include "macros.hpp"

#include <stdlib.h>

class Bogus
{
public:
    Bogus(){}
    ~Bogus(){}
int i;
};


int main(void)
{
    int l = 100;
    Bogus** b = (Bogus**)malloc(l * sizeof(Bogus*));
    for ( int i = 0; i < l; i++ )
    {
        b[i] = new Bogus();
    }

    CHECK_DELETE_AND_FREE(b, l);

    return 0;
```

```
}
```

31 Lecture 31: April 9, 2014

Today was a lab day for Project 2A.

32 Lecture 32: April 11, 2014

Today, we discuss exceptions and the `try` and `catch` statements.

33 Lecture 33: April 14, 2014

Today, we have a post-mortem for Project 2A, which can hopefully help students plan for Project 2B.

33.1 Design Time

An hour spent planning and designing can save 10 in wasted effort!

Top-Down Approach:

1. What is the "Big Picture" problem to solve?
2. What is the strategy?
3. What information do I need to collect in order to execute that strategy?
4. Where does that information live? How can I access it?
5. Can I make a common-format question that can be asked in different ways?
6. Can I translate that question into a helper method?

33.2 Iterative Development

How can we take small steps between runnable executables?

Backup early and often! You can use Blackboard for this!

33.3 Testing

How can we test our project?

33.4 Optimization

"Premature optimization is the root of all evil." – Donald Knuth

Don't optimize anything until you have something that is correct! Only optimize things that need to be optimized!

Ways to optimize:

1. Make use of previously-computed data.

2. Track permanent vs. non-permanent data.

3. Don't try to solve the big problem, but solve little problems!

33.5 Knowing When To Stop

Think of the Law of Diminishing Returns.

34 Lecture 33: April 16, 2014

34.1 Namespaces

Why use namespaces?

How to use namespaces?

What does `using namespace std;` really do?

34.2 Nested Classes

What is a nested class?

Why use a nested class?

Example: Node in a Linked List.

Warning: Nested classes behave very simi

35 Lecture 34: April 21, 2014

(April 18th was Exam 3)

36 Lecture 35: April 21, 2014

Snapshot of the Future: Design Patterns. Object-Oriented Programming is everywhere, and there are a few important patterns that will appear in many different places. We'll discuss a few of these. (This is an important topic if you are ever in a technical interview for a job involving programming!) Hints towards advanced courses in Software Engineering.

37 Lecture 36: April 23, 2014

Snapshot of the Future: Algorithms and Complexity. We talked about BFS in this class, and there are lots of other algorithms for finding shortest paths and distances. How can we compare these things? I'll briefly discuss things like running time, Big-Oh Notation, and harder problems such as the Traveling Salesman Problem. Hints towards advanced courses in Discrete Mathematics, Algorithms, and Theory of Computing.

38 Lecture 37: April 25, 2014

Snapshot of the Future: Applications of Reachability. I mentioned that solving the question "Can I get from u to v in a (directed) graph G?" has many applications. In this lecture, I'll talk about some, including the farmer crossing river puzzle, solving Rubik's cubes, and the 2048 game. Hints towards advanced courses in Artificial Intelligence, Algorithms, and Graph Theory.

38.1 Farmer Puzzle

http://www.mathsisfun.com/puzzles/farmer-crosses-river-solution.html

http://mathforum.org/library/drmath/view/57963.html

http://en.wikipedia.org/wiki/A*_search_algorithm

38.2 Rubik's Cube

http://en.wikipedia.org/wiki/Rubik's_Cube

http://en.wikipedia.org/wiki/God%27s_algorithm

http://www.wikihow.com/Solve-a-Rubik's-Cube-(Easy-Move-Notation)

http://ruwix.com/online-rubiks-cube-solver-program/

38.3 2048

http://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048/22389702#22389702

http://www.flyingmachinestudios.com/programming/minimax/

http://en.wikipedia.org/wiki/Minimax#Minimax_algorithm_with_alternate_moves

http://en.wikipedia.org/wiki/Expectiminimax_tree

http://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

39 Lecture 38: April 28, 2014

Lab Day. Work on Project 2B.

Similar to our Project 2A lab day, here you can get help from me and some TAs on your project when the deadline is looming!

40 Lecture 39: April 30, 2014

Project 2B Due.

Snapshot of the Future: The Basics of Testing. So far, we have used "test cases" as a way to test our full programs. However, it would be helpful to test smaller parts of our programs, so we can be VERY SURE that our code works correctly. We will discuss the basics of unit testing. A friend (who regularly hires programmers) says that the MOST IMPORTANT thing that undergrads should know before their first job is "How to write a unit test." Hints towards advanced courses in Software Engineering, Software Testing, and building quality software in general.

41 Lecture 40: May 2, 2014

Project 2B Post-Mortem. Let's discuss what went well, what did not, what strategies you used, and perhaps watch some student-generated actors compete!