CS3157: Advanced Programming

Lecture #1
May 21
Shlomo Hershkop
shlomo@cs.columbia.edu
Who am I

- Shlomo Hershkop
  - About me
  - Aim: Prof Hershkop
  - My Research

- Actually also a unix command 😊
It summer Session!

- Welcome

- Ask yourself, is it better to spend the summer outside or inside on this stuff?!

- Hope to be very informal

- I hope to convince you this is more fun 😊
Overview

- **Today:**
  - Basic overview of the course and objectives
  - Basic C
  - Setting up environment
  - Intro CGI
  - Basic Shell Programming

- **Goal:**
  - Things are much easier if everyone knows why they are here, and what we are trying to accomplish.
  - This is going to be a very interactive course.
  - We will learn about programming ideas while trying to have fun.
What?

- CS3157: Third course for CS majors.
  - co-req to Data Structures (switched to java)

- Prerequisites:
  - Intermediate knowledge in Programming
  - Object Oriented Programming:
    - What, why, how, and when.
  - Program Designs.
    - Not enough to know how to write the program, need to know how to do it correctly.
    - Need to learn tool of the trade
So what are we going to be doing?

- Cover practical languages:
  - C
  - C++

- Cover practical skills:
  - Debugging
  - Environmental setup
  - CGI/Web based programming
  - Regular expressions
  - Web scripting
  - Plus more!
Point

- Programming is not Java!
- Computer science != programming

- Give you a feel of the real world:
  1. Problem description blurry
  2. Many choices for programmer
  3. Will learn best tools to stay lazy 😊
Basics

- Instructor: Professor Shlomo Hershkop
  (shlomo@cs.columbia.edu)

- Class website:
  - cs.columbia.edu/~sh553/teaching/su07-3157/
  - Check it regularly (at least twice a week).
    - See announcement sections for update info.

- Meet twice a week: 825 Mudd
Resources

- TA's:
  - Zhiyang Cao
  - aim: caozhiyang1980

- Please take advantage of the webboard...for general qs
  - Good:
    - How do I check what version of java is running?
    - Hints for problem X
    - What does Error ?@?!@@ mean?
  - Bad: What is wrong with the following code:
    - public class foo()
  - These kind of questions email privately to TA or Instructor
Requirements

- Interest to learn about Computer Science
- Learn to use cool tools
- Learn to make your own tools
Textbook

- Textbook can be acquired online or at the Columbia Bookstore.
  - Else: borrow, threaten, or ‘acquire’ a book…I don’t ask questions

- C/C++
  - C how to program
  - Dietel and Dietel

- Favorite search engine….
Reading

- I will be posting reading on the website and in class notes

- Please try to keep up with the reading
  - I will try to make up examples for class, but there are random stuff which the book covers which is good to see in print
    - Feel free to ask questions from the book
Course Structure

- 6 Labs – 120 points
  - Out Mondays, Due by class time following week
- 2 Homeworks – 60 points
  - Will have about 2 weeks per homework
- Final online (60 points)
  - open book, test your understanding
- Homework/Lab is very important:
  - Firm believer in hands on learning
  - Start early
  - Come to office hours, and ask questions
    - We are here for YOU!
Class participation and Attendance

- Attendance and participation is expected
  - Very interactive lectures & Labs
  - Small class, means more help
  - Class anonymous feedback system

- If you have to miss class, I expect you to catch up.
  - It’s a short semester, so bear with me
  - There will be class notes posted to the website
  - There will be many examples in class only, so make sure to get someone’s notes.
Makeup classes

- Next Monday is memorial day
  - Will need to makeup that class
- Wednesday…
  - Also need to makeup
Homework & Projects

- Programming:
  - Online submission
  - The tas will attempt to run it on their end...try to include any special instructions..
Labs

- Generally will create a few programs
- If everyone has a laptop we can work in class
- Online submissions – will create script in first lab!
- Will be around to answer questions hints
- Can NOT ask for code from other students
  - Can ask input/output
  - General ideas
  - Use your best judgment
Cheating

Don’t
Cheating Policy

- Plagiarism and cheating:
  - I’m all against it. It is unacceptable.
- You’re expected to do homeworks by yourself
  - This is a learning experience.
  - You will only cheat yourself.
  - My job is to help you learn, not catch you cheating, but....
- Automated tools to catch plagiarizers
  - [http://www.cs.berkeley.edu/~aiken/moss.html](http://www.cs.berkeley.edu/~aiken/moss.html)
  - Moving stuff around, renaming, etc. doesn’t help
- Results: instant zero on assignment, referral to academic committee
  - Columbia takes dishonesty very seriously
  - I’d much rather you come to me or the TAs for help
Feedback System

- Last minute of class will be set aside for feedback:
  - Feel free to leave it anonymous.
  - Content: Questions, comments, ideas, random thoughts.
- Can also email me, aim etc
- Summer is short, so provide feedback!
- Please feel free to show up to office hours or make an appointment at any time
  - I’m around a lot anyway
- Any Questions?

- Let's go around the room introduce each other....
Example:

- Task:
  - Create a program to run a web based game, which will be marketed to both desktop and phone users.
  - Any ideas on how to design the programming backend?
  - Ideas on how to measure requirements.
  - What else is important?
Last plug

- One of the points of computer science is to teach you how to think, learn, and analyze computational related information.
- Each course is a tool which you will collect for later use.
- Lots of tools in this course, since we will be covering many different topics and subjects.
Real programmers code in binary.
Let's get started

- Any questions?

- Our environment:
Eclipse

- www.eclipse.org

- Open source visual programming environment
- Real time feedback
- Will need c compiler
  - Cygwin (windows)
  - Xcode (mac)
Tech side

- Let’s assume you know a programming language
A *programming language* specifies the words and symbols that we can use to write a program.

A programming language employs a set of rules that dictate how the words and symbols can be put together to form valid *program statements*.

We will cover some programming languages:

- The only part to learn a new language involves recognizing the RULES which form the instructions.
- In addition to the theory behind design choices.
- But anything can be turned into a programming languages – even colors 😊.
Outline Today

- Intro to C
  - Background
  - Compiling
  - Basic data structures
  - Basic I/O
  - Types conversion
  - Loops
  - Branching
Why Learn C?

- C provides stronger control of low-level mechanisms such as memory allocation, specific memory locations

- For very specific programming projects:
  - C performance is usually better than Java and usually more predictable
  - No free lunch
Why Learn c continued

- Java hides many details needed for writing code, but in C you need to be careful because:
  - memory management responsibility left to you
  - explicit initialization and error detection left to you
  - generally, more lines of (your) code for the same functionality
  - more room for you to make mistakes

- Most older code is written in C (if you are lucky) might need there skills if you will be hired to upgrade or interface with in place tech
History

– Dennis Ritchie in late 1960s and early 1970s
– Systems programming language
– Goal: make OS portable across hardware platforms
– Not necessarily for real applications—could be written in Fortran or PL/I
C is old
- From early-70s, procedural language

C advantages:
- direct access to OS primitives (system calls)
- more control over memory
- fewer library issues—just execute

C disadvantages:
- language is portable, but APIs are not
- no easy graphics interface
- more control over memory (i.e., memory leaks)
- pre-processor can lead to obscure errors
C vs Java …Running

- Java programs are compiled and interpreted:
  - `javac` converts `foo.java` into `foo.class`
  - class file is not machine-specific—it is byte code
  - byte code is then interpreted by JVM
  - and each JVM is machine-specific

- C programs are compiled into object code
- and then linked into executables
- (to allow for multiple object files and libraries to be compiled together into one program):
C vs Java

- **Java program**
  - collection of classes
  - class containing main method is starting class
  - running java StartClass invokes StartClass.main method
  - JVM loads other classes as required

- **C program**
  - collection of functions
  - one function – main() – is starting function
  - running executable (default name a.out) starts main function
  - typically, single program with all user code linked in— but can be dynamic libraries (.dll, .so)
How to program in c

- You will learn the syntax of the language
  - Think…
  - Write foo.c

- Use compiler to translate your source code
  - gcc compiles foo.c into foo.o and then links foo.o and any other .o files into and a.out executable

- a.out is executed by you on specific OS and hardware
  - Note: the C compiler is machine-specific, creating code that executes on specific OS/hardware
  - So if you move machines, need to take the .c files and recompile
  - Will be learning some easy tools for this….
Code Example: hworld.c

```c
#include <stdio.h>

int main() {
    printf( "hello world!" );
    return 0;
}
```
breakdown

- Lets go over this example line by line

- The program that takes the code and turns it into a program is called the compiler. It includes a linker to link multiple object files together into executable.

- C allows you to drop hints to the compiler on how to behave.
Pre processor directives

- `#include <stdio.h>` to include header file stdio.h
  - `man stdio.h`
    - Will give you the manual page for this library if installed correctly
- Hash (`#`) lines processed by pre-processor and compiler never sees them
- No semicolon at end of pre-processor lines
- Stick to lower-case letters only—C is case-sensitive
Main function

- Mentioned c just collection of functions
- int main() { ... } is the only code executed

- printf( " /* message you want printed */ ");

- \n = newline, \t = tab

- \ (escape character) in front of other special characters
Brief Overview

- For the c section of the course, here are some tips
  1. Write your course code
  2. Try to compile
  3. Debug compile bugs, goto step 1
  4. Try step 2 again
  5. Run debugger to catch run time bugs
  6. Run memory profiler to catch memory bugs
  7. Have running product
  8. Add one last cool feature and jump to step 3 😊

or use eclipse and do everything at once 😊
How to make your c code run - manual

- gcc is the C compiler we’ll use in this class
- it’s a free compiler from Gnu (i.e., Gnu C Compiler)
- gcc translates C program into executable for some target machine platform
- default file name a.out
- behavior of gcc is controlled by command-line switches
- Will create files to help in compiling out programs

$ gcc hello.c
$ . a.out
hello world!
Compiling your program

Can also have a two-stage compilation
1. pre-process and compile: gcc -c hello.c
2. link: gcc -o hello hello.o

linking several modules:
>gcc -c a.c
  == a.o
>gcc -c b.c
  == b.o
>gcc -o hello a.o b.o
  == hello

using a library, for example the “math” library (libm):
>gcc -o calc calc.c -lm
C control flow

- blocks are enclosed in curly brackets
- functions are blocks
- main() is a function
- blocks have two parts:
  - variable declaration ("data segment")
  - code segment
- in C, variables have to be declared before they are used
- initializations can occur at the end of the declaration section, but before the code section
Break down of running program

- A program is a collection of functions

- The function named main is launched first

- when main ends, your program is done
  - or can crash the system earlier 😊
First C Program

/* First C program */

int main(void)
{
    printf("Hello Everyone\n");

    return 0;
}

compile

- gcc –o test simple.c
- ./test
Steps to running program

- Write code
  - Platform independent (for the most part)
- Preprocess the code
  - Understand and reinterpret parts
- Compile the code generate object files
  - Turn it into machine code, use optimizers
- Link object files to executable
- Load executable to running code
Your Own Environment

- **Windows:**
  - can use cygwin (free) with gcc (free)
  - gcc 3.4.4.1

- **Mac**
  - get gcc

- **Unix:**
  - cunix has it already
  - gcc 4.1.1

- **Eclipse:**
  - need to make sure you have c/cpp plug in
Split personalities

- In C and CPP, normal to divide definition of code (header files .h) and working code (.c files)

- So will have function declaration to tell compiler what each function looks like
  ```
  int foo();
  ```

- And function definitions
  ```
  int foo(){. . . . . . }
  ```
functions

- Like in java a function has 3 sections

1. Return type

2. Name

3. Argument lists
Comments

/* any text until this */

- convention for longer comments:
  /*
   * AverageGrade()
   * Given an array of grades, compute the average.
   */

- Don’t get carried away with comment boxed
- **** boxes - hard to edit, usually look ragged.
Where to begin?

- Lets talk about what are the primitive data types:
Data Types

- Very important when trying to resource memory/cpu
- float has 6 bits precision
- double has 15 bits precision
- Range can change depending on machine type, generally int is native to the machine type

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
</tr>
<tr>
<td>long</td>
<td>32</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
</tr>
</tbody>
</table>
Types II

- unsigned char
- unsigned short
- unsigned int
- unsigned long

- Byte size is the same, but can now have greater range
- Can look at /usr/include/limits.h
Use in functions

- in strict c mode:
  - Variables must be declared in the beginning of the function to be used
  - i.e top of your code is all your variables declaration
  - Followed by code logic

- Will see errors if you forget this
A macro

- A macro is a section of code, which has been given a name
- This is dealt with by preprocessor
- Can do a lot with macros

Important to understand:
- When you use the name, the preprocessor will replace it with the code contents
- Compiler only sees changed code
c pre-processor

- the C pre-processor (cpp) is a macro-processor which
  - manages a collection of macro definitions
  - reads a C program and transforms it for the compiler
  - pre-processor directives start with # at beginning of line
- used to:
  - include files with C code (typically, “header” files containing definitions; file names end with .h)
  - define new macros
  - conditionally compile parts of file (later – not today)
- gcc -E shows output of pre-processor
- Can be used independently of compiler
Example

- #define BUFFER_SIZE 1024

- Convention to use upper case
- Will be replaced exactly with the stuff after the name

- int x = BUFFER_SIZE;

- Why would this be useful?
pre-processor II

- library/file inclusion
  
  #include "filename.h"
  
  #include <filename2>

- "filename.h" relative to current directory
- <filename2> relative to /usr/include or in default path (specified by -I compiler directive); note that file is named verb+filename.h+

- import function prototypes (in contrast with Java import) examples:
  
  #include <stdio.h>
  
  #include "mydefs.h"
  
  #include "./home/shlomo/programs/defs.h"
Pre-processor cont.

#define name const-expression
#define name (param1,param2,...) expression
#undef symbol

- replaces name with constant or expression
- textual substitution
- symbolic names for global constants
- in-line functions (avoid function call overhead)
- type-independent code

#define MAXLEN 255
Example

#define MAXVALUE 100
#define check(x) ((x) < MAXVALUE)
if (check(i)) { ...}

- becomes
if ((i) < 100) {...}

- Caution: don’t treat macros like function calls
#define valid(x) ((x) > 0 && (x) < 20)
- is called like:
if (valid(x++)) {...}
- and will become:
valid(x++) -> ((x++) > 0 && (x++) < 20)
- and may not do what you intended...
- conditional compilation

- pre-processor checks value of expression
- if true, outputs code segment 1, otherwise code segment 2
- machine or OS-dependent code

- can be used to comment out chunks of code—bad!
- (but can be helpful for quick and dirty debugging :-)

- example:
  ```c
  #define OS linux
  ...
  #if OS == linux
  puts( "Wow you are running Linux!" );
  #else
  puts( "why are you running something else??" );
  #endif
  ```
- ifndef
- for boolean flags, easier:
  
```
#ifdef name
  code segment 1
#else
  code segment 2
#endif
```
- pre-processor checks if name has been defined, e.g.:
  
```
#define USEDB
```
- if so, use code segment 1, otherwise 2
Intro arrays

- An array is a group of memory locations with the same name and type.
- To get to a particular element in the array we need:
  - data type
  - name
  - Length or position
- Array length can be determined:
  - statically— at compile time (when we code)
    - e.g., char str1[10];
  - dynamically— at run time (more on this later)
    - e.g., char *str2;
Defining a variable is called “allocating memory” to store that variable.

Defining an array means allocating memory for a group of bytes.

Individual array elements are indexed:
- starting with 0
- ending with length -1

Indices follow array name, enclosed in square brackets ([ ])
e.g., name[25]
- Initializing the arrays are your problem
  ```c
  int a[3];
  ....
  X = a[1]; ......
  ```

- Bound checking is your problem
  ```c
  printf("%d", a[100]); ......
  ```
We can say for example
\[
\text{int varX} = \text{C[4]} / 5;
\]

Declarations:
\[
\text{int b[100]}, \text{v[3]};
\]
More arrays

- Can also create arrays in the following manners
  1. int a[] = {1,2,3};
  2. int b[3] = {6,3,7};
  3. int n[10] = {0};

- In general: you need to initialize the array elements
  - 3 is a trick case.
Command Line Args

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

- **argc** is the argument count
- **argv** is the argument vector
  - array of strings with command-line arguments
- **the int value is the return value**
  - convention: return value of 0 means success,
  - > 0 means there was some kind of error
  - can also declare as **void** (no return value)
Library

- Access libraries using the include statement
- Generally include header files
  - Your own c files will include .h
  - .h NEVER includes .c
  - Tell makefile about all .c files
- Compiler links them automatically
- Example:
  - Standard input/output: stdio.h
  - To look up information use the man page:
    ```
    man stdio
    ```
NAME

stdio - standard input/output library functions

SYNOPSIS

#include <stdio.h>

FILE *stdin;
FILE *stdout;
FILE *stderr;

DESCRIPTION

The standard I/O library provides a simple and efficient buffered stream I/O interface. Input and output is mapped into logical data streams and the physical I/O characteristics are concealed. The functions and macros are listed below; more information is available from the individual man pages.

A stream is associated with an external file (which may be a physical device) by opening a file, which may involve creating a new file. Creating an existing file causes its former contents to be discarded. If a file can support positioning requests (such as a disk file, as opposed to a terminal) then a file position indicator associated with
stdio.h

- Access stdio functions by
  - using `#include <stdio.h>`
  - compiler links it automatically

- defines `stdin`, `stdout`, `stderr`
- use for character, string and file I/O (later)

- Example: `printf`
printf Function

- The way printf works is it takes a format to print out and then the data to add to the format

- One or more of the following:
  - `%[flags][width][.precision][modifiers]type`
  - “%d”
    - Means a single number
  - “%d %d %d”
    - ??
printf ("%d %d",a,b);
**stdio.h : printf, type specifier**

- `int printf(const char *format, ...)`: formatted output to stdout

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Type Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c</code></td>
<td>Character</td>
<td><code>a</code></td>
</tr>
<tr>
<td><code>d</code> or <code>i</code></td>
<td>Signed decimal integer</td>
<td>392</td>
</tr>
<tr>
<td><code>e</code></td>
<td>Scientific notation (mantise/exponent) using e character</td>
<td>3.9265e2</td>
</tr>
<tr>
<td><code>E</code></td>
<td>Scientific notation (mantise/exponent) using E character</td>
<td>3.9265E2</td>
</tr>
<tr>
<td><code>f</code></td>
<td>Decimal floating point</td>
<td>392.65</td>
</tr>
<tr>
<td><code>g</code></td>
<td>Use shorter %e or %f</td>
<td>392.65</td>
</tr>
<tr>
<td><code>G</code></td>
<td>Use shorter %E or %f</td>
<td>392.65</td>
</tr>
<tr>
<td><code>o</code></td>
<td>Signed octal</td>
<td>610</td>
</tr>
<tr>
<td><code>s</code></td>
<td>String of characters</td>
<td><code>sample</code></td>
</tr>
<tr>
<td><code>u</code></td>
<td>Unsigned decimal integer</td>
<td>7235</td>
</tr>
<tr>
<td><code>x</code></td>
<td>Unsigned hexadecimal integer</td>
<td><code>7fa</code></td>
</tr>
<tr>
<td><code>X</code></td>
<td>Unsigned hexadecimal integer (capital letters)</td>
<td><code>7FA</code></td>
</tr>
<tr>
<td><code>p</code></td>
<td>Address pointed by the argument</td>
<td><code>B800:0000</code></td>
</tr>
<tr>
<td><code>n</code></td>
<td>Nothing printed. The argument must be a pointer to integer where the number of characters written so far will be stored.</td>
<td></td>
</tr>
</tbody>
</table>
printf flags

%[flags][width][ precision][modifiers]type

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Left align within the given width. (right align is the default).</td>
</tr>
<tr>
<td>+</td>
<td>Forces to precede the result with a sign (+ or -) if signed type. (by default only -(minus) is printed).</td>
</tr>
<tr>
<td>Blank</td>
<td>If the argument is a positive signed value, a blank is inserted before the number.</td>
</tr>
<tr>
<td>#</td>
<td>Used with o, x or X type the value is preceeded with 0, 0x or 0X respectively if non-zero.</td>
</tr>
<tr>
<td></td>
<td>Used with e, E or f forces the output value to contain a decimal point even if only zeros follow.</td>
</tr>
<tr>
<td></td>
<td>Used with g or G the result is the same as e or E but trailing zeros are not removed.</td>
</tr>
</tbody>
</table>
example

int class_size = 35;
char class_name[15] = “3157 adv prog”;

printf("Welcome to our test program\n");

printf( “the %s class size is %d”,
        class_name, class_size);
int array

1. #include <stdio.h>
2. #define MAX 6
3. int main( void ) {
4.  int arr[MAX] = { -45, 6, 0, 72, 1543, 62 };
5.  int i;
6.  for ( i=0; i<MAX; i++ ) {
7.    printf( "[%d] = %d \n", i, arr[i] );
8.  }
9. } /* end of main() */
stdio.h: scanf

- int scanf(const char *format, ...)

Example: scanf/printf

```c
#include <stdio.h>
void main( void ) {
  int n = 0; /* initialization required */

  printf( "how much wood could a woodchuck chuck\n" );
  printf( "if a woodchuck could chuck wood?" ); /* prompt user */

  scanf( "%d",&n ); /* read input */

  printf( "the woodchuck can chuck %d pieces of wood!\n",n 
        );
  return;
}
```
output

$ a.out

how much wood could a woodchuck chuck if a woodchuck could chuck wood? 12345
the woodchuck can chuck 12345 pieces of wood!
Loops

- loops in C are just like in Java

- there are 2 methods for looping:
  - counter-controlled (loop for a fixed number of times)
  - sentinel-controlled (loop while a condition is true)

- there are 3 statements for implementing the 2 methodologies:
  - for
  - while
  - do...while

- as always: beware the infinite loop!

- Ctrl-C interrupts your executing C program
Branching

- branching in C is just like in Java

- there are 2 ways to do branching:
  - if/else
  - switch

- questions:
  - which is more flexible and powerful?
  - one can always be translated into the other, but not the other way around— which is which?
WARNING!!!!
Points

- Unique to c/c++
- Lots of confusion
- Please ask if something doesn’t make sense
Pointer power

- Variables that contain memory addresses as their values
- Data types we’ve learned about in C use direct addressing
- Pointers facilitate indirect addressing

Declaring pointers:
- Pointers indirectly address memory where data of the types we’ve already discussed is stored (e.g., int, char, float, etc.)
- Declaration uses asterisks (*) to indicate a pointer to a memory location storing a particular data type
- Called dereferencing a pointer

Example:
```c
int *count;
float *avg;
```
Pointers: nitty gritty

- ampersand & is used to get the address of a variable location (dereference a pointer)

- example:
  ```c
  int count = 12;
  int *countPtr = &count;
  ```
  &count returns the address of count and stores it in the pointer variable countPtr
Another example

- here’s another example:

```c
int i = 3, j = -99;
int count = 12;
int *countPtr = &count;
printf ( "%d", *countPtr);
```

- Here is the memory picture:
Arrays as pointers

- an array is some number of contiguous memory locations
- an array definition is really a pointer to the starting memory location of the array
- and pointers are really integers
- so you can perform integer arithmetic on them
- e.g., +1 increments a pointer, -1 decrements
- you can use this to move from one array element to another
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int i, *j, arr[5];
    srand( time ( NULL ));
    for ( i=0; i<5; i++ )
        arr[i] = rand() % 100;
    printf( "arr=%p\n",arr );
    for ( i=0; i<5; i++ ) {
        printf( "i=%d arr[i]=%d &arr[i]=%p\n",i,arr[i],&arr[i] );
    }
    j = &arr[0];
    printf( "\nj=%p *j=%d\n",j,*j );
    j++;
    printf( "after adding 1 to j:\n j=%p *j=%d\n",j,*j );
}
Output

\[\text{arr}=0xbfffff4f0\]
\[i=0 \quad \text{arr}[i]=29 \quad \& \quad \text{arr}[i]=0xbfffff4f0\]
\[i=1 \quad \text{arr}[i]=8 \quad \& \quad \text{arr}[i]=0xbfffff4f4\]
\[i=2 \quad \text{arr}[i]=18 \quad \& \quad \text{arr}[i]=0xbfffff4f8\]
\[i=3 \quad \text{arr}[i]=95 \quad \& \quad \text{arr}[i]=0xbfffff4fc\]
\[i=4 \quad \text{arr}[i]=48 \quad \& \quad \text{arr}[i]=0xbfffff500\]
\[j=0xbfffff4f0 \quad *j=29\]
\[\text{after adding 1 to } j:\]
\[j=0xbfffff4f4 \quad *j=8\]
Pointer operations

- HUGE Difference between
  - `ptr++`
  - `*ptr++`

- `int b[5] ....`
  ```
  int *bPtr;
  ```
  ```
  bPtr = b       // or
  bPtr = &b[0]
  ```
Careful when moving pointers

\[ \text{bPTr} += 2; \]

the memory location isn’t simply incremented by 2.....depends on size of type being pointed to.
Strings

- You will be doing this in the lab
  - storing multiple characters as array
  - data type is still char
- BUT it has a length
  - Max length
  - Current length (till the \0)
  - last character the is terminator: '\0', aka NULL
  - string constants are surrounded by double quotes: "

example:
- char s[6] = "ABCDE";
String II

- char s[6] = “ABCDE”;
- Memory storage looks like:
  
  | A | B | C | D | E | \0 |

- Need to remember that you are really accessing indices 0 – (length-2) since the value at length-1 is always \0

- Need to allocate enough memory
Using strings

- printing strings
- format sequence: %s
- example:

```c
#include <stdio.h>
int main() {
    char str[6] = "ABCDE";
    printf( "str = %s\n", str );
} /* end of main() */
```
String Library

- to use the string library, include the header in your C source file:
  `#include <string.h>`
- string length function:
  `int strlen( char *s );`
  this function returns the number of characters in s; note that this is NOT the same thing as the number of characters allocated for the string array ..... 

- string comparison function:
  `int strcmp( const char *s1, const char *s2 );`
  “This function returns an integer greater than, equal to, or less than 0, if the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2 respectively. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes that differ in the strings being compared.”

`man strcmp`
copying functions:

char *strcpy( char *dest, char *source );

- copies characters from source array into dest array up to NULL
- Need to make sure you have space...

char *strncpy( char *dest, char *source, int num );

- copies characters from source array into dest array; stops after num characters (if no NULL before that); appends NUL
Search

char *strchr( const char *source, const char ch );
- returns pointer to first occurrence of ch in source; NULL if none

char *strstr( const char *source, const char *search );
- return pointer to first occurrence of search in source
String Parsing

char *strtok( char *s1, const char *s2 );
- breaks string s1 into a series of tokens, delimited by s2
- called the first time with s1 equal to the string you want to break up
- called subsequent times with NULL as the first argument
- each time is called, it returns the next token on the string
- returns null when no more tokens remain
Example

```c
char inputline[1024];
char *name, *rank, *serial_num;
printf( "enter name+rank+serial number: " );
scanf( "%s", inputline );
name = strtok( inputline,"+" );
r ank = strtok( null,"+" );
serial_num = strtok( null,"+" );
```
Formatting functions

int sscanf(char *string, char *format, ...)
- parse the contents of string according to format
- placed the parsed items into 3rd, 4th, 5th, ... argument
- return the number of successful conversions

int sprintf(char *buffer, char *format, ...)
- produce a string formatted according to format
- place this string into the buffer
- the 3rd, 4th, 5th, ... arguments are formatted
- return number of successful conversions

- format characters are like printf and scanf (see notes from earlier lectures)
Memory allocations

- One of the most powerful features of C is the ability of the programmer to create more memory space during the execution of the program.
- Limited by physical machine memory
- If you want to be able to create memory, you also need to free it manually
malloc /sizeof / free

- charPtr = malloc ( sizeof ( … ) );

- free (charPtr)
Memory allocations

- If you want to be able to create memory, you also need to free it manually.
- When you don’t it is called memory leaking...more on this later.
Array vs memory allocation

- Arrays are great when you have a rough idea of how many items you will be dealing with:
  - 10 numbers
  - 30 students
  - Less than 256 characters of input
Map of memory

- Think of memory as a box
- Main is placed on the bottom and any variable on top of that
- Any function call gets placed on top of that
- This part of memory grows upward
- It is called the stack
- Your program is over when the stack is empty
heap

- The heap is the other side of memory
- Global variables, and allocated memory is created on the heap
- It grows downwards
Dynamic Memory Allocation

- pre-allocated memory comes from the “stack”
- dynamically allocated memory comes from the “heap”
- To get memory you allocated (malloc) memory, and to let it go, you free it (free)
- family of functions in stdlib, including:
  ```c
  void *malloc( size_t size );
  void *realloc( void *ptr, size_t size );
  void free( void * );
  ```
malloc and realloc return a generic pointer (void *) and you have to “cast” the return to the type of pointer you want.

That is if you are allocating a bunch of characters, you say:

Ptr = (char*) malloc....
#include <stdio.h>
#include <stdlib.h>
#define BLKSIZ 10
main() {
    FILE *fp;
    char *buf, k;
    int bufsiz, i;
    // open file for reading
    if (( fp = fopen( "myfile.dat","r" )) == NULL ) {
        perror( "error opening myfile.dat" );
        exit( 1 );
    }
    // allocate memory for input buffer
    bufsiz = BLKSIZ;
    buf = (char *)malloc( sizeof(char)*bufsiz );
// read contents of file
i = 0;
while ((( k = fgetc( fp )) != EOF )) {
    buf[i++] = k;
    if ( i == bufsiz ) {
        bufsiz += BLKSIZE;
        buf = (char *)realloc( buf,sizeof(char)*bufsiz );
    }
}
if ( i >= bufsiz-1 ) {
    bufsiz += BLKSIZE;
    buf = (char *)realloc( buf,sizeof(char)*bufsiz );
}
buf[i] = '\0';
// output file contents to the screen
printf( "buf=[%s]\n",buf );
// close file
fclose( fp );
} // end main()
Dynamic memory

- malloc() allocates a block of memory:
  ```c
  void *malloc( size_t size );
  ```
- lifetime of the block is until memory is freed, with free():
  ```c
  void free( void *ptr );
  ```

- example:
  ```c
  int *dynvec, num_elements;
  printf( "how many elements do you want to enter? " );
  scanf( "%d", &num_elements );
  dynvec = (int *)malloc( sizeof(int) * num_elements );
  ```
Useful Unix commands

- ls -la
- chmod
- man
- uname –a
- pwd
- who
- finger
- cd
- mkdir
- locate
- which