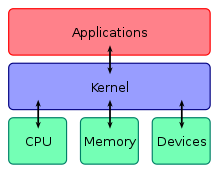
# C Programming

## Lecture 2: Using Command Line



**Kernel (operating system)**:

* It is a program that manages the interactions between the hardware and software

**Operating system:**

* Tanenbaum’s **MINIX** book ***Operating Systems Design and Implementation*** was hugely influential and is still widely used in OS courses
* Linux Operating System– Linus Torvalds
  + Ported bash and gcc
  + Most routers, servers and routers run Linux
  + Lots of Linux distributions
* Unix was the originally operating system – commercial. Linux is the Open Source OS.
  + Linux is inspired by MINIX (a Unix-like OS)

**Command Shell:**

* **It is program for launching other programs**
* Windows Explorer, OSX Finder, Command Line

**Unix File System**

* Hierarchy – only one root directory
* Path separator is “/”
* Unix Convention: everything’s a file, including drivers.

**File System Layout**

|  |  |
| --- | --- |
| **Directory** | **Contains** |
| / | Root directory (only one in the system) |
| /bin | Important executable single-user binaries |
| /boot | Boot loader, kernel, system startup files |
| /dev | Devices |
| /etc | Configuration files |
| /home | Users’ files |
| /lib | System libraries needed by programs in /bin |
| /opt | Optional programs (binaries in /opt/bin, libraries in /opt/lib, etc) |
| /sbin | System binaries |
| /tmp | Temporary files |
| /usr | Secondary file hierarchy for multi-user programs |
| /var | Variable files (files that change frequently) – e.g. files under /var/log and /var/mail |
| /proc | process information |

**Help**

* “man” command
* Example: man ls (more information about ls command)
* Keyword search for man title pages
  + apropos graphics

**Paths:**

* **absolute path:** 
  + begins with a slash
  + / indicates top of the hierarchy
  + E.g. /home/ram/subdir/hello.c
* **Relative path:**
  + Start from the current directory
  + Subdir/hello.c
* Special character ~
  + User’s home directory
  + ~/subdir/hello.c is expanded to /home/ram/subdir/hello.c

Commands:

* pwd – print working directory
* pushd/popd - use when you want to take a brief look into a directory somewhere and then come back to where you currently are
  + works like a stack (hence “push” and “pop”)
  + *pushd takes an argument (directory)*
    - *pushes your current working directory onto* a stack and then changes to the new directory
  + popd pops a directory off the stack and then changes to it
* mv – move or rename files/directories
  + mv /home/ram/subdir/hello.c ~
  + moves hello.c from subdir into my home director
* mkdir: create a directory
  + mkdir /home/ram/subdir2
* List particular files:
  + Ls c\* list files that begin with c
  + Ls \*c list all files that end with c
  + Ls \*.c list all files whose name end in .c
* Permissions
  + Perform access as “root” using the *“sudo” command*
  + Use the chmod command to change permission
    - Chmod u+x ~/myprogram
    - User premissions:
      * u-User – owner of file
      * g-Group-members of group
      * o: other, excusing user and group
      * a: all
    - Values:
      * - removes permission
      * + grant permission
      * =: equals sign
    - Permissions:
      * r: read
      * w: write
      * x: executable

**Installing Software**:

* **Command line:**
  + ***apt-get install jedit*** (install jedit text editor)
  + ***apt-get update (update package)***
  + sudo if required
* **GUI:** Synaptic Package Manager

## Lecture 3 – Beginning C

|  |
| --- |
| **// simple.c**  int main()  {  return 0;  } |

Compile with default output filename: ***gcc simple.c***

* Produce executable called “a.out”

**Compile with different output file name** using ***-o filename tag***

* gcc **-o simple** simple.c
* Run the program: ./simple

About simple.c

* Doesn’t display anything to console
* ***The returned value gets passed to the shell***
  + **Bash** stores the returned value into a ***special variable called $?***
  + **$? Also stores the return values from every command you run**
* **Unix Convention**: **0 means no error**
  + Test using “**echo $?”** command

|  |  |
| --- | --- |
| #include <stdio.h>  int main()  {  printf(“Hello, world!\n”);  return 0;  } | ***\n – means “new line”***  ***#include <stdio.h>***   * Directive(instruction) to tell C pre-processor to copy the file called “stdio.h” into the source   // printf- output text |

**#include**

* **C Preprocessor** – first stage of compilation
  + **Look for directives “#” and perform relevant task**
* **#include**: open the given file and copy into the current source file
  + If filename in <angle brackets>, looks in *system include path* first
  + If file in “straight quotes”, look in the current directory first
* **#define A B**: search-and-replace all instances of A with B
  + **Useful for defining constants** e.g.
  + #define PI 3.14
* Other Directives(instructions) include:
  + #ifdef (“if defined”)
  + #ifndef (if not defined)

**What is stdio.h?**

* **Definitions**: lots of important types such as FILE
* **Declarations**: lots of important of constants
* **Function prototypes:**
  + Tell the compiler what parameters and return type are for each function
  + E.g. printf can take a string as parameter

**More about headers**

* Group related functions together into files
* ***Code stored in the .c files***
* ***Declarations, Definitions (e.g constants #DEFINE A B) and function prototypes*** stored in the .h files
* *Must include #include reference to the .h file*

**Header Guards**

* **Problem**: when you **#include** something that has been **included (e.g. function or constant already declared)**
  + **Can’t include files twice**
* **Solution**:
  + **Use #define and #indef**

|  |
| --- |
| **#ifndef GEOMETRY\_DEFS\_H**  **#define GEOMETRY\_DEFS\_H**  #define PI 3.141592653489  double area(double radius);  **typedef struct** {  float x;  float y;  } point  **#endif** |

**#define at compile time**

**Problem to adding this code for logging purposes to each file: Annoying (e.g. change each file)**

|  |
| --- |
| #ifdef DEBUG  printf("We got here!");  #endif |

**Solution**:

* Define a variable at compile time with -D flag
* gcc -**DDEBUG** -o program program.c

**Locating .h file**

* Specify “**include path**” on the command line
  + **Include path – set of locations that is used for finding resources reference by include statement**
* gcc **–I./includes/myheaders** –o myprog myprog.c

If you want to use more than one include path, simply use more than one –I option:

* gcc **–I./includes/myheaders** \  
   **-I/usr/local/include/somelibrary** \  
   –o myprog myprog.c
* Note the backslash character – you can use it to tell bash that you want to continue the current command on the next line

**printf:**

* is located in a system library called **libc**
* **the C standard library**



* How the C compiler works
  + Preprocessing (resolve directives #include, #define, #ifnelse, #ifelse)
  + compiling
  + linking

**Compiling without linking**

* gcc has option for this:
  + gcc **-c** myfile.c
  + produces “myfile.o” object file
* Don’t want to link because we only want to recompile files that we edited recently (e.g. Don’t recompile all .c source files , but only those that have changed)

**Link another library**

* List of available libraries in
  + /lib
  + /usr/lib
  + /user/local/lib
* Tell linker (resolve unimplemented functions and unresolved symbols)
  + **Use functions from another library**, with the **-L option**
  + gcc –I/usr/include/somelib **–L**/usr/lib/somelib \  
     -o myprog myprog.c

**Other things that the linker does:**

* The **linker** inserts the code called **crt0**, which calls the **main()**
  + crt – C run time
  + 0 – the very beginning
* crt0 is responsible for
  + setting up the parameters to main()
  + calling main()
  + handling the return value and passing it back to the shell

|  |  |
| --- | --- |
| Option | Meaning |
| **-c** | compile source but don’t link |
| **–S** | stop after compilation, do not assemble |
| **-E** | run the pre-processor only |
| **-g** | include debugging information |
| **-O*level*** | optimise the code to *level* (e.g. -O3) |
| **-W*warn*** | turn on/off particular warnings (-Wall is good) |
| **–I*dir*** | specify a directory to look for include files |
| **-L*dir*** | specify a directory to look for library files |
| **-D*macro[=defn]*** | define a macro (#define *macro defn)* |
| **-U*macro*** | undefine a macro |
| **-o *outfile*** | place output in *outfile* |

## Lecture 3 – More about C

C Variables and types

* Must be declared before use
  + int i, j, k = 3;
  + float pi= 3.141592653f;
  + double d,e = 1.0e-32;
  + char c = ‘c’;
  + char newline = ‘\n’;
* Type conversion
  + int i = 3;
  + float f = i;
  + int j;
  + j = (int) f;

**C data types**

* C data types are not guaranteed to be of a fixed size,
* **Base data types in order of size:**
  + **Integer**: char, short, int, long, long long
  + **Floating point**: float and double
* C has an operator that gives you the size of a particular type/variable:
  + sizeof

**Arrays:**

* C supports single and multi-dimensional arrays
  + **int a[10];**
  + char name[MAX\_CHAR];
  + a[2] = 2;
  + **char name2[] = “Jones”**
* *Multi-dimensional arrays* can be also created using *array of pointers* to other arrays
  + int a[10][10];
  + a[i][j] = i \*j;
  + char \* argv[];

**enumerated types**

* **Enumerated define a list of identifiers and constants**
  + **Names must be distinct and values don’t need to be distinct**
  + ***Be default, the first identifier has value of 0, and the next has a value of 1.***
    - ***enum boolean { NO, YES };***
    - NO = 0 and YES = 1
* ***State Explicit values for the identifiers***
  + enum **escapes** { BELL = ‘\a’ , BACKSPACE = ‘\b’, TAB = ‘\t’,

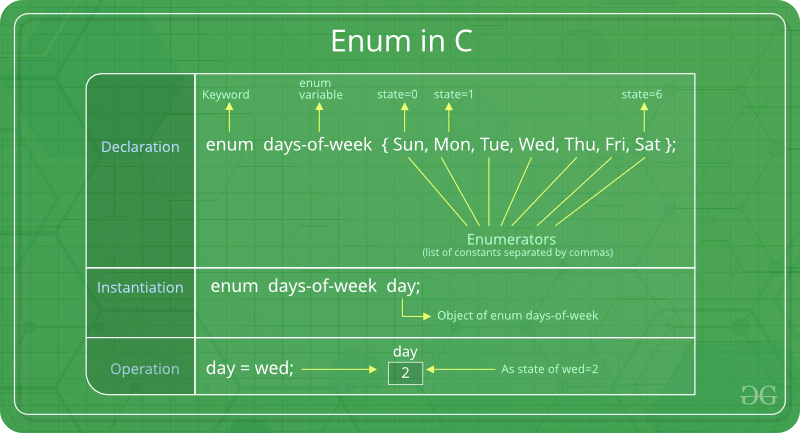
NEWLINE = ‘\n’, VTAB = ‘\v’, RETURN = ‘\r’ };

* + ***enum months = { JAN = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };***

|  |
| --- |
| #include <stdio.h>  enum State {Working = 1, Failed = 0, Freezed = 0};    int main()  {     printf("%d, %d, %d", Working, Failed, Freezed);     return 0;  } // Output: 1, 0, 0 |

|  |
| --- |
| #include <stdio.h>  enum day {sunday, monday, tuesday, wednesday, thursday, friday, saturday};    int main()  {  **enum day** d = thursday;      printf("The day number stored in d is %d", d);      return 0;  }  // Output: The day number stored in d is 4 |

**Source:** <https://www.geeksforgeeks.org/enumeration-enum-c/>



**C Functions**

* A C program consists of a number of functions.
* Functions must be declared before they can be used
  + **Declare Function Prototype in the .h header file**
* **Functions may return:**
  + **a value of a specified type OR**
  + **“void” if they don’t return any value**

Examples:

|  |  |
| --- | --- |
| int foo();  int foo(void);  float bar(float f);  void reset(int i, char c);  void load(int a[] ) | int add(int a, int b) {  return a + b;  }  void setup(int j){  while(--j) { gtk[j] = 0; }  } |

**main function**

**If your program doesn’t have the main(), the linker will terminate.**

The “**main()**” function accepts **2 optional parameters**:

* argc – argument count
  + the number of command line arguments passed to the program
* argv – argument vector
  + 1D array of strings representing the arguments
  + \* - variable length argument/string

|  |  |
| --- | --- |
| int main(int argc, char \*argv[] ) {  // Your Code  return 0;  } | #include <stdio.h>  int main (int argc, char \*argv[])  {  return 0;  } |

**Variable Scope**

* **Global Scope**: Declared outside a function
  + Visible throughout entire program
* **Block Scope**: Declared inside curly braces within a function
  + **Only accessible within that block**
* **File Scope**: Declared with **static modifier** *outside* a function
  + Visible only within the file in which it’s declared
* **Static Local Variable**: Declared with **static modifier** *inside* a function
  + Visible only within the function, but persists between invocations
  + Function can “remember” state between calls



**C Arithmetic Operators (Same as Java)**

|  |  |
| --- | --- |
| Operator | Meaning |
| = | assignment |
| +, -, \*, / | add, subtract, multiply, divide |
| - | unary minus (negation): e.g. x = -y; |
| % | modulus (remainder) |
| ++, -- | increment and decrement (++x; y++; --x; y--) |
| ==, != | equals, not equals |
| <, >, <=, >= | less than, greater than, less than or equal, greater than or equal |
| &&, ||, ! | logical AND, OR, and NOT |

**C bitwise operators**

* **x << n gives result of shifting x left n bits, x >> n shifts right n bits.**
* Used to be useful for quick multiplication/division by powers of 2
  + still useful for bit manipulation

**Set and View Environmental Variable**

Set a variable in “Bash”:

* VARNAME=value
* FOO=bar
* FOO2 =”more than one word”

View the contents of a variable:

* echo $FOO

Bash Commands:

* Copy all files: cp \* /home/tom/backup
* Recursive copy (nested folders and files): cp -R \* directory
* ***“rm -rf”*** is equivalent to “***rm -r -f***”

**(a++)** will return the value of **a** first, and then increment it by 1.

**(++a)** will increment **a** first and then return the new value

**Making global variables visible**

* **Global variables define once, in one source file** (because declaring twice would be an error)
* **How to make global variables visible in other source files**?
  + ***extern*** declarations in the header file

|  |  |  |
| --- | --- | --- |
| box.h | box.c | boxuser.c |
| #if !defined(**BOX\_H**)  #define BOX\_H  #define MAX\_BOX 20  **extern int box\_count;**  typedef struct box {…}  #endif /\* BOX\_H \*/  // Declare Variable and make it gloabl | #include “box.h”  // define once  **int box\_count=0;**  **void init\_box(box b**) {  ...  }  .  .  .  // Declare and Define Variable once (int box\_count) | #include “box.h”  int main()  {  box mybox;  box\_count=1;  init\_box(mybox);  }  .  . |

“extern”:

* Extends visibility of the ***C Variables and C Functions (function prototypes have implied extern)***
* **Declaration**: **state that the variable or function exists somewhere in the program**, but memory is allocated to function/variable. (e.g. **extern int box\_count**)
* Definition: Declaration + allocate memory. (e.g. **int box\_count**)

Variable/Function can be ***declared any number of times.***

Variable/Function can **only be defined once.**

<https://www.geeksforgeeks.org/understanding-extern-keyword-in-c/>

## Lecture 5: Pointers

**Basic I/O**:

* ***stdin,stdout, and stderr*** are the standard steams for Input/ouput
* **printf** write to **stdout**
  + **fprintf** (writes to a specified steams – stdout or other files)
  + **sprintf** ( writes to a specified character array in memory)
* Each output function requires formatting characters for any variables
  + printf(“You are %d %s old\n”, age, age == 1 ? “year” : “years”);

**printf Format Specifications:**

|  |  |
| --- | --- |
| Format Specifier | Meaning |
| **%c** | **Single character** |
| **%d or %i** | **Signed decimal integer** |
| **%e %E** | **Floating-point number, scientific format (e or E)** |
| **%f** | **Floating-point number, decimal notation** |
| **%d** | **Double precision floating-point number, decimal notation** |
| **%g %G** | **Causes %f or %e/%E to be used, whichever is shorter** |
| **%o** | **Unsigned octal integer** |
| **%p** | **Pointer** |
| **%s** | **Character string** |
| **%u** | **Unsigned decimal integer** |
| **%x %X** | **Unsigned hex integer using a-f/A-F** |
| **%%** | **Print the percent sign** |

|  |
| --- |
| **// if statement**  if (hours > 40) {  printf(“You can go home now”);  overtime += (hours - 40);  } else {  printf(“keep working\n”);  printf(“stop playing games\n”);  } |

|  |
| --- |
| while (--argc) {  /\* pre-checked loop \*/  printf(“%s\n” ,\*argv++);  } |

|  |
| --- |
| // for loop  for (count = 0; count < MAX\_COUNT; ++count)  print(“hello **%d**\n”, count);  for (**r = 0, i = 0**, j = 100; j > i; ++i, --j)  r += i \* j; |

|  |
| --- |
| // switch statement  switch (c) {  case ‘a’: /\* do something \*/  break;  case ‘b’: /\* do something else \*/  break;  default: /\* if no other options match \*/  } |

**Pointers**

Background:

* Variables are stored in main memory (stack- stores variable and primitive values)
* **Variables contain reference to address in the memory**
* Linker translate “names” into “addresses”
* name -> memory address -> value

The addresses of local variables and parameters are sorted out by the compiler; the addresses of globals, static variables, and functions are sorted out by the linker.

**Pointers**

* ***A pointer stores an address (reference)***  in memory (e.g. int \*ip)
* This may be the address of another variable or value.



**Pointer Syntax**

* **int \*ip;** // ip is pointer that points to an integer value (memory address containing a integer value)
  + **The variable *ip* represents a memory address**
  + **This address references or points to an integer value.**
* **Pointers use two operators: \* and unary**
  + **\* de-references (i.e. follows the pointer -pointers to point to int):**

\*ip = 3; // assign 3 to the address pointed by i

// set contents of the memory address to 3

* + **& - gives the address of a variable**

ip = &j; // make ip point to j

* Pointer can be assigned to the type the object points to
* **Null – pointer isn’t pointing at anything (e.g. used for initialization)**

**Declaring and using pointers**

**Declare by adding \*s** to indicate levels of indirection

* int i; // int
* **int \*ip ; // pointer to int**
* **int \*\*ip; // pointer to pointer to int**

Adding \*s means you write through levels of indirection:

* i = 3;
* \*ip = 3;
* \*\*ipp = 3;

**Changing Pointers**

|  |
| --- |
| int i = 3;  int \*ip = &i; // declare pointer to an integer and store address of variable i |

Changing the value of **i** also changes the value of **\*ip** ( but not *ip* itself – it still points to the same address)

* ++i; /\* now i == 4 == \*ip \*/
* ++(\*ip); /\* now i == 5 == \*ip \*/

To change what “***ip***” points to, we assign a new value to ***ip*** itself:

* int j = 2;
* ip = &j; /\* now \*ip == 2 \*/

A **pointer** to **NULL** can be tested, but not differenced:

* ip = NULL;
* **if( ip != NULL)** (\*ip)++; // ok
* **\*ip = 4;** // will cause program to crash

**Void Pointers**

|  |  |
| --- | --- |
| Declaration | Meaning |
| int \* i; | Pointer to a int |
| char \* string; | Pointer to a character (normally a string, null terminated) |
| double \* d; | Pointer to a double |
| void \* vp; | Pointer to any type |

* A “**void**” pointer can point to anything – **allows conversion between different types**
* It must be used with care – there is not type checking
  + d = i; /\* illgal\*/
  + vp = (void \*)i; /\* legal \*/
  + d = (double \*) vp; /\* legal \*/

**Call by Value or Reference**

* C doesn’t support call by reference directly
* Pointers can give the equivalent of call by reference.

|  |  |
| --- | --- |
| Call by Value | Call by reference |
| // fn to compute the square  int sqr(float f) {  return f\*f;  }  float sq\_x, x = 3.141;  sq\_x = sqr(x); // legal  sq\_x = sqr(2.7) // legal | **void swap(int \*x), int \*y) {**  **int tmp = \*y;**  **\*y = \*x;**  **\*x = tmp;**  **}**  int a = 3; b =4;  **swap(&a, &b); // a =4 and b =3**  swap(&4,&3); // illegal |

**Pointers and Arrays**

* **a[10] is equivalent to \*(a+10)**
* Pointers have **arithmetic operators** just like any other basic types
  + **int \*i;**
  + i+=5; // make I point 5 ints ahead



Question: why do you think the addresses across the **bottom of the diagram are going up by 4s and not by 1s**? (**int is 4 bytes**) What does this tell you about pointer arithmetic in C?

* \*i = i[0]; // first element
* \*(i+5) = i[5]; // fifth element

**Example:**

|  |
| --- |
| **/\* pointers.c -- show array and pointer are the same \*/**  #include <stdio.h>  int main() {  int i; /\* general purpose counter \*/  /\* create array and initialise it \*/. (alternative: ***int a[] = {11,14,0,8,22,6,201…};*** )  int a[10] = {11, 14, 0, 8, 22, 6, 201, 1, -7, 42};  /\* print array using **array notation** \*/  for (i = 0; i < 10; ++i)  printf("a[%d] = %d\n", i, a[i]);  /\* print array using **pointer notation** \*/  for (i = 0; i < 10; ++i)  printf("\*(a + %d) = %d\n", i, \*(a + i));  return 0;  } |

**Strings**

* **C has no built-in string type like Java**
* In C, a string is an **array of “chars**”, terminated by a **\0 (null) character**

|  |
| --- |
| **char \*name = “Bob”;**  or  **char name[] = “Bob”;** |



|  |
| --- |
| **// print text- fast way**  printf(“%s \n”,name);  // Slow way of printing text - traversing array  while(\*name){  putc(\*name1++,stdout);  } |

**The char value ‘\0’ is guaranteed to be all zeroes.**

**String manipulation functions**

* The library **<string.h>** contains common string manipulation functions
* **Programmer’s responsiblity**: to ensure sufficient memory is allocated and available for strings
* **All strings must be null (\0) terminated,** otherwise string.h functions may fail
  + Keeping searching until they find a null and thus might cause them ***to go outside of the allocated memory***
    - Causing crash or security problem

**string.h functions**

* Must include <string.h> library

|  |  |
| --- | --- |
| function | Meaning |
| char \*  **strcpy(char \*dst, char \*src);** | copy from src to dst |
| char\* strcat(char \*s, char \* append); | append the string append to s |
| char \*  strchr(char \*s, int c); | locate c in s |
| int  strcmp(char \*s1, char \*s2); | **compare s1 and s2** |
| size\_t  strlen(char \*s); | return length of s (not including \0) |

**Reading the command line**

The main function has two parameters:

* **int argc** : count number of arguments
* char \*argv[] – array of strings – each argument



|  |
| --- |
| **/\* argv.c -- print out the command line arguments \*/**  #include <stdio.h>  int main(int argc, char \* argv[]) {  int i;  for (i = 0; i < argc; ++i)  printf("argv[%d] = %s\n", i, \*argv++); // argv[i]  return 0;  }  // Output:  argv[0] = ./argv  argv[1] = -o  argv[2] = test  argv[3] = -w  argv[4] = data.txt |

**options.c**

|  |
| --- |
| /\* options.c -- decode command line options \*/  #include <stdio.h>  #include <stdlib.h>  #define MAX\_HEADS 255 /\* maximum number of -h options \*/  typedef short bool;  #define true 1;  #define false 0;  /\*  \* usage -- print usage string to the stderr stream and exit  \*/  void usage(char \* pname) {  fprintf(stderr, "usage: %s [-a] [-h header] infile outfile\n", pname);  exit(-1);  }  int main(int argc, char \* argv[]) {  char \* infile, \* outfile, \* pName = \*argv;  char \* header\_str[MAX\_HEADS];  int i, num\_heads = 0;  char \*s;  bool append = false;  /\*  \* decode command line arguments  \*/  while (--argc > 0 && (\*++argv)[0] == '-')  for (s = argv[0] + 1; \*s != '\0'; s++)  switch (\*s) {  case 'a':  append = true;  break;  case 'h':  if ((header\_str[num\_heads] = (\*++argv)) == (char \*)NULL)  usage(pName);  else  --argc;  if (++num\_heads >= MAX\_HEADS)  fprintf(stderr,"%s: too many headers (max %d)\n",  pName, MAX\_HEADS);  break;  default:  fprintf(stderr,"%s: illegal option %c\n", pName, \*s);  usage(pName);  break;  }  if (argc != 2)  usage(pName);  else {  /\* set infile and outfile to last 2 arguments \*/  infile = \*argv;  outfile = \*++argv;  }  /\* now print out options and files entered \*/  printf("You entered: %s ", pName);  if (append) printf("-a ");  for (i = 0; i < num\_heads; ++i)  printf("-h %s ", header\_str[i]);  printf("%s %s\n", infile, outfile);  /\* ... \*/  return 0;  } |

**typedef**: give a type a “name” (give alias)

* **typedef <existing\_name> <alias\_name>**

Example 1: define “BYTE” for 1 byte numbers

* **typedef** unsigned char **BYTE**
* **BYTE b1, b2;** // short abbreviation

Example 2: Define data types

<https://overiq.com/c-programming-101/typedef-statement-in-c/>

|  |
| --- |
| #include <stdio.h>  #include <string.h>  **typedef struct Books** {  char title[50];  char author[50];  char subject[100];  int book\_id  } **Book**; // Book Is an alias  int main() {  Book book;  strcpy( book.title, “C Programming”);  strcpy( book.author, “Student”);  strcpy( book.subject, “C Tutorial”);  book.book\_id = 123;  printf(“Book title: %s\n, book.title);  printf(“Book author: %s\n”, book.author);  printf(“Book subject: %s\n, book.subject);  printf(“Book book\_id: %d\n, book.book\_id);  return 0;  } |

|  |
| --- |
| **struct** book  {  char title[20];  char publisher[20];  char author[20];  int year;  int pages;  };  **typedef** **struct** book Book; |

## Lecture 5: User defined data structures

Define you **own data structures** using:

* structs
* typedef

Managing memory

* allocating
* deallocating

Dynamic data structures

**Structs**:

* **Bundle data of the same type together**, use an **ARRAY**
* **Bundle data of different types together**, use a **STRUCT**

**char \*strcpy(char \*dest, const char \*src)**

|  |
| --- |
| Declaring a struct |
| struct person {  int id;  char name[50];  }; |

|  |  |
| --- | --- |
| Using a struct |  |
| struct person fred;  fred.id = 12345;  strcpy(fred.name,”Fred”);  struct person \*fredptr;  fredptr = &fred;  int freds\_id = fredptr->id; | **ptr-> member** and **(\*ptr).member** are the same thing  To access members of a struct:   * Use “.” Dot operator * Via pointer, use -> operator |

**Defining types using typedef**

Define your own type

* Why? Tedious typing “struct xxx”

|  |
| --- |
| // Define type  // tells the compiler that “***StructType***” is alias for “***struct mystruct***”  typedef struct mystruct StructType  // Usage: Declare variable of type “struct mystruct”  StructType a\_struct  // Typdef struct at same time of declaration  **typedef struct {**  **char string[10];**  **int number;**  **} StructType** |

**Alias for in-built types using typedef**

* Example: **typedef unsigned char Byte**
* Aid portability: (different sizes for data type depending on machine)
  + 64 bit machine, use **typedef int Int64**
  + 32 bit machine, use **typedef long Int64**

**Allocating memory**

Example:

* Declare struct to contain pointers

|  |
| --- |
| typedef struct {  char \*first;  char \*middle;  char \*last;  } Name; |

* Declaring an ***instance of Name*** won’t allocate space to store ***first, middle and last***
  + It will only allocate **space to store pointers to these items**
* Manually allocate space. (e.g. request o/s to allocate memory)

**Allocating Memory**

* The **malloc(size)** function **allocates** **size bytes of memory** and returns a **pointer to the allocated memory (returns pointer to newly/requested allocated memory)**
* **malloc()** returns a **NULL pointer** if there is an error (e.g. out of memory)
* **malloc()** returns a **void pointer** which must be cast to the appropriate type

|  |
| --- |
| **//Function prototype:**  void\* malloc(size\_t size)) |
| **// Allocate space for 10 integers**  int \*i = (int \*)malloc( sizeof(int) \* 10 ); // sizeof() gives size of bytes of any type |

**Text, Data, Bss, Stack and Heap**

|  |  |
| --- | --- |
|  | * Each process has its own address space. * Code and static variables in the bottom (text, data and bss) * Above that, heap (grows upwards) * At the top, right under the kernel, stack (grows downwards)   **Text Segment**: contains program code  **Data Segment**: contains global variables  **BSS**: contains variable declared static |

**The Stack**

|  |  |
| --- | --- |
| process stack.png | A function call results in a *new frame being added to the bottom of the stack*.  Stack Frame contains:   * Arguments * Space for local variables * Call linkage (values of stack pointer and program counter) |

**Stack allocation and deallocation**

* Allocation happens on function call (exception for **alloca(**) )
* Deallocation automatically happens on function return
* O(1) operation

**The Heap**

* The OS uses the ***heap to provide memory malloc()***
* Heap Management:
  + Keep track of ***list of unused chunks and their sizes***
  + when **malloc()** call is made, find a chunk big enough and return that
  + when **free()** is called, place the **chunk back** on the list
* Not constant time O(1)
* Fragmentation is possible

**Releasing memory using free() function**

* Memory allocated **during run-time** must be *release back to the O/S.*
* *Failure to free dynamic memory results in* ***MEMORY LEAK*** – a common bug in c
  + Memory leak consumes resources, prevents other programs from using free memory and result in system slowdown
* Syntax:
  + **free(i);**
  + /\* i points to mallocated memory \*/

|  |
| --- |
| // Reverse.c  **#define MAX\_BUFFER 1024 /\* maximum line buffer size \*/**  **/\***  **\* read\_line**  **\* read a line from fp and allocate memory to store**  **\* removes the '\n' at the end of the line**  **\*/**  **char \* read\_line(FILE \*fp) {**  **char buffer[MAX\_BUFFER];**  **char c, \* line = 0;**  **int idx = 0;**  **/\* read in a line and remove trailing \n \*/**  **while ( idx < MAX\_BUFFER - 1 && (c = fgetc(fp)) != EOF && c != '\n' )**  **buffer[idx++] = c;**  **buffer[idx] = '\0'; /\* terminate string \*/**  **/\* now allocate space for the line and copy \*/**  **if ( (line = (char \*)malloc(sizeof(char) \* (strlen(buffer) + 1))) != NULL )**  **strcpy(line, buffer);**  **return line;**  **}**  **int main(int argc, char \* argv[]) {**    **FILE \* fp;**  **char \* line;**  **if (argc != 2) {**  **fprintf(stderr, "usage: %s file\n", \*argv );**  **exit(-1);**  **}**  **if ( (fp = fopen(argv[1], "r")) == NULL ) {**  **fprintf(stderr,"%s: error: can't open %s for reading: ", argv[0], argv[1]);**  **perror(0); /\* perror prints the system error message \*/**  **exit(-1);**  **}**  **while (!feof(fp)) {**  **line = read\_line(fp); /\* read a line \*/**  **print\_reverse(line); /\* reverse the line \*/**  **free(line); /\* free the allocated space \*/**  **}**  **/\* close the file \*/**  **fclose(fp);**  **}** |

**Freeing memory**

* **C has automatic garbage collection**
* Any allocated space must be freed when no longer required
* For compound data structures that refer to pointers to other data structures (e.g. lists, tree and graphs), **we need to “unwind” the data structure, freeing from the deepest element backwards**.

|  |
| --- |
| void **free\_VehicleRec**(**VehicleRec \* vrec**) {  **free(owner\_name);**  **free(registration);**  **free\_OwnerHistoryRec(vrec->ohr);**  ***free(vrec);***  } |

**NULL Pointers**

Trying to ***de-ference a NULL or freed pointer*** may result in bad things

|  |
| --- |
| // **vr doesn’t point to a valid VehicleRec**, so it and ***(its members) cannot be de-reference***.  VehicleRec \* vr;  // Causes Program to Terminate  // **Segment Fault: attempt to access memory that is not allow to access (os has not allocated memory to program)**  vr->owner\_name = (char \*) malloc( ….); |

**Solution: Test whether the pointer is pointing to NULL**

|  |
| --- |
| **if (vr != NULL)**  **vr->owner\_name = (char \*)malloc(…);**  **if(vr->owner\_name)**  **printf(“Owner: %s\n”, vr->owner);**  free(vr->owner);  vr->owner\_name = (char \*)NULL; // OK |

You should not dereference a pointer that has been freed:

|  |
| --- |
| free(vr);  free(vr->owner\_name); // wrong – vr doesn’t point to valid VehicleRec anymore |

**Pitfalls:**

* C runtime does not provide “internal memory” protection

char \*evil\_ptr;

printf("Enter a memory address: ");

**scanf("%d", & ((int) evil\_ptr));**

**printf("%s\n", evil\_ptr);**

* Memory Leaks – cause system to run out of memory
* **Valgrind** – program to detect memory leaks and out of bound array references

**Linked List in C**

Link List stores: An element with a pointer to :

* the next element in the list or NULL at the end
* -> operator dereferences an individual data member through a pointer to struct

|  |
| --- |
| typedef struct **list\_node** {  int data;  **struct list\_node** \*next;  } **ListNode // new type called ListNode (aka struct list\_node)**  ListNode \*ln = (ListNode \*)malloc(sizeof(ListNode));  **ln->data = 0;**  **ln->next = NULL;** |

// Another struct stores pointers to the head of the list and the **current position in the list**

**typedef struct** {

ListNode \*head;

ListNode \*previous;

ListNode \*current;

} **LinkedList**

|  |
| --- |
| LinkedList \*l = (LinkedList \*) malloc(sizeof(LinkedList));  **l->head = l->previous = l->current = NULL;** |



**List functions**

A **set of functions** operate on a LinkedList to **insert, delete and iterate** through the data.

* LinkedList \* new\_list();
* void free\_list(LinkedList \*l);
* void insert\_node(LinkedList \* l, DataType d);
* void delete\_node(LinkedList \* l);
* DataType get\_current(LinkedList \* l);
* bool next\_node(LinkedList \* l);
* bool at\_end\_of\_list(LinkedList \* l);
* void set\_head(LinkedList \* l);

**new\_list()**

* allocate space for an *empty list and initialize data members*
* equivalent ***to constructor in Java; \_\_init\_\_ in Python***

|  |
| --- |
| linkedList \* new\_list()  {  LinkedList \* l = malloc(sizeof(LinkedList));  if(l != NULL) // memory allocation was successful – no error as NULL  {  l->head = NULL;  l->current = NULL;  l->previous = NULL;  }  return l; // return list list    } |

**free\_list()**

* Dynamic memory used by the list must be explicitly freed by the program

|  |
| --- |
| **void free\_list(LinkedList \*l)**  {  assert(l); // assert ensures a condition is true, otherwise the program terminates  set\_head(l);  while(l->current) {  l->current = l->current->next;  free(l->head); // free each node  l->head = l->current;  }  free(l); // free the LinkedList  } |

**insert\_node**

|  |
| --- |
| void insert\_node(LinkedList \* l, **DataType d**)  {  assert(l);  ListNode \* ln = malloc(sizeof(ListNode));  if (ln) {  ln->data = d;  if (l->previous)  {  ln->next = l->current;  l->current = ln;  l->previous->next = l->current;  }  else {  ln->next = l->head;  l->head = ln;  l->current = l->head;  }  }  } |

**delete\_node**

|  |
| --- |
| void delete\_node(LinkedList \* l)  {  assert(l);  if (l->current) {  if (l->current == l->head) {  l->previous = l->head->next;  l->head = l->head->next;  free(l->current);  l->current = l->head;  }  else {  l->previous->next = l->current->next;  free(l->current);  l->current = l->previous ->next;  }  }  } |

**Multi-dimensional arrays**

* Pointers and dynamic memory can be used for multi-dimensional arrays of arbitrary size

|  |
| --- |
| typedef **int** **DataType**;  DataType \*\* myArray; // pointer to a pointer (or array of arrays) |

****

**Dynamic Multidimensional arrays**

* **First allocate space for the outer dimension**
* **Then fill the inner dimension**

|  |
| --- |
| **DataType\*\*** create2Darray(int dimx, int dimy, DataType init\_value)  {  int i, j; /\* counters \*/  DataType \*\*array;  **// Allocate an array of pointers – sizeof(int)**  array = (DataType\*\*)malloc(sizeof(DataType \*) \* dimy);  assert(array);  for (i = 0; i < dimy; ++i) {  **// Allocates an array of DatatTypes sizeof(int)**  array[i] = (DataType \*)malloc(sizeof(DataType) \* dimx);  assert(array[i]);  for (j = 0; j < dimx; ++j)  array[i][j] = init\_value;  }  return array;  }  **// [row][column] corresponds to [inner][outer]** |

**Release Memory – Free in reverse (inner/deepest) and then outer**

|  |
| --- |
| void **free2Darray**(DataType \* \* array, int dimy) {  int i;  for (i = 0; i < dimy; ++i)  free( array[i] ); /\* free each row \*/  free(array);  } |

**Arrays and Pointers**

What is the ***difference*** between the following?

* int a[10][20]
* int \*b[10];

**Conclusion**:

* a[3][4] are both legal references to a single int
* a is true 2D array ( 200 int sized locations)
* b **allocates 10 pointers, but does not initialise them**
  + **Need to allocate the rows before they can be used**
  + Rows may be of variable length

**Array of Pointers**

* char \* argv[] / \* won’t know at compile time \*/
* char \*name[] = { “illegal month”, “Jan”, “Feb”, …, “Dec” };
* malloc() fails gracefully if allocation is not possible

**Other Memory Management Functions**:

* calloc – like malloc, but initializes allocated bytes to zero
* realloc – resizes a malloc/calloc’d memory chunk
* alloca – allocates space on stack rather than heap (so automatically deallocated at end of function)
* Useful function prototypes:
  + void \* calloc(size\_t count, size\_t size);
  + void \* valloc(size\_t size);
  + void \* realloc(void \* ptr, size\_t size);
  + size\_t malloc\_size(void \* ptr);
* calloc allocates space for count objects of size, and sets it all to zeroes
* realloc changes the size of the space already allocated by ptr

**crl-C ( exit the terminal)**

***crd-D( send EOF ) end of file character and quits***

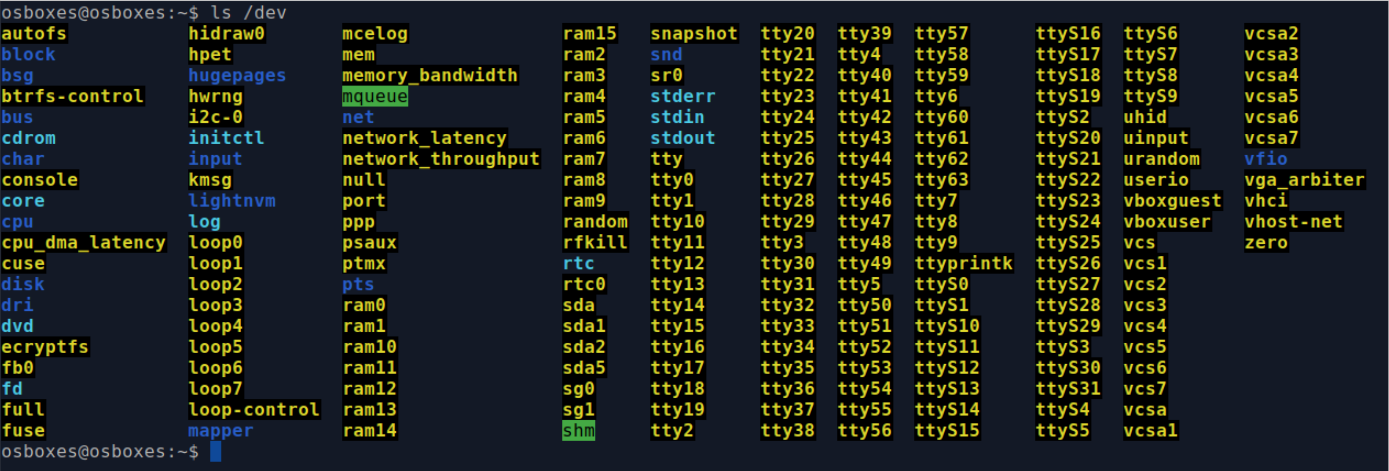
## Lecture 7: Files

**Linux Perspective - Files**

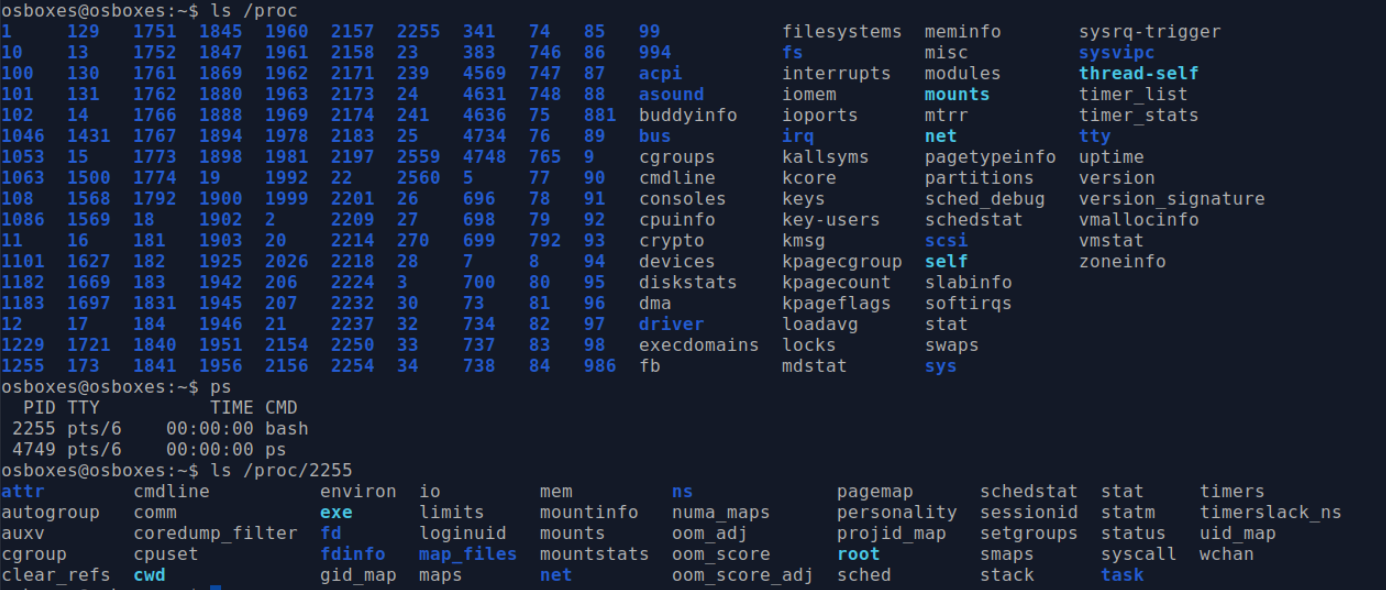
* In Unix and Linux**, everything is a file**
* **A file is a sequence of bytes**
  + Icon and extensions are irrelevant in Unix
* **Most things in the O/S act** as either ***(1) sources or (2) recipients of sequences*** **(streams) of bytes**
  + e.g. keyboards, hard disks, network ports, printers, etc
  + “stream”

**Linux File System**

* Many physical devices are mapped into the file system
  + Drivers are treated as files
  + The directory **/dev** contains the physical devices



The “**ps” command gives information about currently running processes**. This command gets its information from **/proc directory**



**File Concepts**

* A **file** is a *sequence of bytes*
* A file has name by which it can be accessed
* There are many *different* ***disk formats (i.e. ways of mapping files onto physical storage hardware***)
  + Programmers shouldn’t care about disk formats
* Steams, Opening/Closing, binary, file cursor, EOF, permissions, ls -l

**Text Files and Binary Files**

* No distinction between text files and binary files
* File functions in C reside in the <stdio.h> library

**Using files in C**

Basic Operation:

* fopen() - open the file
* fclose() - close the file

Files:

* At OS Level, each **opened file** is **represented** by a ***file descriptor***
* OS has a ***limit on the # of file descriptors that a process can own (e.g. max # of files that can be opended)***
* Must close opened files, to avoid reach the upper limit on opened files/file descriptors
* File I/O may be buffered

Type:

* Important Type: **File \***

What is File \*?

* **File\*** (opened file object) is a ***struct defined in stdio***
* **Stdin, stdout and stderr** are all FILE\*

The File\*:

* Keep track of the file that is open
* Keeps track of the current position in the file

**Opening Files**

FILE \*fopen(const char \*path, const char \*mode);

* Returns a
  + “pointer to the opened file if it succeeds” OR
  + NULL if it fails
* **Path** - absolute/relative path of your file
* **mode** - tellsthe OS how to open your file

**fopen modes**

|  |  |
| --- | --- |
| **String** | **Meaning** |
| r | Open for reading |
| r+ | Open for reading and writing |
| w | Open for writing. If already exists, truncates to zero length; otherwise creates new file. |
| w+ | Open for reading and writing. If already exists, truncates to zero length; otherwise creates new file. |
| a | Open for writing at end of file (i.e. append) |
| a+ | Open for reading and writing – will read from start of file, write at end of file |

**Reading/Writing - Byte by Byte**

* Byte wise operation
  + int fgetc(FILE \*stream);
  + int fputc(int char, FILE \*stream);
* These functions read/write to fiven file one byte at a time
* Both functions return **EOF** on error
  + fgetc() returns EOF when it encounters end of file
  + EOF is a int. That why fgetc() returns an int rather than unsigned char
  + EOF is #defined as “marco” in stdio.h

**Reading/Writing - Block at a time (Buffer)**

A **block** is a *chunk of data of a given size*

* e.g. reading into/writing from a fixed size buffer in memory

Functions: ***fread()*** and ***fwrite()***

Signatures:

* size\_t fread(void \*ptr, size\_t size, size\_t nmemb, FILE \*stream)
* size\_t fwrite(void \*ptr, size\_t size, size\_t nmemb, FILE \*stream)

Parameters:

* **ptr** - start of address of chunk to read into/write from
* **size** - the size of each member/chunk
* **nmemb** - the number of members to read/write
* **stream**- is the file

Useful for arrays:

* 100 **MyStructs** stored **the\_array**, can write them all to **outfile** like this:

fwrite(the\_array, sizeof(MyStruct), 100, outfile)

**Writing strings to files**

Signature: ***int fprintf(FILE \*stream, const char \*format, …);***

Examples:

* fprintf(stdout, …)
* fprintf(stderr, “error message”);

**Reading strings from files**

Signatures:

* **char \*fgets(char \*s, int size, FILE \*stream);**
* int fscanf(FILE \*stream, const char \*format, …);

fgets():

* reads up to size-1 ***characters*** into s
* Stops if it hits EOF or reads a new line (\n)
* Example:
  + **char buf[BUFSIZE];**
  + **fgets(buf, BUFSIZE, infile);**

fscanf() :

* uses a printf like format string and does parsing for you
* if using %s, will only read up to the next whitespace(i.e. reads a word at a time, not a line at a time)
* **Unlike fprintf, the arguments need to be pointers**
* **Argument: strings (i.e. char \*) are already pointers**
  + If passing a numeric type/char, ***need to pass the address***
  + If passing a string or array, they are pointers already
* Example:
  + **int i;**
  + **char c, string[STR\_LEN];**
  + ***fscanf(infile, “%d %d %s”, &i,&c, string);***

**File Functions**

The file-oriented string I/O functions all have counterparts that read from stdin or write to stdout

* fprintf/printf
* fscanf/scanf
* fgetc/getc
* fputc/putc

**gets() should never be used**:

* It doesn’t specify maximum size to read from keyboard
* Buffer overrun bug

**Are we at end of file?**

* Use **feof()**: ***int feof(FILE \*stream)***
  + Non-Zero Value if reached EOF
  + Zero if not EOF
* Can get more information from kernel using ***stat() system call***
  + Linux. Get info including file creation/access/modification dates, size, owner’s UID/GID,etc

## Lecture 8: Structuring Project - MakeFiles

Outline:

* Break down project into multiple files
* **Using “make” to build large projects**

**Anything that causes the compiler to generate code should go in the .c file**

* Function bodies
* Global variable declarations

Anything that is intended as message to compiler are stored in .h file

* Function prototype
* Struct definitions
* Typedefs
* #defines

Only .h files should ever be #included - never #include .c files

* end up with duplicate function and variable names

C Documentation

* <https://www.kernel.org/doc/Documentation/CodingStyle>

**How to build complex C Program**

* Standard gcc -o foo foo.c bar.c approach does not scale
* Solution: **make**

**Make**

* It is a utility that automates the building of program binaries from source files
* **make** requires a “makefile” that specifies the ***source and targets*** to build
* The basic format for a **makefile** is:
  + **target: dependencies**

**commands**

**commands**

**A “make” target:**



**How make “target” works**

* Check whether the target exists
* Check whether dependencies exist
  + If any dependency doesn’t exist, recursively invoke “make” to build it
* Check to see whether any **dependencies are newer than the target**
  + If any dependencies are newer than “target”, build the target
  + Otherwise, do nothing: **target** is already update to date

**Make Variables:**

* Macro replacement
* Must put variable name in parentheese if more than character
* Syntax similar to shell variables
  + = set
  + $ using value
  + $PATH

**Setting variables**:

* **CFLAGS**=`-Wall-O3`

**Using**:

* gcc -o foo.o foo.c **$(CFLAGS)**

**Special Variables:**

* make also has *several* ***automatic*** *variables*
  + You don’t set these (e.g. $? In bash)
* Useful **Automatic variables**:
  + **$@ - filename of the target**
  + $< - filename of the first pre-requisite
  + $? - space separated list of list pre-requisites that are more recent tthan the target
* Example:

|  |
| --- |
| **backup: a.c b.c c.c d.c e.c**  cp **$?** backupdir  **touch backup** |

**Make Example**

|  |
| --- |
| #standard variables used by built-in rules  CFLAGS = -Wall -g  RCS = slen.c  #other variables  OBJS = slen.o recstrlen.o  PROGRAM = slen  # $@ is an built-in variable that expands to the target name  $(PROGRAM): $(OBJS)  gcc -o $@ $(OBJS)  clean:  rm -f $(PROGRAM) $(OBJS) |

**Make Pattern Rules**

* Define Pattern rules (for repetitive code - e.g. make .o files from .c files)

|  |
| --- |
| %.o : %.c  $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@ |

* Can use these pattern rules for any transformation you would usually do on the command line
  + building PDFs from LaTeX source files
  + converting image files from JPG to PPM with ImageMagick

## Lecture 9: Debugging with gdb

Crummy Debugging/testing toolchains and hardware vendors who overpromise and underdeliver have been part of the IT scene for a VERY LONG TIME.

**Errors in C:**

* Buffer overruns or misuse of pointers
* C lacks safeguards:
  + It doesn’t hide implementation details from the programmer
  + It doesn’t prevent you from attempting to access memory that’s not yours ( Segmentation Fault or SegFault)

**What is segmentation fault?**

* Attempt to access memory that has not been allocated to it
* The OS manages the memory: It won’t programs read/write to memory segments it don’t own

**Defensive Programming Techniques:**

* Check return values from functions (e.g. malloc() and fopen())
* Check that pointers are not null, before you deference
  + if(myptr)

\*myptr = 5;

* Initialise pointers to null on creation

Logging statements: using print statements to debug errors

**Gdb**

* A debugger is a program that allows the programmer to run a program and observe its running state
* Gdb - GNU Debugger

Steps:

* Compile program with **-g flag**
* Start program:
  + gdb ./myprogram
  + Enter “**run**” into the command prompt
* If you program has command line arguments:
  + Enter “**run abcde**” where abcde is passed as a command line argument
  + Alternative: Before “run”, use “set args”

|  |
| --- |
| // Debug Approach  gcc –Wall –g –o myprog myprog.c gdb ./myprog run abcde 1234  // Normal Execution  gcc –o myprog myprog.c  myprog abcde 1234 |

**Fixing Bugs**

* After the program crashses, enter “**where**”
* Gdb will show line number and name of source file

**Ser Breakpoints using** “break” command

* Break at line 5 of current file: **break 5**
* Break at line 5 of main.c: **break main.c:5**
* Break at start of myfun() in utilities.c:
  + break utitilies.c:myfunc

**Delete breakpoint**, use “clear”

* clear 5 (delete breakpoint at line 5)

**Tracing code**:

* **step**: go to next line, stepping into any functions that the current line calls
* **next**: go to the next line in this file, stepping over any functions that the current line calls
* **finish**: finish the current function and break when it returns

use “**print**” to see values of any expressions/variables

**Conditional breakpoints**

* to break at a specific line or function, use break…if  
   break main.c:23 if i < 0
* to break whenever a given variable is changed, use watch  
   watch i

## Lecture 11 - Processes

**What is a Process?**

* An **instance of a program** that is ***current loaded into memory*** and being run

(An instance of a program that is currently running the memory)

* **The OS manages the processes (e.g allocate resources CPU/Memory/files to processes)**

**Processes and programs**

* Each process is running **1 copy of the program** at any given time
* It is possible to have multiple instances of the same program running at the same time
* **Process Table**: stores information about the state of each process
* **ps (process status) command** shows the **current state of the process table**

**The process table**

* PID - A **unique** process ID
* UID - User ID of the owner of the process
* PPID - PID of the parent process
* TTY (terminal ID) - associated with the process, if it’s running in console mode

“**ps -ef**” lists all processes currently running on your computer

**Creating processes in C**

* The **fork()** function creates a new process

|  |
| --- |
| #include <unistd.h>  **pid\_t pid = fork();** |

* The **new process starts as a *copy of the parent process***
  + ***Same code***
  + *Same environment*
* After executing ***a fork()***, how can you tell if you’re in the parent process or child process?
  + **fork() *returns the child’s PID to the parent process***
  + **fork() *returns 0 to the child process***

**Parents and Children**

* After a fork(), the parent and child processes have similar state
  + The child’s variable are copies of the parent’s
  + The child’s program code is the same as the parent’s
  + The child has the same ***stdin,stdout and stderr*** file descriptors as the parents
* There are differences:
  + **Parent and child don’t share their memory**
  + **The child process has its own UID**
  + The child process PPID is the same as the parent’s PID
* OS Kernel, will manage the parent and child process separately

**Loading a new program**

* You want the child process to load up a different program
* Use **exec()** family of functions
* Example: Suppose you want the child process to run the command “**ls -alR**”

**char \*parms[3] = { “/bin/ls”, “-alR”, 0};**

**execv**(“**bin/ls**”, **params**);

* + “0” indicator - end of the parameters because execv doesn’t know where the end is.
  + 0 also defined as NULL

**Command Shells**

|  |
| --- |
| **while user has not quit:**  display prompt  read line from keyboard  establish that first “word” in line represents an executable program  **if fork() == 0:**  **execv(program, arguments)**  else wait for child to finish |

**$PATH**

* Mystery: “**ls**” is the same as “**/bin/ls**”
* **$PATH** is an ***environment variable*** that ***stores list of directories to search***
  + echo $PATH // display current contents

**Environmental variables**:

* In **Unix/Linux**, main() can have UP TO 3 arguments:
  + int main(int argc, char \*argv[], **char \*envp[]);**
  + This won’t work under every OS (not portable)
* Portable Access to environmental variables, use *getenv()*
  + char \*path = **getenv**(“PATH”);
* execvp() automatically searches $PATH

**Waiting for the child to terminate**

* The **last step** of the program is to wait for the child process to terminate
* **waitpid()** function:

|  |
| --- |
| int status;  pid\_t pid = fork();  // If current process is parent: wait for child  if(**pid**){  **waitpid**(pid, **&status**, 0);  } |

* **waitpid()** causes the *parent to suspend operations* until the *child process is terminate*
* Information about what cause the child to terminate goes into “status”

**Zombies**

* After a process terminates:
  + the OS keeps track of its status in the process table,
  + which may be used when the *parent calls waitpid()*
* If the **parent process** **does not perform** a **wait(),** then this information is never cleared out *and the process hangs around and taking up a slot in the process table*
  + Size of process table is limited
  + **Zoombie Processes**: **A process whose execution is completed but it still has an entry in the process table.**
  + Zombie processes usually occur for child processes, ***as the parent process still needs to read its child’s exit status***
  + **Wait() - read’s the process exist status and clears the process from the process table**
* Good practice for processes to either: ***wait() or waitpid()*** after sprawning child processes so that the process doesn’t end up cluttered with zombies.

**Threads and Processes**

* The use of **fork() and exec()** under Linux is the same as multi-threading
* Difference b/w a process and thread
  + **OS** does not *treat threads as separate processes*
  + New-spawned threads operate in the same address space as their parents
  + The OS doesn’t protect threads’ memory form the actions of other threads in the same process
* p**threads** library

Communication:

* Processes communicate to each other using IPC (InterProcess Communication)
* Threads by default share their memory, so communication is easy
  + Need to protect data from simultaneous access

## Lecture 12 - Job Control and Signals

**Running processes in the background**

**Tell the shell not to wait for the child process (e.g. finish displaying output) to terminate, before displaying bash prompt**.

Example:

* The bash prompt is displayed before the *ls output* because Bash didn’t wait for the “ls” to finish.
* **Add “&” to the end of the command**
* **ls -alR &**
  + where R - recursive listing

**Stopping and Starting Programs:**

* Crtl-C terminates the **process permanently**
* **Crtl-Z** makes a **process stop temporary**
  + You can make the program run in background using the “**bg**” command
  + **Test this using the “top” command** - which shows current processes that uses the most resources
  + To move “background-process” back into foreground
    - **Use the “fg” command**

**If you have more than 1 background process**, then use **“jobs” command** to list of all background jobs in current bash session

* Move a certain background process into foreground
  + fg %job\_number
  + fg %3

|  |
| --- |
| The output of jobs looks like this:  [1] Stopped nano  [2] - Stopped top  [3] + Stopped nethack  **fg %3 - the command to foreground nethack process** |

**Kill Process:**

* “kill” command
* Use **ps** (process id) or **jobs** (job id)
* Syntax:
  + **kill *-level PID***
  + **kill *-level %jobID***
* Example:
  + **kill -1 153**
  + Send signal 1 (polite) to process number 153
* **Level indicates the signal** you want to send to the process you’re killing
  + **1 - Police**
  + **9 - terminate with extreme prejudice**
  + The kernel also send signals to processes (e.g. seg fault, divide by 0 error, Crtl-C/Z)

|  |  |
| --- | --- |
| **Signal** | **Name** |
| 1 | SIGHUP (hang up) |
| 2 | SIGINT (interrupt, ctrl-C) |
| 3 | SIGQUIT |
| 4 | SIGILL (illegal instruction) |
| 5 | SIGTRAP (trace trap, used in gdb etc.) |
| 8 | SIGFPE (floating point exception, e.g. divide by zero) |
| 9 | SIGKILL (can’t be caught or ignored) |
| 11 | SIGSEGV (segmentation fault) |
| 18 | SIGCONT (continue if stopped, e.g. via fg) |
| 20 | SIGTSTP (terminal stop, ctrl-Z) |

## Lecture 15 - Linux IPC Part 1

**What is IPC?**

* **IPC** stands for ***interprocess communication***
* Allows processes to share data
* Why?
  + Break down complex problems into simpler ones

**Mutual Update Problem**

***Two processes*** are trying to access the same memory concurrently

* Imagine a shared int x = 5
  + process 1 reads x, gets 5
  + process 2 reads x, gets 5
  + **process 1 increments x giving 6**
  + **process 2 increments x giving 6**
  + process 1 stores x back to shared memory
  + process 2 stores x back to shared memory
  + x has been **incremented twice** (by two processes) but **x has** **only increased in value by 1 (**in memory**)**
  + ERROR!

**Approaches to IPC in Linux**

There are different approaches.

* Signals
* Pipes
* Named Pipes (pipes that exist in the filesystem)

**Reference: *The Linux Programming Interface* by Michael Kerrisk, from No Starch Press.**

**Signals**

* Kernel uses signals to tell processes what to do
  + Sends **SIGSEGV** on segmentation fault
  + Sends **SIGFPE** on divide by zero
* Use the “kill” command to send signals manually
  + kill -9 123
  + Send signal 9 (**SIGKILL**) to process number 123
* Very simple form of IPC
  + Target process only knows which signal it has received
  + No other data gets transferred

**Sending Signals in C**

* You can send signals in C using the “**kill()**” function

**#include <signal.h>**

**int kill(pid\_t pid, int signal);**

* Example:
  + **kill(123,9)** equivalent to “kill -9 123” in Bash
  + **kill(123,SIGKILL)** does the same thing as “***names***” for signals are defined in ***signal.h***

**Catching signals in C**

* Customize your ***C program’s*** response to **certain** signals
  + E.g. not all signals - you can’t ignore “kill -9”
* Write a callback function
  + Must take 1 int parameter and return void
  + E.g. ***void sighandler(int signal)***
* **Register call-back using the signal() function**:
  + **signal(SIGINT, sighandler)**
  + tells the program to execute sighandler(SIGINT) when it receives a SIGINT (interrupt signal)

**When to use signals**

* Communication is simple
  + Limited number of different signals
  + Can’t pass a message with a signal
* Order of processing doesn’t matter

**Pipes**

* Simple Concept: hook ***stdin*** of one process to ***stdout*** of another
* Command Line Operator |
  + E.g. send output of **ls** to **wc -l**

Count # of lines in ls (e.g. number of files/subdirectories)

***wc -l*** stands for “***word count, lines***”

**File Descriptors:**

* A **file descriptor** is a ***low-level identifier for a file***
  + The FILE\* data structure contains a file descriptor
* The process table contains a list of all files that process has open
* A **file descriptor** is ***an index into this list***
  + it's just an int
* There are low-level file functions that work on file descriptors: open(), read(), write(), close()

**Pipes in C**

* How this works: *the kernel keeps track of stdin, stdout, and stderr for every process in the system.*
* Create a pipe using the **pipe()** function: ***pipe(int fds[2])***
* If the call succeeds, **the array fds[] is populated with 2 open file descriptors(fds)**
  + fds[0] contains the read end
  + fds[1] contains the write end
* Data written at the write end will be readable at the read end
* Not useful for a single process, but calls to ***pipe()*** are usually followed by calls to ***fork()***
* The process that will that will write *close() fds[0]*, and the process that will read close() fds[1]
* Can read and write using the read() and write()

**Pros and cons of pipes**

* Only allow one-way communication
* Can only communicate between parent and child processes (so not between sibling processes)
* Communication is one-to-one, so not too good for managing families of processes
  + e.g. multiple producers of data and multiple consumers of data
* Pipes contain buffers (so unread data is stored for a time) but they are limited in size
  + size limit differs from version to version of Unix
  + even under Linux, differs between kernel versions
  + for recent Linux kernels, it’s 64Kb

**Named pipes (FIFOs)**

* Pipes work like files
  + They are associated with file descriptors
  + They are accessed by low-level file functions read() and write()
* Pipes don’t exist in the filesystem
  + They don’t have filenames/paths
  + They can only be accessed by parent/child pairs
  + Can only access the file descriptor in the process that created it, or in a process that inherited it via fork()
* **Name pipes** are variation that **does exist in filesystem**

**Creating a named pipe**

* **To create a named pipe**, use the **mkfifo()** function:  
  **int mkfifo(const char \*path, mode\_t mode)**
  + the mode should be in the same format as setting file permissions using **chmod** command
* Example:  
  **mkfifo(“/etc/mypipe”, 0666);**  
  creates a **named pipe /etc/mypipe** that is readable and writeable by all users
* Can also create named pipes using the mkfifo command on the command line:  
  mkfifo /etc/mypipe –m 0666
* Named pipes work just like ordinary ones – you can open them using the open() command

**Limitations of named pipes**

* Can connect unrelated processes (i.e. not parent and child)
* But still only one-to-one
* Data still only available sequentially (i.e. can’t seek back and forth in the stream as with a normal file)
* Attempts to read from an empty pipe will cause the process to block, waiting for a write

## Lecture 16 - Linux IPC Part 2

* Last lecture, we covered:
  + signals (small, cheap, basic)
  + pipes (parent-child only, one-way, one-to-one)
  + named pipes (still one-way and one-to-one)
* This Lecture:
  + **Memory-mapped files**
  + **Shared Memory**

**Memory-mapped files**

* **Basic idea: Use a file as though it were memory**
* **Use the nmap() function:**

**void \*mmap**(void \*addr, size\_t len, int prot, int flags, int fd, off\_t off)

* + **addr** is the destination address for the mapping (or NULL if you don’t mind where it goes)
  + **len** is the size of the mapped area in bytes
  + **prot** allows you to specify protections by bitwise-ORing these flags: PROT\_WRITE, PROT\_READ, PROT\_EXEC (to allow write, read, and execution respectively)
  + **prot** allows you to specify protections by bitwise-ORing these flags: PROT\_WRITE, PROT\_READ, PROT\_EXEC (to allow write, read, and execution respectively)
  + **fd** is the file descriptor of the file you want to map
  + **off** is the offset into the file, or 0 if you want to go from the start

**Accessing mapped memory**

* Once you’ve mapped the file into memory, you can use it as though it were any other memory (e.g. the result of a **malloc()** )
* Here’s an example:

**int fd = open(“/path/to/file”, O\_RDWR);  
(char \*)mapped = mmap(NULL, 4096,  
 PROT\_READ | PROT\_WRITE,  
 MAP\_SHARED, 0);**

**strcpy(&mapped[10], “Hello, world!”);**

* This writes the string “**Hello, world!**” into the /path/to/file, starting at the 11th character

**When you’re finished**

* When you’re done with the file, call ***munmap()*** to ***remove the mapping***:

***int munmap(void \*addr, size\_t length);***

* Trying to access a region you’ve unmapped causes a segmentation fault
* Note that closing the file descriptor associated with that memory does **not** unmap the region
* All of a process’s mapped regions are unmapped where the process terminates
  + Note the similarity with **malloc() / free()**

**Preventing simultaneous access**

* If *multiple processes* are *simultaneously reading and writing* to the same part of the file, might strike problems
* Can use ***synchronization mechanisms*** to prevent this
  + E.g. semaphores, pthreads, mutexes, etc
* Can also use **fcntl()** to lock a region of the file

**Pros and Cons**

* Excellent for loading data into memory quickly (PRO)
* If a file is being stored on disk, access likely to be very slow (CON)
* Synchronization could be a problem if multiple processes need to read and write
* Can use ***file locking*** or ***semaphores***

**Shared Memory**

* **Shared memory** means mapping the *same piece of physical memory(RAM)* into *multiple processes’* memory space
* **Tools**: (1) System V and (2) POSIX
* **System V** is older and therefore supported on more systems
* **POSIX** has a cleaner interface

**How it works?**

* There is a dedicated **virtual filesystem** mounted under **/dev/shm**
* The filesystem actually lives in physical memory ( ramdisk)
* Kernel creates shared memory objects in ramdisk
* Processes map the shared memory blocks into their own address space using the **mmap()** function

**Accessing a shared memory object**

* Similar to accessing a file:
  + open with **shm\_open():**  
     int shm\_open(char \*name, int oflag, mode\_t mode);
  + need to give it a name
  + the **oflag** argument can include:  
     **O\_CREAT** create object if it’s not already there  
     **O\_RDONLY** open read only  
     **O\_RDWR** open read/write
  + these arguments form a ***bitmask***, so if you want to use more than one argument, bit-wise-OR them together:
    - **fd = shm\_open(“mySHM”, O\_CREAT | O\_RDWR, 0666);**
  + the **mode** flag sets permissions

**Setting up a shared memory object**

* Newly-created ***shared memory objects*** have zero-length
  + **fd = shm\_open(“mySHM”, O\_CREAT | O\_RDWR, 0666);**
* Set the Length of the shared memory object
  + **ftruncate(fd,10000);**

Set length of shared memory object to 10,000bytes

**Map into memory**

* **shm\_open()** returns a file descriptor
* Pass this file descriptor to ***mmap()*** - works just like a shared mapped file

**When you’re finished**

* As with ***memory-mapped files***, call ***munmap()*** when you’re finished with the shared region
* Calling the **munmap()** doesn’t delete the shared region
  + The current process can’t touch it unless the process remaps it

**When you’re *completely* finished**

* Reboot: automatically deletes the shared object
* Manually delete a ***shared memory object***:
  + use **shm\_unlink()** function
* **int shm\_unlink(const char \*name);**
  + doesn’t delete it straight away
  + tells the kernel to refuse to allow any more calls to shm\_open() on the object
  + **kernel will delete the object once all processes have unmapped it**
* Can also manually delete the entry in **/dev/shm**, using **sudo rm** on the command line

Problems with shared memory

* Does not prevent mutual update and other race conditions.
* You must prevent with your own implementation (e.g. using semaphores)
* A synchronization mechanism that prevents different processes from simultaneously accessing the same objects in shared memory

## Lecture 17 - Scripting with Bash

**Shell Scripts**:

* **Bash** is a ***programming language interpreter***
* Scripts:
  + List of commands in file that will executed by Bash

**Simple Script**

|  |  |
| --- | --- |
| #!/bin/bash  echo “Hello!” | ***Absolute path to interpreter***  ***Command(s) to run*** |

**Making the Script Run**

**Method 1: Make it an executable using chmod +x myfile**

* **./myfile.sh**
* This will create a bash process which will interpret myfile.sh

**Method 2: use the “source” command**

* **source myfile.sh**
* This tells your current bash to interpret myfile.sh

**Choose the approach that suits your needs**

* If you want to change environmental variables in your current shell, such as $PATH, use source

**Variables**

* **Set with = and access with $**
  + **No spaces allowed around the = sign**
* Can put {} brackets around variable name
  + **basename**=myfile

gcc -o $basename **$**{**basename**}.c

* + This would execute the command

gcc -o myfile myfile.c

* **Only 1 type: string**
* To make visible in subshells, declare with ***export***:
  + **export basename=myfile**

**Quotes**

* Quotes required for strings that contain spaces
  + E.g. myvar=”string with spaces”
* **Double Quotes: Performs Variable substitution**
  + echo “$HOME is my home”
  + Output: /home/ram is my home
* Single quotes: Doesn’t preform variable substitution
  + echo ‘$HOME is my home’
  + Output: $HOME is my home

**Wildcards**

There are ***special characters*** that you can use in the shell to match multiple files

* \* matches any sequence of characters
* ? matches one character
* [ab] matches “a” or “b”
* [a-c] matches “a”,”b” and “c”
* {foo, bar} matches the string “bar” or “foo”

The ***expansion*** of these special characters is ***globbling*** (not the same as regular expressions)

**Getting output from programs**

* **Can set a variable to the output from a command you’ve run**

**nfiles**=**$**(**ls|wc -l)**

Count the number of lines

* **Can also use backticks (same thing):**

nfiles = `ls | wc -l`

**Operators**

Use ***Double parentheses*** for ***arithmetic operations***

* Use the $(( )) arithmetic expansion.

Examples:

* myvar=$((x+2))
* num=$(( $num + $metab))

You can use **all the usual operators**

* arithmetic: +, -, \*, /, % (also \*\* for exponentiation)
* assignment: +=, -=, \*=, /=, %=
* logical: &&, ||, !
* bitwise: &, |, ~, ^, <<, >>

**Short-circuiting**

* and **&&**
  + cmd1 && cmd2
* OR **||**
  + Can be used for simple error handling
  + tar zxvf $TARFILE **||** echo “Couldn’t extract archive”
* Both logical operators short-circuit - their arguments are evaluated from left to right and execution stops, once the results is computed

**Conditional Expressions**

In Bash, conditionals have **square brackets []**

Check the *existence of files*

* [ **-e FILE** ] true if FILE exists
* [ **-f FILE** ] true if FILE exists and is a regular file
* [ **-r FILE** ] true if FILE exists and is readbale

String Operators:

* [ **S1 == S2** ] true if S1 and S2 are equal strings
* [ **-z S1** ] true if S1 has zero length
* [ **-n S1** ] true if S1 has non-zero length
* [ **S1 != S2** ] true if S1 and S2 aren’t equal strings
* [ **S1 > S2** ] true if S1 comes after S2
* [ **S1 < S2** ] true if S1 comes before S2

Danger!

* Solution: Use operators for numbers
* Both “<” and “>” in bash only compare strings in lexicographical order
  + “1000” comes before “50”
* a=1000

b=50

**if** [ $a < $b ]; then

echo “help! Maths is broken!”

**fi**

**Conditionals with numbers**

* [ N1 -gt N2 ] true if ***N1*** is greater than ***N2***
* [ N1 -lt N2 ] true if ***N1*** is less than ***N2***
* [ N1 -eq N2 ] true if N1 is equal to N2
* [ N1 -ne N2 ] true if N1 is not equal to N2
* [ N1 -ge N2 ] true if N1>=N2
* [ N1 -le N2 ] true if N1 <= N2

Note: **Must leave space between the square brackets**

**If statements**

Basic structure: **If**/**then**/elif/else/**fi**

* **The elif and else are optional**

Example 1:

if [ -e $FILE ]

then

echo “File Exists!”

fi

Example 2:

if [ -d $FILE ]; then

echo $FILE is a directory

elif [ -f $FILE]; then

echo $FILE is an ordinary file

else

echo I don’t know what $FILE is

fi

**Note:** The semicolons here aren’t needed if you put the “then” on a new line. **The semicolons are acting as a command separator**

**Loops: while and until**

|  |  |
| --- | --- |
| While Loop | Until Loop |
| i = 0  while [ $i -lt 10 ]; do  echo $i  (( i++))  done | **Terminates when condition becomes true**  i=0  until [ **$i -eq 10** ]; do  echo $i  (( i++ ))  done |

**For Loop**

Similar to foreach loop

Example: Iterate over (whitespace-separated) words in string

|  |
| --- |
| for file in $( ls ); do  echo I can see $file  done |

**Command Line Parameters**

* Command Line parameters are stored in variables ***$1,$2,$3,…, ${10}, ${11}, …***
* The name of the shell script goes in **$0** (different from argv[] in c)
* Other special variables for handling parameters
  + **$#** is the number of command line parameters (cf. argc in C)
  + **$@** is a list of all the command line parameters, separated by a space (so you can iterate over them with a for loop)

**Reading from the keyboard**

***Example: Confirmation from the user***

* Do you really want to delete everything? (Y/N)

Use “**read**”:

|  |
| --- |
| echo “Are you sure? (y/n)”  **read choice**  if [ $choice == “y” ]; then  rm -rf \*  else  echo “That’s probably safest”  fi |

## Lecture 18 - Useful Commands

**echo**

* Equivalent to “print” statement
* Useful Options:
  + ***-n argument*** tells it not to print a trailing newline (userful for prompting input)
  + **-e argument** enables processing of certain ***formatting characters*** when escape with backslash, including
    - \n for new line
    - \t for tab
    - \q for alert

**cat**

* Short for “***concatenate***”
* Takes one or more files as inputs and outputs them one after the other to **stdout**
  + Example:
    - ***cat text1.txt text2.txt***
    - ***cat file1 file2 file3 > bigfile***
  + ***If you don’t provide input files, it defaults to reading input from stdin***
* Often use with I/O redirection
* Options
  + **-n** number each line
  + **-T** show tabs (as ^I)
  + **-e** show line endings with $
* Works on binary files and text files

**head**

* Displays the first few lines of a file (or files)
  + By default, show first 10 lines
  + Reads from ***stdin***, if you don’t ***the filename***
  + output goes to ***stdout***; can capture it with I/O redirection
* Useful for seeing what’s in a file (e.g. header docs)
* Options:
  + -c ***NN*** show first ***NN*** characters
  + -n ***NN*** show first ***NN*** lines
  + -n -***NN*** show everything except the last ***NN*** lines
* Example: Show first 10 lines of errors produce when building the project
  + make 2> errfile
  + head errfile

**tail:**

* Displays the last few lines of a file(s)
  + By Default, show last 10 lines
* -c option
  + -c NN - shows last *NN* characters
* To show all, but the first *NN* lines, use *tail -n +NN*
* **-f** keep file open, continue to display lines as they are added
  + Useful for watching errors pop up in log files
* Example:

tail /var/log/syslog

tail -f /var/log/apache2/error.log

**sort:**

* Sorts its input, line by line
  + By default sorts **stdin**, but can give 1 or more files
  + Example: ls | sort
* Options:
  + **-n** sort numerically (so 50 comes before 100)
  + -r reverse order
  + -u remove duplicates
* Uses mergesort O(N\*logN)

**Sort with multiple fields**

* Can sort tabular data
  + Default column separator is whitespace
* Options:
  + -t***SEP*** define ***SEP*** as column separator
  + -k ***N*** *sort the Nth field (first is 1)*
* Sort on multiple keys
  + E.g. sort by surname, then by first name
  + Use Option “**-k**” multiple times
  + e.g. sort –k 1 –k 2 myfile.txt
* Sort CSV file by its third column
  + sort –t“,” –k 3 myfile.csv

**Dealing with headers**

Phonedir.csv where first line is header

Want to sort the file, but don’t want to lose the header

|  |
| --- |
| name, ID, phone ext John,1235,x55555  Erica,2205,x90993 |

**Solution**:

* Use ***head*** to transfer the header line to ***stdout***
* Use tail to get rest of the file
* Pipe the result to sort

|  |
| --- |
| **head –n 1 phonedir.csv**; **tail –n +2 phonedir.csv** | **sort** |

This line sends everything to stdout, but if you **append >outfile** to the head call and >>outfile to the sort call you can easily redirect it to a file.

**wc**

* Word Count
* Works on Text Files
  + If you give **wc** a list of files, it will display statistics for each file and then a total at the end
  + If you don’t give **wc** a file, **wc** will read from **stdin**
* Options:
  + **-c** count bytes
  + **-m** count characters
  + **-w** count words
  + **-l** count lines (actually counts newlines)

**which and whereis**

* **which** tells you which file will be run in response to a typed command
  + If a file of that name appears in the path, ***its full pathname will be displayed***
  + Otherwise, nothing is output ( and $? is set to 1)
  + Common Use: Display the current version of the running program
* **whereis** can locate executable binaries, sources and manual pages for commands
  + By default, locates all three
  + Can choose which to find using the ***-b, -s and -m*** options respectively

Example - which vs whereis

which python3  
 /usr/bin/python3  
  
 whereis python3  
 python3: /usr/bin/python3.4 /usr/bin/python3.4m /usr/bin/python3 /etc/python3.4 /etc/python3 /usr/lib/python3.4 /usr/lib/python3 /usr/bin/X11/python3.4 /usr/bin/X11/python3.4m /usr/bin/X11/python3 /usr/local/lib/python3.4 /usr/share/python3 /usr/share/man/man1/python3.1.gz  
  
 whereis –m python3  
 python3: /usr/share/man/man1/python3.1.gz

**locate**

* Syntax: locate ***FILENAME***
* Tells you where files and directories
* By default, tells you about every file and directory that contains ***FILENAME*** as part of its name
* Doesn’t search the filesystem. It uses a database prepared by the ***updatedb*** command
  + ***updatedb*** command is run periodically by operating system
  + means that ***locate*** won’t find new files (e.g. recently created files)

**find**

* Searches the filesystem for a file matching your criteria
* Syntax: find *SEARCHPATH CRITERIA ACTIONS*
* Example: find a file called gdbinit under /etc
  + find /etc -name “gdbinit”
* Options:
  + -mtime ***N*** find files last modified ***N*** days ago
  + -ctime ***N*** find files created ***N*** days ago
  + -atime N find files accessed ***N*** days ago
  + ***-N*** means you want files touched wihtn the last *N days*
  + ***+N*** means you want files last touched more than *N days ago*
* The “time” option also has min versions, for specifying intervals shorter than a day
  + find . -mmin -30
  + will find all files in the current directory, including its subdirectories, that were last modified no more than half an hour ago

**More options for find**

* Search by type of file as well
  + **Example 1**: Find all the named pipes on the system

find / -type p

* + **Example 2:** Find all C Source files in **$HOME/src**

find ~/src –type f –name “\*.c”

* By default, prints the full pathname, but you can tell it do other things (e.g. delete)
  + find ~/src/project -name “\*.o” -delete
  + Delete all object files in $HOME/src/proj
* You can use the **-exec** option to run commands on the file
  + a pair of curly brackets “{}” in the command’s arguments will be replaced by the found filename
  + must terminate the command with a semicolon (and you’ll need to escape that with a backslash) – this allows other options to follow the exec
  + so can count all the lines of C source code under the current directory like this:  
    **find . –name “\*.[ch]” –exec ls –l {} \;**

**find vs locate**

* locate is easier to use
* locate is always faster
* find will find files that have been created recently
* find is more powerful
  + e.g. find by owner, permissions, etc

**grep**

* Find text inside files
* Syntax: grep OPTIONS PATTERN FILES
* Simple Usage: Show all the lines in the file(s) that contain the pattern you give it
  + ***grep ‘struct’ \*.h***
  + shows each line containing the word “struct” in the C header files in the current directory
  + Useful options include
    - -v show only lines that *don’t* contain the pattern
    - **-i case-insensitive match**
    - -r recursively search directories
    - -l show only the names of files containing a match (not the lines)
    - -w search for whole word
* The single quotes are optional in this case, but considered good practice prevents bash from interpreting special characters in patterns

**Patterns in grep**

* grep searches for **regular expressions**
* Useful grep syntax  
   . matches any single character  
   ^ matches start of a line  
   $ matches end of a line  
   *p*\* matches zero or more repetitions of pattern *p*  
   (*p*) defines *p* as a pattern (groups characters together)  
   [a-z] matches characters from a to z inclusive  
   [*abc*] matches character *a*, *b*, or *c* [^*abc*] matches any character other than *a* , *b*, or *c*
* Brackets must be escaped with a backslash ‘**\(** … **\)**’ unless you are using extended syntax
  + do this by using the –E option
* Example:  
   grep –E ‘^b(an)a\*’ /usr/share/dict/words  
   ba  
   banana

**GNU coreutils**

* Many of these utilities can be found in the GNU coreutils package
* There are many other programs in coreutils that we have not covered today, e.g.  
   shuf randomly shuffle lines in a file  
   tac like cat, but reverses the file  
   **expand convert tabs to spaces**  
   **unexpand convert spaces to tabs**  
   **yes print a string repeatedly to stdout**
* coreutils also contains a number of other programs that you have seen before:  
  cp, mv, rm, rmdir, mkdir, mkfifo, echo, test, pwd, chown, chmod
* The coreutils are installed by default on nearly all desktop Linux distributions

Other Notes: <https://www.tutorialspoint.com/cprogramming/c_pointers.htm>

* **A pointer is a variable whose value is the address of another variable**, i.e., direct address of the memory location.
* Declaration:
  + int \*ip; // pointer to an integer
  + double \*dp; // pointer to a double
  + float \*fp; // pointer to a float
  + char \*ch; // pointer to a character

|  |
| --- |
| #include <stdio.h>  int main () {  int var = 20; /\* actual variable declaration \*/  int \*ip; /\* pointer variable declaration \*/  ip = &var; /\* store address of var in pointer variable\*/  printf("Address of var variable: %x\n", &var );  /\* address stored in pointer variable \*/  printf("Address stored in ip variable: %x\n", ip );  /\* access the value using the pointer \*/  printf("Value of \*ip variable: %d\n", \*ip );  return 0;  }  // Result:  Address of var variable: bffd8b3c  Address stored in ip variable: bffd8b3c  Value of \*ip variable: 20 // VALUE OF ADDRESS IN MEMORY |

* Define Pointer Variable
* Assign Address of a Variable to the Pointer
* Access the value at the address in the pointer value
  + \* unary operator that **returns the value** of the variable **located at the address**

NULL Pointer

* Constant - Value of zero
* Good Practice, initialise pointers to NULL.

|  |
| --- |
| Live Demo  #include <stdio.h>  int main () {  int \*ptr = NULL;  printf("The value of ptr is : %x\n", ptr );    return 0;  }  // Result:  // The value of ptr is 0 |

* Memory Address at 0 is reserved by the operating system.

TO check for null pointer:

* If(ptr) // succeeds if p is not null
* If(!ptr) // succeeds if p is null

**Arithmetic Pointer:**

Source: <https://www.tutorialspoint.com/cprogramming/c_pointers.htm>

* **A pointer is a numeric value.**
* Perform arithmetic oper

TOREAD:

<https://opensource.com/article/19/5/how-write-good-c-main-function>

A **boolean** is a data type in the C Standard Library which can store true or false. Every non-zero value corresponds to true while 0 corresponds to false.

**typedef enum {false, true} bool;**

<https://www.educative.io/edpresso/what-is-boolean-in-c>