**Contents**

* Logging into our Linux box
* Intro to UNIX/Linux
* Overview of UNIX/Linux editors
* Intro to C

**Logging into your class Linux account**

**Windows users:** First obtain and unzip the SSH Secure Shell Client at

<http://my.cdm.depaul.edu/resources/software/SSHSecureShellClient-3.2.9.zip>

The SSH Secure Shell Client app can be used to connect to the class Linux machine:

* Open the SSH Secure Shell app and click "Quick Connect"
* Enter the Host Name (perko.cdm.depaul.edu) and your User Name (the first 3 letters of your last name OR, for a few of you, the first 2 letters of your last name plus your first initial)
* Enter your password (your student ID number without leading zeros)
* You're now logged in

The SSH app can also be used to transfer files to/from the Linux box.  
  
**Mac or Linux users:** Open a Terminal window and simply type

$ ssh <your user name>@perko.cdm.depaul.edu

(Don't type the $; it represents the command line prompt.) Your user name is as described in the "Windows users" section. For example, I would login with my user name "lperko" as follows

$ ssh lperko@perko.cdm.depaul.edu

To copy a file, say "sample.txt", from your machine to a directory, say "hw1/example2/", on the Linux box do

$ scp sample.txt <your user name>@perko.cdm.depaul.edu:hw1/example2/

To copy a file, say "sample.txt", from directory "hw1/example2/" on the Linux box to the current directory of your machine, do

$ scp <your user name>@perko.cdm.depaul.edu:hw1/example2/sample.txt .

The period (.) refers to the current directory in Unix/Linux.

**The Linux operating system**

For an introductory tutorial on the Linux operating system use <http://www.ee.surrey.ac.uk/Teaching/Unix/index.html>

We summarize below a few key commands.

**To change your password**

Command: **passwd**  
  
Example:

$ **passwd**  
Changing password for stu.  
(current) UNIX password:   
Enter new UNIX password:   
Retype new UNIX password:

**Note 1: Please change your password the first time you log in.  
  
Note 2: The Linux box is for classwork only and you should not assume privacy. All files on this machine may be read by the Prof.**

**Display current working directory**

Command: **pwd**

Description: Print Working Directory; i.e., display the name of the current directory.

Example:

$ **pwd**  
/home/stu

This directory is the home directory of user stu

**To list the content of a directory**

Command: **ls**

Arguments: Optional list of files and or directory names

Options: -l

Description: Lists the specified files and or directories. For a directory, all its files and directories are listed. If no file or directory is specified, the current directory is assumed.

Example: 

$ **ls**  
$ ls .  
$ ls ..  
aci  buj  chs  flo  gha  kim  loo     mom  pen  rak  sar  swe  wuq  
aga  car  chu  for  glo  kin  lperko  mor  pow  rid  sco  ter  yuc  
als  cha  diz  fri  han  kli  mcc     mus  pug  sah  sie  too  zhe  
bow  chh  fin  geo  kha  leg  mir     ome  raf  san  stu  wil  zho  
$ ls -l ..  
total 8  
drwx------.  2 aci    aci      72 Jan  4 08:50 aci  
drwx------.  2 aga    aga      72 Jan  4 08:50 aga  
drwx------.  2 als    als      72 Jan  4 08:50 als  
drwx------.  2 bow    bow      72 Jan  4 08:50 bow  
drwx------.  2 buj    buj      72 Jan  4 08:50 buj  
drwx------.  2 car    car      72 Jan  4 08:50 car  
drwx------.  2 cha    cha      72 Jan  4 08:50 cha  
drwx------.  2 chh    chh      72 Jan  4 08:50 chh  
drwx------.  2 chs    chs      72 Jan  4 08:50 chs  
drwx------.  2 chu    chu      72 Jan  4 08:50 chu  
drwx------.  2 diz    diz      72 Jan  4 08:50 diz  
drwx------.  2 fin    fin      72 Jan  4 08:50 fin  
drwx------.  2 flo    flo      72 Jan  4 08:50 flo  
drwx------.  2 for    for      72 Jan  4 08:50 for  
drwx------.  2 fri    fri      72 Jan  4 08:50 fri  
drwx------.  2 geo    geo      72 Jan  4 08:50 geo  
drwx------.  2 gha    gha      72 Jan  4 08:50 gha  
drwx------.  2 glo    glo      72 Jan  4 08:50 glo  
drwx------.  2 han    han      72 Jan  4 08:50 han  
drwx------.  2 kha    kha      72 Jan  4 08:50 kha  
drwx------.  2 kim    kim      72 Jan  4 08:50 kim  
drwx------.  2 kin    kin      72 Jan  4 08:50 kin  
drwx------.  2 kli    kli      72 Jan  4 08:50 kli  
drwx------.  2 leg    leg      72 Jan  4 08:50 leg  
drwx------.  2 loo    loo      72 Jan  4 08:50 loo  
drwx--x--x. 17 lperko lperko 4096 Jan  4 08:47 lperko  
drwx------.  2 mcc    mcc      72 Jan  4 08:50 mcc  
drwx------.  2 mir    mir      72 Jan  4 08:50 mir  
drwx------.  2 mom    mom      72 Jan  4 08:50 mom  
drwx------.  2 mor    mor      72 Jan  4 08:50 mor  
drwx------.  2 mus    mus      72 Jan  4 08:50 mus  
drwx------.  2 ome    ome      72 Jan  4 08:50 ome  
drwx------.  2 pen    pen      72 Jan  4 08:50 pen  
drwx------.  2 pow    pow      72 Jan  4 08:50 pow  
drwx------.  2 pug    pug      72 Jan  4 08:50 pug  
drwx------.  2 raf    raf      72 Jan  4 08:50 raf  
drwx------.  2 rak    rak      72 Jan  4 08:50 rak  
drwx------.  2 rid    rid      72 Jan  4 08:50 rid  
drwx------.  2 sah    sah      72 Jan  4 08:50 sah  
drwx------.  2 san    san      72 Jan  4 08:50 san  
drwx------.  2 sar    sar      72 Jan  4 08:50 sar  
drwx------.  2 sco    sco      72 Jan  4 08:50 sco  
drwx------.  2 sie    sie      72 Jan  4 08:50 sie  
drwx------.  5 stu    stu    4096 Jan  4 08:52 stu  
drwx------.  2 swe    swe      72 Jan  4 08:50 swe  
drwx------.  2 ter    ter      72 Jan  4 08:50 ter  
drwx------.  2 too    too      72 Jan  4 08:50 too  
drwx------.  2 wil    wil      72 Jan  4 08:50 wil  
drwx------.  2 wuq    wuq      72 Jan  4 08:50 wuq  
drwx------.  2 yuc    yuc      72 Jan  4 08:50 yuc  
drwx------.  2 zhe    zhe      72 Jan  4 08:50 zhe  
drwx------.  2 zho    zho      72 Jan  4 08:50 zho  
$

.    refers to the current directory  
..  refers to the parent directory  
  
The permissions are set so you cannot see each other's files. You may all view stu's files, where I will do class examples.

**Copy a file**

Command: **cp**

Arguments:   existing\_file   new\_copy

Description: Copy *existing\_file* to the file *new\_copy.* If *new\_copy* doesn't exist, create it. If it does exist, first delete its contents.

If *new\_copy* is a directory, a copy of *existing\_file* is created in directory *new\_copy* with the same file name - *existing\_file*.

Example: 

$ cp /home/lperko/public/lecture1/hello.c .  
$ ls  
hello.c  
$ ls -l  
total 4  
-rw-r--r-- 1 stu stu 70 2013-01-07 09:47 hello.c  
$ cp hello.c test.c  
$ ls  
hello.c test.c  
$ ls -l  
total 8  
-rw-r--r-- 1 stu stu 70 2013-01-07 09:47 hello.c  
-rw-r--r-- 1 stu stu 70 2013-01-07 09:53 test.c  
$

**Create a directory**

Command: **mkdir**

Arguments: new\_subdirectory\_name

Description: Create a subdirectory of the current directory

Example: 

$ mkdir hw1  
$ mkdir hw2  
$ ls  
hello.c hw1 hw2 test.c  
$ ls -l  
total 16  
-rw-r--r-- 1 stu stu 70 2013-01-07 09:47 hello.c  
drwxr-xr-x 2 stu stu 4096 2013-01-07 10:00 hw1  
drwxr-xr-x 2 stu stu 4096 2013-01-07 10:00 hw2  
-rw-r--r-- 1 stu stu 70 2013-01-07 09:59 test.c  
$

**Delete Files/Directories**

Command: **rm**

Arguments: file list

Options: -i -r

Description: Removes (deletes) the specified files

-i Ask before removing each file **(Good idea since it is not possible to reverse a removal)**  
-r Recursive remove directories and subdirectories in  
 the list. Files will be removed in these directories  
 first and then the directory itself.**(Useful but dangerous)**

Example:

$ rm test.c   
$ ls  
hello.c  hw1  hw2  
$ rm hw2  
rm: cannot remove `hw2': Is a directory  
$ rm -r hw2  
$ ls  
hello.c  hw1  
$ rm -ri hw1  
rm: remove directory `hw1'? n  
$ ls  
hello.c  hw1  
$

**Rename a file/directory**

Command: **mv**

Arguments: existing\_file new\_file

Options: -i

Description: Renames *existing\_file* to have the name *new\_file* -i Prompts if mv would overwrite an existing file.

Example: 

$ **mv** hello.c test.c  
$ ls  
hw1 test.c  
$ mv hw1 homeworks  
$ ls  
homeworks test.c  
$

**Change the current directory**

Command: **cd**

Arguments: target\_directory

Description: Change the current directory (*working directory*) to the specified *target\_directory*. If no target\_directory is specified, change to the login/home directory.

Example: 

$ **cd** homeworks/  
/homeworks$ pwd  
/home/stu/homeworks  
/homeworks$ cd ..  
$ ls  
homeworks  test.c  
$ pwd  
/home/stu  
$

**Viewing a file**

There are several tools available for viewing files. The most popular is **more** invoked as follows:

$ **more** test.c   
#include <stdio.h>  
  
int main()  
{  
  printf("hello, world\n");  
}   
$

For files larger than your console window, you will need to scroll:

* Space bar to move forward one page.
* b to move backward one page,
* Enter to move forward one line,
* / to search for key\_word in the text,
* q to quit.

To scroll the contents of a file to the screen, you can use the **cat** command.  
  
Example:

$ **cat** test.c  
#include <stdio.h>  
  
int main()  
{  
printf("hello, world\n");  
}   
$

The cornmand cat gets its name from concatenate: it reads its input file names and writes them to the screen. If you give it more than one file name then cat will first the file contents one after another.  
  
Example:

$ cat test.c test.c  
#include <stdio.h>  
  
int main()  
{  
   printf("hello, world\n");  
}   
#include <stdio.h>  
  
int main()  
{  
   printf("hello, world\n");  
}   
$

Redirecting this output to a file

$ cat test.c test.c > test2.txt  
$ more test2.txt   
#include <stdio.h>  
  
int main()  
{  
  printf("hello, world\n");  
}   
#include <stdio.h>  
  
int main()  
{  
  printf("hello, world\n");  
}   
$

would give you a file named test2.txt whose content is the contents of the concatenation of test.c and test.c. This is a good way to add files together.

**Print a File**

To print a file, you need to transfer the file to your PC. You can then print the file as you would any other file on the PC.

**Unix online Manual**

To view the Unix online manual, you need to know the exact name of a manual topic; e.g., fork. In some cases you may not know the exact name. You can try the man command with the -koption (for keyword) 

$ man -k fork  
fork (2) - create a child process  
forkpty (3) - tty utility functions  
vfork (2) - create a child process and block parent

Once you know the name of the manual page for a topic use the man command again with that specific topic:

$ man fork  
FORK(2)                    Linux Programmer's Manual                   FORK(2)  
  
NAME  
       fork - create a child process  
  
SYNOPSIS  
       #include <unistd.h>  
  
       pid\_t fork(void);  
  
DESCRIPTION  
       fork()  creates  a new process by duplicating the calling process.  The  
       new process, referred to as the child, is an  exact  duplicate  of  the  
       calling  process,  referred  to as the parent, except for the following  
       points:  
  
       \*  The child has its own unique process ID, and this PID does not match  
          the ID of any existing process group (setpgid(2)).  
  
       \*  The  child's  parent  process ID is the same as the parent's process  
          ID.  
  
       \*  The child does not inherit  its  parent's  memory  locks  (mlock(2),  
          mlockall(2)).  
  
 Manual page fork(2) line 1

To move around the manual page use:

* Space bar to move forward one page.
* b to move backward one page,
* Enter to move forward one line,
* / to search for key\_word in the text,
* q to quit.

For additional info about UNIX go to [Norman Matloff's intro to Unix](http://heather.cs.ucdavis.edu/~matloff/unix.html)

**Linux/Unix text editors**

To edit your programs, you will need a text editor. A very basic one is nano; documentation is available at <http://www.nano-editor.org/docs.php>. To open a file, say 'file.txt',  for editing using nano, simply type

$ nano file.txt

nano is a basic editor that is not commonly used by Unix/Linux programmers. The two most popular editors in UNIX/Linux are vi and emacs. They are more difficult to master, however. You can learn about vi at  
  
 <http://heather.cs.ucdavis.edu/~matloff/UnixAndC/Editors/ViIntro.html>  
  
and emacs at  
  
<http://heather.cs.ucdavis.edu/~matloff/UnixAndC/Editors/Emacs.html>  
  
Here is a short primer on emacs. To start emacs type

$ emacs

To open a file file.c using emacs, type

$ emacs file.c

You can start editing know. When you want to **save** your file, type  
  
Ctrl-X Ctrl-S (This means pressing the Ctrl key and pressing x and then s, while still pressing the Ctrl key)  
  
To **save a file under a different name**, type  
  
Ctrl-X Ctrl-W  
  
To **exit** emacs, type  
  
Ctrl-X Ctrl-C  
  
Here is a short list of important emacs editing commands  
  
To **move the cursor up, down, left, or right** use the arrow keys.  
  
To **delete a character** pointed to by the cursor, type  
  
Ctrl-D  
  
**You can also use the Delete or Backspace keys if you change the SSH Secure Shell Client settings (under Edit -> Settings -> Keyboard)**  
  
To **cut and paste**, move the cursor to the first character of the part you want to cut and paste. Then type  
  
Ctrl-SPACE (SPACE is the space bar)  
  
Then move the cursor down to the last character of the part you want to cut. Then type  
  
Ctrl-W  
  
to cut the text. You can then**move the cursor to the position where you want to paste**, and type  
  
Ctrl-Y  
  
to paste. You can paste the cut text again and again with Ctrl-Y.  
  
To**exit the Emacs editor**, type  
  
Ctrl-X Ctrl-C.  
  
You should be able to modify the properties of the VT102 terminal emulator in SSH to customize some of the key bindings.  
  
For additional info about emacs go to [Norman Matloff's intro to Unix](http://heather.cs.ucdavis.edu/~matloff/unix.html)

**Introduction to C**

The following is a short intro to C. A good online tutorial is also available at 

<http://www.cprogramming.com/tutorial/c-tutorial.html>

**Compiling a C program and running it**

To compile a single file C program in a file named *hello.c* use the gcc compiler. 

$ cp /home/lperko/public/lecture1/hello.c .  
$ gcc -o hello hello.c

The command gcc (GNU C Compiler) compiles the program hello.c, links it with any standard c library routines called (e.g. i/o routines) and produces the executable file named hello. 

The file *hello* is an executable and could run at the command line

$ hello

if the PATH environment variable includes the current directory (denoted .) . The value of the PATH environment variable is a sequence of directories the shell will look in for executables. To view this sequence, do:

$ echo $PATH  
/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/home/stu/.local/bin:/home/stu/bin

Note the the current directory (.) is not included. To execute hello, you will need to specify where the executable hello is:

$ ./hello  
hello, world  
$ /home/stu/hello   
hello, world  
$

/home/stu/hello is the **absolute pathname** of file hello  
./hello is the relative **pathname**of file hello, and it is relative to directory /home/stu

**From source to executable**

**hello.c** is compiled as follows:

$ gcc -o hello hello.c

[The compilation system](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/compilation.jpg) performs the translation of the source file **hello.c** to an executable **hello** which consists of a sequence of low-level machine language instructions.  
  
To obtain the "preprocessed" source program hello.i, do:

$ gcc -E hello.c

To obtain the assembly program hello.s, do:

$ gcc -S hello.c  
$ ls  
hello  hello.c  hello.s  
$ more hello.s  
    .file    "hello.c"  
    .section    .rodata  
.LC0:  
    .string    "hello, world"  
    .text  
    .globl    main  
    .type    main, @function  
main:  
.LFB0:  
    .cfi\_startproc  
    pushq    %rbp  
    .cfi\_def\_cfa\_offset 16  
    .cfi\_offset 6, -16  
    movq    %rsp, %rbp  
    .cfi\_def\_cfa\_register 6  
    movl    $.LC0, %edi  
    call    puts  
    popq    %rbp  
    .cfi\_def\_cfa 7, 8  
    ret  
    .cfi\_endproc  
.LFE0:  
    .size    main, .-main  
    .ident    "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-4)"  
    .section    .note.GNU-stack,"",@progbits  
$

To obtain the relocatable object file (binary file that is not yet executable because it is not linked to helper library functions) hello.o, do:

$ gcc -c hello.c  
$ ls  
hello  hello.c  hello.o  hello.s  
$ more hello.o  
  
\*\*\*\*\*\*\*\* hello.o: Not a text file \*\*\*\*\*\*\*\*

A binary file cannot be viewed using text viewing tools...

**First C program**

We already saw the classical first program:

[hello.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/hello.c)

**You can get the examples that follow by copying folder lecture1 to your current directory:**

$ cp -r ../lperko/public/lecture1 .  
$ ls  
hello  hello.c  hello.o hello.s  lecture1  
$ cd lecture1  
$ ls  
arithmetic.c  count2.c      flow.c       pointers3.c  power.c  
ascii.c       count.c       hello.c      pointers4.c  sort.c  
blocks.c      data-types.c  loops.c      pointers5.c  sum.c  
cmdline.c     demo1.c       pointers2.c  pointers.c   temp.c

**The basics of a C program**

Consider

[demo1.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/demo1.c)

The **#include** line at the top are similar to an import in Java or Python but they differ significantly. In C, the #include is a pre-compilation directive that instructs the compiler to replace the #include line with the text from the indicated file. This file is called a header file and it contains references to library function executables that the program uses.  
  
The **main** function has some similarity to Java also in that just like a Java program, a C program must have exactly one main function (now that we are in C, we refer to methods as functions from now on). (The Python equivalent to main is the top-level code in a module, i.e. code that is not inside a function or class definition.) The main function can be prototyped to return other types, but void and int are the most common. For this course we will adhere to the most common practice which is to return an int. The usefulness of this returned value is however moot for our purposes since we do no script programming which might examine and use the value returned by main.  
  
**Variable declarations** are similar to Java. However in C, the compiler makes no initializations as in Java. Newly declared variables contain garbage values. An **int** is a 32 bit signed value having a range of about negative 2G to positive 2G (1G = 230). An **unsigned int** is a strictly positive 32 bit value ranging from 0 to 4G. C has floating point types **float** and **double**. It is common practice in C (as in Java) to just use the **double** type with its increased precision for floating point numbers to minimize accumulated rounding errors. The **char** type is an 8 bit signed quantity that stores the ascii code for a character. Thus a char is just an 8 bit int and has an unsigned version also. We will discuss data types some more later.  
  
**Console I/O** can be done via the many **print**, **scan** and **get** functions (see the **man pages).**  
  
Our first C program uses: 

**printf("Please enter an integer: ");**  
 and  
 **printf("x: %d y: %d\n", x,y );**

In the first printf we supply a string literal. What you see is what you get. In the second printf call we pass in a **format string**, followed by one or more arguments ( the x and y variables). The format string is printed as is except for placeholders denoted with a % symbols. The values of the arguments (x and y) will be printed exactly where the placeholders are. Each placeholder (%) is followed by a **conversion instruction**---d in our examples---that says how the value is to be printed (in decimal notation, or binary, hexadecimal, etc.) In ou case, %d means decimal notation. At the end of the format there is an escape sequence ***\n*** which denotes a newline.

To read data from the keyboard our first C program uses:

**scanf("%d", &x );**

The first argument is, again, a format string. It is used to instruct the function scanf how to interpret what the user types. In this case it instructs scanf to interpret the user's input as an integer in decimal notation. The second argument contains the name of the variable where the input is to be stored (x). Note the ampersand character ( **&** ) before the x variable. This ampersand character is the **address-of operator**. This operator when placed before an variable produces the **address** of that variable, rather than the **value** of the variable. This address value is a positive memory address between 0 and however much memory you have in your computer. **scanf** needs the **address** of x not the value of x because scanf wants to know **where** to store the numeric conversion of the string you typed into the keyboard.

Recall that when we just want the value of **x** we just use the name x in a statement such as:

**x=15;** // assignment into the **value** of x  
 or  
 **y=x+5;** // lookup the **value** of x  
 or  
 **printf("value of x: %d", x );** // lookup the **value** of x

This is the first time we have ever been concerned with the address of where a variable is stored in memory. Java intentionally shields us from any such concerns. C does not. Thus it is important to understand the distinction between the **value** of a variable, and the **address** of a variable. They are not the same. If we want the address we must put the **&** operator immediately to the left of the variable name.

**Basic data types**

The four basic data types are int, float, double, and char.

[data-types.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/data-types.c)

In

 printf("%d %c\n",x,a);  
 printf("%3d %5c\n",x,a);  
 printf("%f %e\n",e,d);  
 printf("%.9f %.9e\n",e,d);  
 printf("%20.9f %20.9e\n",e,d);

conversion instructions other than d are used. Here are the conversions we will use:

|  |  |
| --- | --- |
| **Conversion Specification** | **Output** |
| %c | character |
| %s | string of characters |
| %d or %i | decimal integer |
| %e | floating point number in e-notation |
| %f | floating point number in decimal notation |
| %p | pointer |
|  |  |
| %u | unsigned decimal integer |
| %o | octal integer |
| %x | hexadecimal integer, using lower case |
| %X | hexadecimal integer, using upper case |
| %% | Prints a % sign |

A number right after the placeholder (%) specifies the width (i.e., the number of characters) that the value should occupy, with blank spaces used as fillers when necessary:

printf("%3d %5c\n",x,a);  
printf("%20.9f %20.9e\n",e,d);

For float and double values, a precision can be specified:

printf("%.9f %.9e\n",e,d);  
printf("%20.9f %20.9e\n",e,d);

The following is a list of some common escape sequences: 

|  |  |
| --- | --- |
| **Escape Sequences** | **Meaning** |
| \n | New break |
| \b | Backspace |
| \f | Form feed |
| \r | Carriage return |
| \t | Horizontal tab |
| \\ | Prints a \ |
| \' | prints a ' |
| \" | prints a " |

To learn more about printf, we use the man page

$ man -k printf  
asprintf (3)         - print to allocated string  
dprintf (3)          - print to a file descriptor  
fprintf (3)          - formatted output conversion  
fwprintf (3)         - formatted wide-character output conversion  
printf (1)           - format and print data  
**printf (3)           - formatted output conversion**  
snprintf (3)         - formatted output conversion  
sprintf (3)          - formatted output conversion  
swprintf (3)         - formatted wide-character output conversion  
vasprintf (3)        - print to allocated string  
vdprintf (3)         - print to a file descriptor  
vfprintf (3)         - formatted output conversion  
vfwprintf (3)        - formatted wide-character output conversion  
vprintf (3)          - formatted output conversion  
vsnprintf (3)        - formatted output conversion  
vsprintf (3)         - formatted output conversion  
vswprintf (3)        - formatted wide-character output conversion  
vwprintf (3)         - formatted wide-character output conversion  
wprintf (3)          - formatted wide-character output conversion

The printf C function is described in man 3:

$ man 3 printf  
PRINTF(3)                  Linux Programmer's Manual                 PRINTF(3)  
  
NAME  
       printf,   fprintf,  sprintf,  snprintf,  vprintf,  vfprintf,  vsprintf,  
       vsnprintf - formatted output conversion  
  
SYNOPSIS  
       #include <stdio.h>  
  
       int printf(const char \*format, ...);  
       int fprintf(FILE \*stream, const char \*format, ...);  
       int sprintf(char \*str, const char \*format, ...);  
       int snprintf(char \*str, size\_t size, const char \*format, ...);  
  
       #include <stdarg.h>  
  
       int vprintf(const char \*format, va\_list ap);  
       int vfprintf(FILE \*stream, const char \*format, va\_list ap);  
       int vsprintf(char \*str, const char \*format, va\_list ap);  
       int vsnprintf(char \*str, size\_t size, const char \*format, va\_list ap);  
  
   Feature Test Macro Requirements for glibc (see feature\_test\_macros(7)):  
  
       snprintf(), vsnprintf(): \_BSD\_SOURCE || \_XOPEN\_SOURCE >= 500 ||  
       \_ISOC99\_SOURCE; or cc -std=c99  
 Manual page printf(3) line 1  
...

In C, the char type is really an integer type. We use the conversion instructions to illustrate this:

[ascii.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/ascii.c)

**Algebraic operations**

C supports classical algebraic operators.

[arithmetic.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/arithmetic.c)

Notes:  
/ is the integer division operator when the numerator and denominator are integers.  
% is the modulus (remainder) operator.

**Loops**

There are three types of loops in C:

* for loop, with a starting *and* an ending condition
* while loop, with an ending condition *only* that is tested *before* the first iteration
* do-while loop, with an ending condition *only* that is tested *after* the first iteration

[loops.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/loops.c)

Some more examples

[sum.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/sum.c)  
[temp.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/temp.c)

Note the use the #define line to define **symbolic constants** LOWER, UPPER and STEP. It is good practice to use them instead of "burying" numbers like 300 and 20 deep inside the code. Symbolic constants can refer to values of any type (int, char, string, float, etc.)

**Conditionals**

The usual if-else statement is the conditional statement in C.

[blocks.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/blocks.c)

Note that the conditional expression must be in parentheses. Compound conditional expressions are created using || (or), && (and), and ! (not).

[terminal.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/terminal.c)

**Flow Control**

C supports two statements that modify the execution of a loop:

* continue: skip the rest of the loop iteration and continue with the next iteration (if the loop condition is still true though)
* break: skip the rest of the loop iteration and continue with statement that follows the loop body

[flow.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/flow.c)

**Functions**

Functions in C are defined as expected:

[power.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/power.c)

**Parameter passing**: In C, function arguments are passed **"by value"**. This means that the called function is given copies of the original values. For example, when the main program executes power(2, i) when the value of i is 5, copies of 2 and 5 are written to local variables base and n (they are local to the execution of function power):

int power(int base, int n)  
{  
  int i, p;  
  p = 1;  
  for (i = 1; i <= n; ++i)  
    p = p \* base;  
  return p;  
}

**C pointers**

We aleady saw that the & operator is used to obtain the (memory) address of a variable:

scanf("%d", &x );

Since a C programmer may need to store a variable's address for later use, C provides a data type called the **pointer type**. The pointer type allows us to declare variables **that are meant to hold the address of some other variable**. So, a pointer is a variable that contains the address of another variable.  
  
Declaring a pointer variable is done as follows:

int \*pi /\* creates a 32 bit variable whose data type is **pointer to int** \*/  
 int i; /\* an int var \*/  
 pi = &i; /\* pi now holds i's address - i.e it *points to* i \*/

[pointers.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/pointers.c)

Every variable in C has 2 at least 2 properties: 

* it's value (the contents of the variable)
* it's address (the memory address WHERE that value is stored)

A pointer variable is itself a variable and its value can change. A pointer can hold only one of 3 values:

* the NULL value ( value ZERO, indicates not pointing to any variable)
* the memory address of any variable of it's point type
* garbage

Pointer variables are NOT the data type of the variable they point to. The data type of pi (above) is *pointer to int* NOT INT. The name of the pointer variable is pi    NOT \*pi . **You can declare a pointer to ANY data type.**

char \*pc; /\* pc is type: pointer to char \*/  
float \*pf; /\* pf is type: pointer to float \*/  
double \*pd; /\* pd is type: pointer to double \*/  
  
typedef struct  
 double cost;  
 int age;  
} student\_type;  
  
student\_type \*ps; /\* ps can store the address of any student\_type variable \*/  
  
student\_type one\_student;  
ps = &one\_student; /\* ps has the address where one\_student is stored in memory \*/

**Dereferencing pointer variables**

Let's go back to our int example:

int \*pi; /\* variable of type pointer to int \*/  
int i; /\* plain old int var \*/  
pi = &i; /\* pi now contains address of var i \*/

We can use pointer variable pi to manipulate the contents of the int variable i. To do so we must ***dereference*** pi as follows:

\*pi = 15; /\* i now contains 15. \*/  
printf(" value pointed to by pi is: %d\n" \*pi ); /\* deref's pi and prints 15 \*/

The expression \*pi can be thought of as *the value at the address which is stored in pi.*  
  
Using pi with the \* in front of it is called **dereferencing** the pointer variable.  
  
In summary, when pi is defined as:

int \*pi;

* pi is a pointer (to an integer value)
* \*pi is an integer

**C pointers as function arguments**

In C arguments to functions are passed **by value**.

[pointers2.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/pointers2.c)

So, does the below function swap work correctly?

void swap(int x, int y)  
{  
  int temp;  
  
  temp = x;  
  x = y;  
  y = temp;  
}  
  
int main()  
{  
  int a = 0;  
  int b = -123;  
  swap(a, b);  
  return 0;  
}  
  
[pointers3.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/pointers3.c)

To alter the value of a variable (e.g., val in example pointers2 and a and b in example pointers3) in the calling function (i.e., function main), the calling function must pass a pointer to the variable.

[pointers4.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/pointers4.c)

The correct way to swap in C:

[pointers5.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/pointers5.c)

**C arrays and strings**

Arrays store a set of "indexable" values of the same type. 

[arrays.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/arrays.c)

For an arrays with n entries, the valid index values are from 0 to n-1.  
  
A C string is a special kind of array of characters: the string ends with character '\0', the NULL character which is not necessarily the last character of the array.

[str.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/str.c)

Notes:

* The null character '\0' contains 0 bits, so it can be replaced by integer 0
* Rather than defining variable s to be an array of 6 or 10 (or ...) characters, it is traditional to define it simply as char \*s.

**C pointer arithmetic**

Pointers can be incremented:

int y;  
int \*p = &y;  
p = p + 5;

Pointer p is incremented by 5\*4 = 20 bytes. This is because p is an integer pointer and what p + 5 really means is *the 5th integer after the one that p references*.  
  
Example: [pointers6.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/pointers6.c)

**C pointers and arrays**

Anything you can do with arrays, you can do with pointers (faster!) More precisely:

* the name of the array is really a pointer to the beginning of the array
* array indexing is really pointer arithmetic

After executing this code fragment

int a[10];  
int \*p;  
p = a;

element a[0] can be accessed as \*pa, or \*a, or p[0]  
element a[1] can be accessed as \*(pa+1), or \*(a+1), or p[1]  
element a[2] can be accessed as \*(pa+2), or \*(a+1), or p[2]  
etc.  
  
The same is true if a (respectively, p) was an array of (respectively, a pointer to) chars, doubles, etc   
  
In fact, C compiles a[i] to \*(a+i)  
  
Example: [pointers7.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/pointers7.c)

**Arrays as function arguments**

Example: [sort.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/sort.c)

**Structures**

A structure is a grouping of several variables, possibly of different type, under a single name.

[struct.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/struct.c)

The typedef instruction, which defines a new (user-defined) type, can be used to facilitate the usage of structs:

[struct2.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/struct2.c)

**Structures and pointers**

If e is defined as a pointer to an employee structure

employee \*eptr;

then the pointer must be dereferenced in order for the structure content to be accessed:

(\*eptr).first = "Sam";  
(\*eptr).last = "Smith";  
(\*eptr).age = 55;  
  
printf("%s %s, age %d\n", (\*eptr).first, (\*eptr).last, (\*eptr).age);

An alternative notation is typically used:

// alternative notation                                                         
eptr->first = "Sam";  
eptr->last = "Smith";  
eptr->age = 55;  
  
printf("%s %s, age %d\n", eptr->first, eptr->last, eptr->age);

[struct3.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/struct3.c)

Note: the call malloc(12) allocates 12 bytes and returns the addess of the first byte (similar to new() in C++ and Java)

**Function malloc()**

Function malloc() is C's version of the new operator you have seen in Java or C++ programs. Operator new is used to create a new object dynamically, i.e. at run time, i.e. while the program executes. What that means is the space in memory is allocated to store the object's instance variables.  
  
C is not object-oriented and does not have objects. It does however support dynamically allocated memory using the malloc() function. Function malloc() takes a input a number n and allocates n consecutive bytes in memory; it return a pointer to the first of the n bytes. The signature of malloc is

void\* malloc(int n)

Note that the input is an integer and the return type is a generic address: void\*. When you need to dynamically create a new integer, for example, you will use malloc() to allocate 4 bytes. However, in order to use these 4 bytes as an int value, you will need to cast the void\* pointer as an int\* pointer. Therefore, the typical usage of malloc() to "create" an integer dynamically is:

int \*iptr = (int \*) malloc(4);

Then you can assign an integer value to this new, dynamically allocated integer:

\*iptr = 5

To dynamically create an array of 10 integers you would need 40 bytes:

int \*array = (int \*) malloc(40)

Even better, for code portability reasons, would be using the C sizeof() function that takes a type and returns the number of bytes the type requires:

int \*array = (int \*) malloc(10\*sizeof(int))

Then you can use array just like a statically defined array:

for (int i = 0; i < 10; i++)  
  array[i] = i;

[malloc\_example0.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/malloc_example0.c)  
  
At the end of the above program the code

free(array);

gives back the dynamically allocated memory space pointed to by array to the operating system. This freeing of memory must be done when the dynamically allocated memory space is no longer needed by the program. Failure to do so may lead to a "memory leak" and, eventually, no more memory being available for dynamic memory allocation. (We will go much deeper into this subject in weeks 5-6.)

**Two-dimensional arrays**

A program that uses a statically defined 2D array:  
  
    [array2D.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/array2D.c)

**Two-dimensional dynamically allocated arrays**

There are two ways to create dynamically allocated 2D arrays. The first is to simply use a 1D array.  
  
    [malloc\_example1A.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/malloc_example1A.c)  
  
This approach requires a function that maps row index i and column index j to the appropriate index of the 1D array.  
  
Another approach is to create a multi-level array. To do this, it is helpful and cleaner to use the C typedef instruction to define new types for a table row and for the table itself:

typedef int\* row\_t;  
typedef row\_t\* table\_t;

The new type row\_t is just a new name for the int\* type; As we saw above, it can be used to refer to a dynamically allocated array of integers, i.e. a row of a 2-D array of integers.  
  
The new type table\_t is defined as row\_t\* and can thus be used to refer to a dynamically allocated array of row\_t entries. In other words, a table\_t variable will refer to an array of integer arrays, i.e. a 2-D array of integers.  
  
Here is the complete program that creates a 2x3 array of integers dynamically and initializes all its values to 0:  
  
    [malloc\_example1.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/malloc_example1.c)  
  
Another example illustrating how to dynamically create a 2-D array of structs:  
  
    [malloc\_example2.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/malloc_example2.c)  
  
In both cases, note how the dynamically allocated memory is freed when it is no longer needed.

**File Input/Output and parsing strings**

To open a text file test.txt for reading do:

FILE\* fp = fopen("test.txt", "r");

Then, to read the first line of the file into a character array buf, use function fgets():

fgets(buf, 1000, fp)

Functions fgets() will read a line or the first 999 bytes if the line is 1000+ bytes long) from the file refered to by fp and store it into the character array buf. The following is a complete program that opens file test.txt, reads it line by line, prints each line to standard output, and then closes the file:  
  
    [fileIO\_example1.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/malloc_example1.c)  
  
When executed on file [test.txt](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/test.txt), it will show on the screen the content of file test.txt:

$ ./fileIO\_example1   
a 100,:,11  
 b 200,:,22  
c 300,:,33  
 d 400,:,44  
e 500,:,55

Suppose now that you want to parse the lines in the above test.txt file that start with a blank space in order to extract the first character and the two integers. For example, you would like to extract b, 200 and 22 from line

 b 200,:,22

and store them, respectively, in variables, c, a, and b defined as:

  char c;  
  unsigned int a;  
  int b;

The function sscanf() can be used to do this. It takes as input: the character array buf that contains the line, a format string, and addresses of variables c, a, and b:

sscanf(buf, " %c %u,:,%d", &c, &a, &b);

The format string describes the format of the string stored in buf and the placeholders %c, %u, and %u indicate the places where the values to be extracted into c, a, and b are.  
  
    [fileIO\_example2.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/fileIO_example2.c)

**Command line arguments**

Programs sometimes take command line input arguments:

$ ./cmdline0 Ljubomir  
Hello, Ljubomir!  
[lperko@perko lecture1]$ ./cmdline0 Ellie  
Hello, Ellie!  
[lperko@perko lecture1]$ ./cmdline0  
Hello, World!

[cmdline0.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/cmdline0.c)  
  
If you want your C program to handle *any* command line input, you need to define function main() like this:

int main(int argc, char\* argv[])

The argument argc will be assigned the number of (blank separated) command line arguments when the program is executed; this includes the name of the executable (./cmdline0 in the above examples).  The argument argv is an array of strings that will contain all the command line arguments. For example, in 

$ ./cmdline0 Ljubomir  
Hello, Ljubomir!

the argc is will be 2, argv[0] will be string "./cmdline0" and argv[1] will be "Ljubomir".  
  
The following example uses a program (cmdline1.c) thar prints the values of argc and of every every string in array argv:

$ ./cmdline1 how many arguments?  
4 arguments  
 0: ./cmdline1  
 1: how  
 2: many  
 3: arguments?

[cmdline1.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/cmdline1.c)

**Function getopt()**

Function getopt() is used to process Unix-like command line options of the type -x such as the -l option of the ls (list directory) instruction. Here is some sample usage of a program getopt\_example.c that can take 4 command line options a, b, c, and d. Option a takes an integer argument, option b takes a string argument and options c and d take no argument (they are simply flags):

$ ./getopt\_example                         
a = 0, b = (null), flagc = 0, flagd = 0     // all options receive default values  
$ ./getopt\_example -a 3  
a = 3, b = (null), flagc = 0, flagd = 0     // option a is 3  
$ ./getopt\_example -a 3 -b hello!  
a = 3, b = hello!, flagc = 0, flagd = 0     // a is 3 and b is string "hello!"  
$ ./getopt\_example -a 3 -b hello! -c  
a = 3, b = hello!, flagc = 1, flagd = 0     // flag c is set as well  
$ ./getopt\_example -a 3 -b hello! -d  
a = 3, b = hello!, flagc = 0, flagd = 1     // example with flag d instead  
$ ./getopt\_example -b hello! -d  
a = 0, b = hello!, flagc = 0, flagd = 1     // using b and d options only  
$ ./getopt\_example -e  
./getopt\_example: invalid option -- 'e'     // an invalid option  
ERROR!

[getopt\_example.c](http://reed.cs.depaul.edu/lperkovic/csc406/lecture1/getopt_example.c)  
  
The last argument of function getopt in the above program is "a:b:cd": that is the option string that contains the legitimate option characters. Any character followed by a colon (:) is an option that takes an argument.  
  
By repeatedly calling getopt() in a while loop as shown above, we can retrieve the command line options one by one. Any option that takes an argument can access this argument through the global variable optarg defined in the getopt.c library.

**Extracting the set index and tag from an address**

In order to extract the set index and the tag from a memory address stored in (unsigned long long int) variable address, you need to define a "mask" and then use C logical operators & (logical and) and >> (logical right shift).  
  
Assume that variables s and b contain the number of set index and byte offset bits in a memory address. First you create the mask which is an integer that contains, in its low order bits, s ones:

unsigned long long int mask = (pow(2, s) - 1);

Then the set index is obtained as follows:

unsigned long long int set\_index = (address >> b) & mask;

and the tag is obtained as follows:

unsigned long long int tag = address >> (s+b);