|  |  |
| --- | --- |
|  | Betty linter To run the Betty linter just with command betty <filename>:   * Go to the [Betty](https://intranet.alxswe.com/rltoken/wQ4sMfsWfxvyfN67Sc11zA) repository * Clone the [repo](https://intranet.alxswe.com/rltoken/wQ4sMfsWfxvyfN67Sc11zA) to your local machine [https://github.com/holbertonschool/Betty.git ] * cd into the Betty directory * Install the linter with sudo ./install.sh * emacs or vi a new file called betty, and copy the script below:   Function Declarations & Definitions   * Parameters on the same line if they fit. * Return type on the same line as function name. * Wrap parameter lists which do not fit on a single line. * The indentation is 4 spaces. * Choose good parameter names. * All parameters should be aligned if possible. * The open curly brace is always on the next line by itself. * The close curly brace is always on the last line by itself. * Wrapped parameters should indent to the function’s first arguments. * The open parenthesis is always on the same line as the function name. * There is never a space between the function name and the open parenthesis. * There is never a space between the open parentheses and the first parameters. * All parameters should be named, with identical name in declaration and implementation.   **Vertical Whitespace**   * Minimize use of vertical whitespace. * Do not end functions with blank lines. * Do not start functions with blank lines. * Do not use blank lines when you do not have to. * Do not put more than one or two blank lines between functions. * Blank lines inside a chain of if-else blocks may well help readability. * Blank lines at the beginning or end of a function very rarely help readability. |
|  | Not for bettry linter   1. Add empty new line after declaring a variable   Int a;  For (a=0; a<10; a++)  {  Printf (“hello”);  }   1. For line continuation |
|  |  |
|  | In the Vi/Vim text editor, there are several delete commands for removing text:   1. x - Deletes the character under the cursor. 2. **dd** - Deletes the entire line the cursor is on. 3. dw - Deletes from the cursor to the end of the word. 4. db - Deletes from the cursor to the beginning of the word. 5. **d$ = D** - Deletes from the cursor to the end of the line. 6. d0 - Deletes from the cursor to the beginning of the line. 7. dG - Deletes from the cursor to the end of the file. 8. :delete - Deletes the specified line range, for example :1,3delete deletes lines 1 to 3. 9. $ vi <filename> — Open or edit a file. 10. i — Switch to Insert mode. 11. **Esc** — Switch to Command mode. 12. :w — Save and continue editing. 13. :wq or ZZ — Save and quit/exit vi. 14. :q! — Quit vi and do not save changes. 15. yy — Yank (copy) a line of text. 16. p — Paste a line of yanked text below the current line. 17. o — Open a new line under the current line. 18. O — Open a new line above the current line. 19. A — Append to the end of the line. 20. a — Append after the cursor’s current position. 21. I — Insert text at the beginning of the current line. 22. b — Go to the beginning of the word. 23. e — Go to the end of the word. 24. x — Delete a single character. 25. dd — Delete an entire line. 26. Xdd — Delete X number of lines. 27. Xyy — Yank X number of lines. 28. G — Go to the last line in a file. 29. XG — Go to line X in a file. 30. gg — Go to the first line in a file. 31. :num — Display the current line’s line number. 32. h — Move left one character. 33. j — Move down one line. 34. k — Move up one line. 35. l — Move right one character.   Note that these commands can be combined with a number to delete multiple lines, words, etc. For example, 3dd deletes 3 lines.  Cutting and pasting ghp\_qtanNwEmqthexzj8jgsEDt2urx6dFI44v5Q5  Cut with dd paste using p or P  Copy with yy paste using p or P  To copy code block copy with nyy -> paste with p or P | cut with ndd -> paste with p or P  Search **?**  Replace **?**  **Save- (:w)**  **Quit- (:q) (zz)**  **Rename- (: w filename)**  **Save without the changes - (:q!)**  **Force write (­­­­­write to none writable file) - (:w!)** |
|  |
|  | **0x01. Shell, permissions** |
|  | File Owner, group owner, all users= -rwxrwxrwx = 777   * [**chmod**](http://linuxcommand.org/lc3_man_pages/chmod1.html) - modify file access rights * [**su**](http://linuxcommand.org/lc3_man_pages/su1.html) - temporarily become the superuser * [**sudo**](http://linuxcommand.org/lc3_man_pages/sudo8.html) - temporarily become the superuser do * [**chown**](http://linuxcommand.org/lc3_man_pages/chown1.html) - change file ownership * [**chgrp**](http://linuxcommand.org/lc3_man_pages/chgrp1.html) - change a file's group ownership   r - Allows the contents of the directory to be listed if the x attribute is also set.  w - Allows files within the directory to be created, deleted, or renamed if the x attribute is also set.  **x - Allows a directory to be entered (i.e. cd dir)**  rwx rwx rwx = 111 111 111  rw- rw- rw- = 110 110 110  rwx --- --- = 111 000 000  rwx = 111 in binary = 7  rw- = 110 in binary = 6  r-x = 101 in binary = 5  r-- = 100 in binary = 4   * change the owner of some\_file from "me" to "you". We could:   [me@linuxbox me**]$ sudo chown you some\_file**   * To change/set user and group for a file at the same time   **or $ sudo chown user1:group1 filenamae**   * In fact, modern distributions don't even set the root account password thus making it impossible to log in as the root user. A root shell is still possible with sudo by using the "-i" option:   [me@linuxbox me]$ sudo –i   * The --reference option specifies the file whose permissions should be used as the basis for the new permissions, in this case "olleh". The file whose permissions are being changed is specified at the end, in this case "hello".   chmod --reference=olleh hello  Explanation:  Ads execute permission to all subdirectories of the current directory tom for the owner, the group owner and all other users. Regular files should not be changed.  **find**  **find tom\_1 -type d -exec chmod 755 {} +**   * find: the command that searches for files and directories. * .: the starting directory for the search (the current directory). * -type d: searches only for directories (not files). * -exec: executes the command following it for each directory found. * chmod 755: adds execute permission for the owner, the group owner and all other users. * {}: placeholder for the current directory being processed by find. * +: allows multiple directories to be passed to chmod at once, improving performance.   The number 755 is a combination of the binary values rwxr-xr-x, which means:   * 7: rwx - the owner has read, write, and execute permissions. * 5: r-x - the group owner and others have read and execute permissions, but not write.4   [**find**](http://linuxcommand.org/lc3_man_pages/find1.html)**. -type f -print | [wc](http://linuxcommand.org/lc3_man_pages/wc1.html) –l**  **( to find directory use –type d, -l: to count line, -c: to count character, -w to count words**   * find is used to search for files in a directory hierarchy. The . argument specifies the starting directory, i.e., the current directory. The -type f option is used to search only for files (as opposed to directories, symbolic links, etc.). The -print option tells find to print the names of the files it finds to standard output. * **wc (word count)** is used to count the number of lines, words, and characters in a file or in the input it receives. In this case, it receives its input from the output of the find command via the pipe (|) operator. The -l option tells wc to count only the number of lines in its input.   **Find –type f –name test.txt**  **Find . – name \*.txt**  **Find –empty** > find empty files  **Find – delete** > find empty files and delete them  **Find . –type d –mindepth 1 | wc –l >** count the number of directories and subdirectories except the current directory  The -mindepth 1 option tells find to start the search one level below the current directory.  **Wc**  **Wc names> displays words, line, bytes**  **Wc –l names> displays words**  **Wc –c names> count the number of bytes in the input.**  **Wc –w names > display lines**  **Wc –wl names > display line and words**  **-l: Counts the number of lines in the input.**  **-c: Counts the number of bytes in the input.**  **-w: Counts the number of words in the input.**  **Sort names > sort alphabetically**  **Sort namer –r> sort in reverse order**  **Sort names | uniq > without duplicate (NB: unique can only be used if the file is sorted)**  **Sort names | uniq –c > without duplicate but with duplicate count**  **Sort names | uniq –u > only uniqe words (that occur only once)**  **Grep**  **.Grep ( global regular expression pattern)**  **.Grep “no” names> lines containing “no”**  **.Grep -v “no” names> lines not containing “no”**  **.Grep –I “NO” names > case insensitive**  **.Grep –n “no” names >finds patterns and display the line on which they are found.**  **.Grep –c “no” names > count the number of occurrences for “no”**  **.Grep –l “no” file1 file2 -- > files containing the word “no”**  **.Grep –w “no” : line containing the full word no ( nono not included)**  .Cartor symol ^ (used to represent the beginning of a line)  **.Grep ^z names> lines that start with letter z**  **.Grep s$ names > lines that end with letter s**  **.grep –r “names” >** -r (or --recursive): This option tells grep to search for the pattern recursively in all files under the specified directory.  **------------------------------------------**  **.grep -r "pattern" directory/ will search for "pattern" in all files within the "directory" and its subdirectories.**  **This is equivalent to**  **File . –type f –exec grep –n “pattern” {} \;**  **-------------------------------------**  **Tr (text transformation)**   * **Case conversion** * **Deleting characters** * **Basic text replacement** * **Can’t read files directly** * **Used together with Pipes(|) and also append(> or >>) signs**   **Tr e o >d**  **Tr –d \n > delete ‘new line character’** d flag deletes the newline character (\n). This is useful for combining multiple lines of text into a single **line, or for removing any trailing newlines from a line of text.**  **Echo “Africa” | tr [:lower:] [:upper:]**  **Rev prints content in reverse order**  **/etc/passwd ( is like user account )**  **is a text file that contains information about all users on the system**  **Cut:**  command selects and prints parts of lines from files  To use cut command specified field delimiters must be present in the input file.  **Cut –c1** - cuts and prints the first character  **Cut -d ‘:’ –f1,2 -** (use : as delimiter and print 1st, 2nd fields)  **cut -d $'\t' -f1 > tab as delimiter** |
|  | Most commonly used shell commands  awk # pattern scanning and processing language  basename # strip directory and suffix from filenames  bg # resumes suspended jobs without bringing them to the foreground  cat # print files  cd # change the shell working directory.  chmod # change file mode  chown # change file owner and group  crontab # maintain crontab files  curl # transfer a URL  cut # remove sections from each line of files  date # display or set date and time  dig # DNS lookup utility  df # report file system disk space usage  diff # compare files line by line  du # estimate file space usage  echo # display a line of text  find # search for files in a directory hierarchy  fg # resumes suspended jobs and bring them to the foreground  grep # print lines matching a pattern  kill # send a signal to a process  less # read file with pagination  ln # create links  ls # list directory contents  lsb\_release # print distribution-specific information  lsof # list open files  mkdir # create  mv # move files  nc # arbitrary TCP and UDP connections and listens  netstat # print network connections, routing tables, interface statistics...  nice # execute a utility with an altered scheduling priority  nproc # print the number of processing units available  passwd # change user password  pgrep # look up processes based on name and other attributes  pkill # send signal to processes based on name and other attributes  printenv # print all or part of environment  pwd # print name of current/working directory  top # display Linux processes  tr # translate or delete characters  ps # report a snapshot of the current processes  rm # remove files or directories  rmdir # remove directories  rsync # remote file copy  scp # secure copy (remote file copy program)  sed # stream editor for filtering and transforming text  sleep # suspend execution for an interval of time  sort # sort lines of text file  ssh # OpenSSH SSH client (remote login program)  ssh-keygen # SSH key generation, management and conversion  su # substitute user identity  sudo # execute a command as another user  tail # output the last part of files  tar # manipulate archives files  tr # translate or delete characters  uname # Print operating system name  uniq # report or omit repeated lines  uptime # show how long system has been running  w # Show who is logged on and what they are doing  whereis # locate the binary, source, and manual page files for a command  which # locate a command  wc # print newline, word, and byte counts for each file  xargs # build and execute command lines from standard input  | # redirect standard output to another command  > # redirect standard output  < # redirect standard input  & # send process to background |
|  | **cut -d, -f1,3 file.csv**  cut is a command-line utility that is used to extract specific fields (columns) from a file or input. The fields are delimited by a specified character, such as a tab, comma, or space.   * **-d** option is used to specify the field delimiter * **-f** option is used to specify the field(s) to be extracted.   In this example, the field delimiter is a comma (,) and the -f1,3 option tells cut to extract the first and third fields from each line of the file file.csv.  **head -n 14 log.txt**  You can use the head command to display the first n lines of a file. The default value of n is 10, but you can specify a different value with the -n option.  grep -A 3 root /etc/passwd  Explanation:   * grep is a command-line tool that searches for a specified pattern in a given file or input. * -A 3 is an option for grep that tells it to display 3 lines after each match of the pattern. * root is the pattern to search for in the /etc/passwd file. * /etc/passwd is the file to search for the pattern in. |
|  | **\\\\*\\\\\'\"Best School\"\\\'\\\\\\*\$\\\?\\\\*\\\\*\\\\*\\\\*\\\\*\:\)** |
|  | Suppose you have a text file named example.txt in your current working directory. To view the contents of the file using less, simply type the following command in the terminal:  less example.txt  alias [-p] [name="value"]  When used with no arguments and with or without the *-p* option, alias provides a list of aliases that are in effect for the current user, i.e.,  alias  alias ls="ls -al"  Such an alias can be disabled temporarily and the core command called by preceding it directly (i.e., with no spaces in between) with a backslash, i.e.,  \ls  The following is an example of creating two separate aliases simultaneously, as contrasted with creating a single alias that launches two separate commands:  alias p="pwd"; l="ls -al"   The alias name and the replacement text can contain any valid shell input except for the *equals* sign ( = ). |
|  | Convert binary to decimal  **echo "$((2#101010))"**  **count number of directories in $path environmental variable**  **echo $PATH | tr ':' '\n' | wc –l**  **convert from one base to another**  **echo $((2#101010)) base 2 to 10**  **echo $(( 16#FF)) base 16 to 10**  **printf '%x\n' 255 hexadecimal value of 255** |
|  | **printf**  printf is a powerful command in Linux that can be used to format and print output to the terminal. In addition to %x, there are several other format specifiers that can be used with printf to format output in different ways. Here are some commonly used format specifiers:   * %d: This specifier is used to print an integer in decimal format. For example, printf "%d" 42 would print the number 42. * %f: This specifier is used to print a floating-point number. For example, printf "%.2f" 3.14159 would print the number 3.14. * %s: This specifier is used to print a string. For example, printf "%s" "Hello, world!" would print the string "Hello, world!". * %c: This specifier is used to print a single character. For example, printf "%c" "A" would print the letter A. * %e or %E: These specifiers are used to print a number in scientific notation. For example, printf "%.2e" 100000 would print the number 1.00e+05. * %u: This specifier is used to print an unsigned integer in decimal format. For example, printf "%u" 42 would print the number 42. * %o: This specifier is used to print an integer in octal format. For example, printf "%o" 42 would print the octal number 52. * %x or %X: These specifiers are used to print an integer in hexadecimal format. %x prints the number in lowercase, while %X prints it in uppercase. For example, printf "%x" 42 would print the hexadecimal number 2a. |
|  | Here are some common shell initialization files, along with their purpose and typical location:   1. ~/.bash\_profile - This file is a user-specific shell initialization file for the Bash shell. It is executed when a user logs in to the system and starts a Bash shell. It is used to set up the user's environment variables and aliases, and to execute any commands that should be run every time the user logs in. If ~/.bash\_profile does not exist, Bash will look for ~/.bash\_login or ~/.profile as a fallback. 2. ~/.bashrc - This file is a user-specific shell initialization file for the Bash shell. It is executed every time a new Bash shell is started, not just when the user logs in. It is typically used to set up aliases, functions, and other shell customizations that should be available to the user in all Bash sessions. 3. ~/.zshrc - This file is a user-specific shell initialization file for the Zsh shell. It is executed every time a new Zsh shell is started, and it is typically used to set up environment variables, aliases, and other customizations. 4. ~/.cshrc - This file is a user-specific shell initialization file for the C shell (csh) and its derivatives. It is executed every time a new C shell is started, and it is typically used to set up environment variables, aliases, and other customizations. 5. /etc/bashrc - This file is a system-wide shell initialization file for the Bash shell. It is executed every time a new Bash shell is started, and it is typically used to set up system-wide environment variables, aliases, and other customizations that should be available to all users on the system. 6. /etc/csh.cshrc - This file is a system-wide shell initialization file for the C shell (csh) and its derivatives. It is executed every time a new C shell is started, and it is typically used to set up system-wide environment variables, aliases, and other customizations that should be available to all users on the system. |
|  |  |

|  |  |  |
| --- | --- | --- |
| the GNU Awk programming language is explained in [Chapter 6](https://tldp.org/LDP/Bash-Beginners-Guide/html/chap_06.html). | | https://tldp.org/LDP/Bash-Beginners-Guide/html/chap\_06.html |
|  | |  |
|  | |  |
| C PROGRAMING | | |
| Header libraries:  #include <stdlib.h>: malloc()  #include <string.h>: strlen()  #include <stdio.h>: printf()  #include <stddef.h>:  #include <fcntl.h>: (file descriptor operations) , open()  #include <sys/type.h> to use datatypes found in unix based operating systems. Like pid\_t, uid\_t, gic\_t, off\_t, size\_t.  There are three standard streams in C: stdin (standard input), stdout (standard output), and stderr (standard error).  stdin is used to read input from the user or from a file.  stdout is used to display output to the user or to a file.  stderr is used to display error messages or diagnostics that are not part of the normal output of a program. (Typically associated with the console or terminal where the program is run.) This means that error messages will be printed directly to the terminal or console, separate from the normal output of the program that is printed to stdout.  **fprintf** is used to print messages to stderr. ( This can be useful for debugging purposes, or for providing more detailed error messages to the user.) | | |
| Puts | puts is a function that writes a string of characters to the console, followed by a newline character. It takes one argument, a pointer to a null-terminated string, and returns a non- | |
| Putchar | putchar is a function that writes a single character to the console. It takes one argument, an integer representing the character to write, and returns the character written or EOF if an error occurred. | |
| printf | printf is a function that writes formatted output to the console or to a file. It takes a format string and a variable number of arguments, and returns the number of characters written. | |
|  |  | |
| Question: Write a script that runs a C file through the preprocessor and save the result into another file.   * The C file name will be saved in the variable $CFILE * The output should be saved in the file c   Answer: gcc -E $CFILE -o c  This command uses the -E flag to tell gcc to run only the preprocessor phase and  Question: Write a script that compiles a C file but does not link.   * The C file name will be saved in the variable $CFILE * The output file should be named the same as the C file, but with the extension .o instead of .c.   + Example: if the C file is main.c, the output file should be main.o   Answer: to compile c source file into object file   * gcc: The GCC compiler is invoked to compile the source code in the specified C file. * -c: This option tells GCC to compile the source code into an object file rather than an executable. * $CFILE: This is a variable that holds the name of the C source file to be compiled. * -o: This option specifies the output file name that will be produced by the GCC compiler. * ${CFILE%.\*}.o: This expression uses parameter expansion to generate the output file name. ${CFILE%.\*} expands to the name of the input file without its extension. .o is added to the end to specify the output file type as an object file.   Question: Write a script that generates the assembly code o  f a C code and save it in an output file.   * The C file name will be saved in the variable $CFILE * The output file should be named the same as the C file, but with the extension .s instead of .c.   + Example: if the C file is main.c, the output file should be main.s   Answer: This command uses the following options:   * -S: tells GCC to stop after the assembly code generation step, without creating an object file or executable. * $CFILE: expands the variable $CFILE to the name of the C file that you want to compile. * -o: specifies the output file. * ${CFILE%.c}.s: expands to the name of the output file, which is the same as the input file with the extension changed to ".s".   For example, if the C file is named "main.c", the command will produce an output file named "main.s" with the assembly code for the C file.  Question: Write a script that generates the assembly code (Intel syntax) of a C code and save it in an output file.   * The C file name will be saved in the variable $CFILE. * The output file should be named the same as the C file, but with the extension .s instead of .c.   + Example: if the C file is main.c, the output file should be main.s   Ans: gcc -S -masm=intel $CFILE -o ${CFILE%.c}.s  This command uses the following options:   * -S: tells GCC to stop after the assembly code generation step, without creating an object file or executable. * -masm=intel: specifies the use of Intel syntax instead of the default AT&T syntax for the assembly code. * $CFILE: expands the variable $CFILE to the name of the C file that you want to compile. * -o: specifies the output file. * ${CFILE%.c}.s: expands to the name of the output file, which is the same as the input file with the extension changed to ".s".   NB. Change file extension | | |
| **Pointers**  char \*s2 = NULL; initializes the pointer s2 to a null pointer, which means that it does not point to any memory address. A null pointer is often used to indicate that a pointer does not currently point to a valid memory location. In contrast to s1, s2 does not point to any valid memory location and any attempt to access its contents will result in undefined behavior. | | |
| Pointers | | |
| char \*s1 = "";   * initializes the pointer s1 to point to a string literal that contains only the **null character '\0';** * This creates a character array of length 1 with the null character as its only element * s1 **points to a string of length 0**, which is different from a null pointer * The string pointed to by s1 **is read-only**, and any **attempt to modify its contents will result in undefined behavior.** | | char \*s2 = NULL; char \*s2 = 0  The value of NULL is typically defined as 0,   * initializes the pointer s2 to a **null pointer**. * which means that it does not point to any memory address. * A null pointer is often used to indicate that a **pointer does not currently point to a valid memory location.** * The location pointed by s2 is **invalid memory location.** * Any **attempt to access its contents will result in undefined behavior**. |
| Function pointers  Declaring the function  Void neyo(int a);  Void(\*f)(int);  **\*f = &neyo;** [to call it \*f()]  Or **f = neyo;** [to call it f()] | | **0x1A. C - Hash tables** *1 . suppose you have this 2 arguments*  *const char \*key*  *const char \*value:*  *a*  *However you have this struct*  *typedef* struct hash\_node\_s  {       char *\**key;       char *\**value;       struct hash\_node\_s \*next;  } hash\_node\_t;  Assigning list->key = key will raise error since we are not allowed to assign const char to char so to come across this we use strdcpy  **Allocating memory for list->key and list->value using strdup is necessary because the key and value parameters passed to the hash\_table\_set function are declared as const char\*. This means that they point to constant strings, and you cannot directly assign them to non-const pointers.** |

lib\_putchar.o

|  |  |
| --- | --- |
| * **'ar', for 'archiver'.**   This program can be used to create static libraries (which are actually archive files), modify object files in the static library, list the names of object files in the library, and so on  **C** flag tells ar to create the library if it doesn't already exist  **r** flag tell ar to replace older object files in the library, with the new object files  example: ar rc libutil.a util\_file.o   * **randlib**   The command used to create or update the index is called 'ranlib', and is invoked as follows:  Example: ranlib libutil.a  **Advantage of using Static Library**   1. **Portability:** you can write code that will work the same way on different systems without having to modify it for each platform. 2. **Time-saving**: Standard libraries provide pre-written functions. This can save you time and effort compared to writing the functions from scratch. 3. **Efficiency:** Standard libraries are typically well-written and optimized for performance, which means that they can be faster and more efficient than code you might write yourself. This is especially true for complex algorithms or data structures. 4. Consistency: By using standard libraries, you can ensure that your code follows established conventions and best practices. This can make your code easier to understand and maintain for other programmers who may work on your code in the future. 5. Reliability: Standard libraries are thoroughly tested and maintained, which makes them more reliable than code that you might write yourself. This can help to ensure that your code is robust and free from errors. | **Preprocessor**: (-E converts to preprocessor)   * Removes comments * Include header file code in the file itself * Replace macro with their value * Generates (.c extension)   **Compiler**: ( -S converts to assembly code)   * Generates assembly code (mnemonics)   **Assembler**: ( -C converts assembly to object )   * Converts to assembly code into object code or into zero and one. * **Generates (.o extension)**   **Linker**:   * Links object library with objects and create exe file. * Generates (a.out (on unix), a.exe (in widows sys)   3-11-2023  **Creating and Use static libraries in C programming**   * + 1. Create source code ( filename.c file)     2. Create object code for fname.c   **Gcc - c fname.c**  Output: fname.o   * + 1. Create static library using fname.o   $**ar rcs libfname.a fname.o**  >ar: archive  >r: replace old object files with new one  >c: create object file if it doesn’t exist  >s: create index of the symbols in the archive   * + 1. Link it to your program and use it   $gcc –o fname fname.c –L –lfname  (note: it is customary to use lib infront of our archive name, and we can leave it too.   * With lib:   Archive name: libfname.a  Link: gcc –o fname fname.c –L –lfname   * Without lib:   Archive name: fname.a  Link: gcc –o fname fname.c –L –lfname  >-o: output to  >**-L.**: specifies the directory for the library file look for library files in current folder  > -l : Specifies the name of the library file  **Note:**  **gcc filename.c -o filename -L. –lmylib and**  **gcc -o filename filename.c -L. –lmylib are both the same**  When copying the archive file to another  location, use 'cp -p', instead of only 'cp'. The '-p' flag tells 'cp' to keep all attributes of the file, including its access permissions, owner (if "cp" is invoked by a superuser) and its last modification date. This will cause the compiler to think the index inside the file is still updated. This method is useful for makefiles that need to copy the library to another directory for some reason. |
| Array initialization   * 1. Brace enclosed initialization   Char digit [ ] = {‘1’, ‘2’, ‘3’, ‘4’}  2 – string literal initialization  Char word [ ] = “thomas kitaba”  Nb: name of array: - is an alias for the address of the arrays first element ( base address)  IN C:  You can not reassign an array name  in python and java array memory mgmt occurs on heap while in c memory is allocated on stack  char \*\*temp  simply it is a pointer to a string    when you think of strings in C think of it as you've got a pointer to a  pointer to a character array and that's what an array of strings looks like | An array's name is an identifier that  evaluates to the address of its first element  \* Nota variable. Cannot be reassigned!  + You can think of it as a special, restricted case  of a pointer. ( we can not change where the array name points to we can’t increment it like pointer)  **Pointer vs Array**  ====  NB: size of is an operator not a function  Operator: sizeof()  Operand: typename  Return type: size\_t  Display return type: %lu (long unsigned)  Or : %zu (z specifies u is of type size\_t)  ( that is why we don’t have to specify any library to use it)   * To get length of array without using strlen or without loop (when it is initialized)   Array type array[ ] ={ . . . . . .};  Size\_t length = sizeof(array) / sizeof (array type)  Or Size\_t length = sizeof(array) / sizeof (array[0]) |

|  |
| --- |
| * **calloc()**allocate a block of memory dynamically in the heap with a specified size and initialize all the bytes to zero. ( the memory locations are contigious) * **memset()** function is used to set a block of memory with a specified value.   takes three arguments: a pointer to the memory block to be set, the value to be set, and the number of bytes to be set.   * **strncpy()** is used to copy a specified number of characters from one string to another. It takes three arguments: the destination string, the source string, and the maximum number of characters to be copied. It ensures that the destination string is not overwritten by the source string and that the copied string is null-terminated. If the length of the source string is less than the specified length, the remaining characters are filled with null bytes. * **memcpy()** is to copy a block of memory from one location to another. When you call memcpy(), it copies the contents of the source memory location to the destination memory location, overwriting the previous contents of the destination memory location. * **strdup**() function creates a new copy of a string in memory, and returns a pointer to the newly allocated memory that contains a copy of the original string. The memory is allocated using malloc() function. The strdup() function is also declared in the string.h header file. |
| * **Struct box {**   Int width;  Int height;  };   * **Decleration**   // struct <struct name> <variable name>     * **Initialization**  1. Struct box box1 = { 0}; c99 standard 2. Struct box box1= { 1, 3}; 3. Box1.width = 10, Box1.height = 20; 4. box box1 = {.width = 2, .height = 4 };  * **Aliasing with typedef**   Syntax: Typedef <existing type> <new-name>;  Example1- Typedef unsigned int whole\_number;  Example2- Typedef struct box box;  Example3- typedef **Struct box {**  Int width;  Int height;  } **box**;   * take address of   box \*ptr\_to\_box1 = &box1;   * copy over all member of a struct   box copy\_of\_box1 = box1;  \*ptr\_to\_box1 = anotherpoint;  NB-  Static variables are placed on the heap  Example, array, functions, structs  Daynamic variables are placed on the on the heap  Example, malloced pointer variable |
| In addition to formatting signed decimal integers like %d, the %i format specifier in C can also be used to format integers in other number systems such as octal and hexadecimal.  When using %i, the printf() function automatically detects the base of the number based on the prefix of the argument. If the argument starts with "0x" or "0X", it is interpreted as a hexadecimal number. If the argument starts with "0", it is interpreted as an octal number. If the argument does not have any prefix, it is interpreted as a decimal number. For example:  c  int num1 = 42; // decimal number  int num2 = 052; // octal number (52 in decimal)  int num3 = 0x2A; // hexadecimal number (42 in decimal)  printf("%i %i %i\n", num1, num2, num3); // Output: 42 42 42  Note that when using %i to format hexadecimal and octal numbers, the output is printed in decimal form. If you want to print the numbers in their original base, you can use %x for hexadecimal and %o for octal.  int num4 = 0x2A; // hexadecimal number (42 in decimal)  printf("%x\n", num4); // Output: 2a (hexadecimal)  printf("%o\n", num4); // Output: 52 (octal)  In general, it's best to use %d for decimal integers, %u for unsigned decimal integers, %x for hexadecimal integers, and %o for octal integers. However, %i can be useful when you need to format numbers in different bases using the same format specifier.  how to change octal to decimal  To convert an octal number to decimal in C programming, you can use the atoi() or strtol() function to convert the octal number represented as a string to an integer, and then perform any necessary arithmetic operations to convert it to decimal.  Here's an example using atoi() function:  c  #include <stdio.h>  #include <stdlib.h>  int main() {  char octal\_str[] = "52"; // octal number as a string  int decimal\_num;  decimal\_num = atoi(octal\_str); // convert the octal string to an integer  printf("The decimal equivalent of octal %s is %d\n", octal\_str, decimal\_num);  return 0;  }  In this example, we first define an octal number as a string octal\_str which represents the octal number 52. We then use the atoi() function to convert the string to an integer, which gives us the decimal equivalent of the octal number. Finally, we print the decimal equivalent using printf() function.  Alternatively, you can use the strtol() function, which allows you to specify the base of the number system that you're converting from (in this case, base 8 for octal). Here's an example using strtol() function:  c  #include <stdio.h>  #include <stdlib.h>  int main() {  char octal\_str[] = "52"; // octal number as a string  int decimal\_num;  decimal\_num = strtol(octal\_str, NULL, 8); // convert the octal string to an integer  printf("The decimal equivalent of octal %s is %d\n", octal\_str, decimal\_num);  return 0;  }  In this example, we use the strtol() function to convert the octal string to an integer, specifying base 8 as the number system. The decimal equivalent of the octal number is then stored in the variable decimal\_num, and we print it using printf() function.  The strtol() function returns a long int value that represents the converted number. If the conversion is successful, it returns the converted number. If the string does not represent a valid number, strtol() sets the errno variable to indicate the error, and it returns 0. In this case, you can check the value of errno to determine the type of error that occurred.  However, the third argument of strtol() is a pointer to the first invalid character after the number. If the entire string was converted to a number, then \*endptr will be equal to '\0'. If the conversion was not successful, then \*endptr will point to the first invalid character in the string.  So, you can check the value of \*endptr to determine if the entire string was converted to a number or not. If \*endptr is equal to '\0', then the entire string was converted to a number. If \*endptr points to the first invalid character in the string, then the conversion was not successful.  #include <stdlib.h>  void intToHex(int num, char\* str) {  int i = 0;  while (num != 0) {  int remainder = num % 16;  str[i++] = (remainder < 10) ? remainder + '0' : remainder + 'A' - 10;  num /= 16;  }  str[i] = '\0';  // reverse the string  int len = i;  for (i = 0; i < len / 2; i++) {  char temp = str[i];  str[i] = str[len - i - 1];  str[len - i - 1] = temp;  }  }  #include <stdio.h>  int main() {  int num3 = 0x2A; // assign a hexadecimal integer value to num3  char str[10]; // declare a character array to hold the formatted string  intToHex(num3, str); // convert the integer to a hexadecimal string  printf("The hexadecimal value of num3 is %s.\n", str);  return 0;  } |