```
-------------------------------------------------
** Pointers **
-------------------------------------------------
    #include <stdio.h>
    #include <stdlib.h>

    int main(int argc, char **argv)
    {
        int tuna = 10;

        //This will print out the address of tuna
        printf ( “%p”, &tuna );
        //This will print out the name of tuna
        printf ( “%s”, “tuna”);
        //This will print out the value of tuna
        printf ( “%d”, tuna );


        Address     Name        Value
        0028FF1C    tuna        10

        //This will create a pointer to the address of tuna
        int * pTuna = &tuna;

        printf ( “%p”, &tuna );
        printf ( “%p”, pTuna );
        return(0);
    }

-------------------------------------------------
** Headers **
-------------------------------------------------

NOTE : #include "myownheaderfile.h" is used instead
of <myownheaderfile.h> when the header files
are found in the same directory.
    ---------------------------------------------
    [bucky.h]
    ---------------------------------------------
    #define MY_NAME “Brandon from Header”
    #define MY_AGE 20

    ---------------------------------------------
    [main.c]
    ---------------------------------------------
    #include <stdio.h>
    #include <stdlib.h>
    #include "bucky.h"

    int main() {

        //This will print out MY_NAME from bucky.h
        printf ( “My name is %s”,  MY_NAME );

        //This will use MY_AGE from bucky.h
        int girlsAge = (MY_AGE / 2 ) + 7;
        printf(“ %s can date girls aged %d or older” ,  MY_NAME , girlsAge);
    }



-------------------------------------------------
** Structures && Typedefs **
-------------------------------------------------

Structs and typdefs are used to represent a data
structure and to make life easier when handling
large sets of data with shared properties.

    ---------------------------------------------
    [bucky.h]
    ---------------------------------------------
    struct user{
```

```
        int userID;
        char firstName[25];
        char lastName[25];
        int userID;
        float userID;
    };

    typedef struct _Book{

        int bookID;
        char title[25];
        char author[25];
        float price;
    } Book;

    ---------------------------------------------
    [main.c]
    ---------------------------------------------

    #include <stdio.h>
    #include <stdlib.h>
    #include <bucky.h>

    int main() {

        struct user brandon;
        struct user john;

        Book  myBook;

        brandon.userID = 21;
        john.userID = 18;
    }


--------------------------------------------------
** Makefiles & Complining **
--------------------------------------------------

target: dependencies
    action

When a program is compiled source files ( main.c )
are compiled into object files ( main.o )
which are then further compiled into executables ( output ).

When using multiple source files, each individual
source file is compiled into individiual object files
( main.o , partb.o, partc.o ) these are then linked
together with any C libraries used and combined
together into one executable.


    ---------------------------------------------
    [Makefile]
    ---------------------------------------------

    output : main.o
        gcc main.o -o output

    main.o : main.c
        gcc -c main.c


--------------------------------------------------
** Memory Allocation (Malloc) **
--------------------------------------------------

    ---------------------------------------------
    [malloc1.c]
    ---------------------------------------------
    #include <stdio.h>
    #include <stdlib.h>
```

```
int main(int argc, char **argv)
{
    char *x=malloc(100);
    if (!x) {
        printf("Malloc failed!\n");
    } else {
        printf("Malloc succeeded!\n");
        free(x);
    }
    return(0);
}


---------------------------------------------
[malloc2.c]
---------------------------------------------

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


int main() {

    //Create pointer to an integer
    int * points;
    //Pointer  points to start of heap for 5 ints
    points = ( int *) malloc (5 * sizeof ( int ) );
    //Frees up memory back to PC
    free( points )
}

--------------------------------------------------
** User Input **
--------------------------------------------------


To read input from the user, you will need to use
the printf function aswell as the scanf function
to both print and read to/from the user.

    ------------------------------------------------
    [inputread.c]:

    ** This method uses the broken 'scanf' approach**
    -------------------------------------------------

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

    int main() {

        int a;

        printf(  “ Please enter an integer value :  ” );
        scanf(  “%d ”, &a);
        printf(  “ You entered: %d ” , a);

        return 0;
    }

    ------------------------------------------------
    [inputread2.c]:

    **This method uses the better 'fgets' approach**
    ------------------------------------------------

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

    int main() {

      printf("Enter a string: ");

      char buf[200];
```

```
  fgets(buf, 200, stdin);

  //Remove newline if present
  i = strlen(buf)-1;
            if( buf[ i ] == '\n')
                    buf[i] = '\0';


  printf( "\nYou entered: %s", buf);

}

------------------------------------------------
[inputread3.c]:

**This method processes command line args**
------------------------------------------------

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

int main() {

  printf("Enter a string: ");

  char buf[200];
  fgets(buf, 200, stdin);

  //Remove newline if present
  i = strlen(buf)-1;
            if( buf[ i ] == '\n')
                    buf[i] = '\0';


  printf( "\nYou entered: %s", buf);

}


//Process command line args
    for(n=i=1;i<argc;i=n) {
            n++;
            //If arg starts with '-' do this
            if (argv[i][0] == '-' && argv[i][1]) {
    //Process what letter comes after -
    for(j=1;argv[i][j];j++) {
                    switch(argv[i][j]) {
                    case 'h':
                      hostflag = true;
                      memcpy(host, argv[i+1], strlen(argv[i+1])+1);
                      break;
                    case 'p':
                      portflag = true;
                      port = atoi(argv[i+1]);
                      break;
                    case 'H':
                      helpflag = true;
                      break;
                    case 'w':
                      webflag = true;
                      memcpy(resource_path, argv[i+1], strlen(argv[i+1])+1);
                      break;
                    case 'f':
                      fileflag = true;
                      f.open(argv[i+1],ios::out);
                      break;
                    case '?':
                      helpflag = true;
                      break;
                    default:
                      //If no flag has been set, then invalid input
                      if(!hostflag && !portflag && !helpflag && !webflag && !fileflag) {
                              fprintf(stderr,"knock: Invalid argument -`%c'.\n",argv[i][j]);
                              usage(1);
```

```
                                exit(1);
                                break;
                        }
                }
                  }
                }
        }



    --------------------------------------------------
    ** Debugging - GDB **
    --------------------------------------------------

    A basic way of debugging your C programs is through
    the use of assert() statements.


        #include <assert.h>
        assert(condition expected to be true);


    You can also use various printf statements throughout
    your program to print the value of variables at
    certain points in an attempt to find the bug.

        /* Set to 0 to disable debug output, nonzero to enable. */
        #define DEBUG_INFO 1

        void buggy(int x) {
          int i;

          for (i = 0; i < x; i++) {

            #if DEBUG_INFO
              printf("i = %d, a[i] = %s\n", i, a[i]);
              fflush(stdout);
            #endif
          ... /* do buggy stuff with i and a */
          }
        }

    Probably the best method of all is to use the GNU Debugger
    (GDB). GDB allows you to invoke it upon your binary file.
    It includes functionality such as setting breakpoints,
    finding memory leaks, printing variables and much more.

        -----------------------------------------
        1. Compile your C file to a binary file
        -----------------------------------------

          $ gcc foo.c -o foo

        -----------------------------------------
        2. Invoke gdb once your binary file
        -----------------------------------------

          $ gdb foo

            ...gdb startup information...
            (gdb)

        -----------------------------------------
        3. Run gdb
        -----------------------------------------

          $ (gdb) run
            Starting program: foo...

            Program received signal SIGSEGV, Segmentation fault.
            0x08048546 in add_num (lst=0xbffff3c4, num=3) at numlist.c:16
            16 lst->tail->next = n;

        -----------------------------------------
```

```
4. Check surrounding code of error
-----------------------------------------

  $ (gdb) list

  [shows 10 lines of code]
  11 void add_num(list *lst, int num) {
  12 node *n = (node *) malloc(sizeof(node));
  13 n->value = num;
  14 n->next = NULL;
  15
  16 lst->tail->next = n;
  17
  18 lst->tail = n;
  19 lst->size++;
  20 }

-----------------------------------------
5. Check variable values around error
-----------------------------------------

  $ (gdb) print lst

  [Hmm, lst looks okay...]
  $1 = (list *) 0xbffff3c4
  (gdb) print *lst
  $2 = {head = 0x0, tail = 0x0, size = 0}
  (gdb) print lst->tail
  [Ah ha! lst->tail is NULL.]
  $3 = (node *) 0x0
  (gdb) print n
  $4 = (node *) 0x804b008
  (gdb) print *n
  [Well, at least n is okay...]
  $5 = {value = 3, next = 0x0}

-----------------------------------------
6. Set breakpoints
-----------------------------------------

  $ gdb foo

  ...gdb startup information...

  //Set breakpoint in reverse():
  $ (gdb) break reverse

  Breakpoint 1 at 0x80485eb: file numlist.c, line 50.

-----------------------------------------
7. Run to breakpoint
-----------------------------------------

  $ (gdb) run
    Starting program: nummain...

    Original list: 3 1 4 1 5
    Breakpoint 1, reverse (lst=0xbffff3c4) at numlist.c:50
    50 prev = NULL;

  //Step over
  $ (gdb) next

    51 curr = lst->head;

  //Step over
  $ (gdb) next

    53 while (curr != NULL) {

  //Step over
  $ (gdb) next

    54 next = curr->next;
```

```
------------------------------------------------
** Debugging - Valgrind **
------------------------------------------------

Valgrind is a tool that can be used to check memory
allocation within your program and to ensure that
there are no memory leaks or incorrect memory writes
present.

-If you normally run your program like this:

  $ myprog arg1 arg2

-Use this command line:

  $ valgrind --leak-check=yes myprog arg1 arg2

-This may produce an error message that looks like this:

  $ valgrind --leak-check=yes myprog arg1 arg2

  ==19182== Invalid write of size 4
  ==19182==    at 0x804838F: f (example.c:6)
  ==19182==    by 0x80483AB: main (example.c:11)
  ==19182==  Address 0x1BA45050 is 0 bytes after a block of size 40 allocd
  ==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
  ==19182==    by 0x8048385: f (example.c:5)
  ==19182==    by 0x80483AB: main (example.c:11)
```