gcc

`gcc` uses include files located in `/usr/include`. These files can be viewed to find out structure definitions. `gcc` also uses libraries located in `/usr/lib`. Again, copious amounts of information on `gcc` can be found using `man` as well as for any ANSI C function.

`gcc` addes debugging information in the output program when the `-g` flag is used:

```
# cc -g my_c_file.c
```

The debugging information allows users to debug a program using `gdb` or other C source level debugger. For example, type in the following code and save it as `l06a.c`:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char * argv[])
{
        int temp;
        char * pointer;

        printf("Using gdb to debug a program\n");
        pointer = (char *) malloc(80);

        strcpy(pointer,"More Text");

        printf("%s\n",pointer);
        pointer=NULL;

        pointer[0]='E';
        printf("%c\n",pointer[0]);

        return 1;
}
```

This code sets the pointer `pointer` to an invalid address of `0x0`. On the PC, whether its Windows XP or Windows 8, the program will fail when the program runs. In Linux, this program will cause a segmentation fault, possibly dumping a core (memory image of the program) and then exiting. The rest of the system is completely unaffected. To effectively debug programs that cause segmentation faults in Linux, a core file is required. Type the following to allow core dumps:

```
# unlimit
```

Compile the above code like this:

```
# gcc -g -o seg_fault l06a.c
```

The `-o` flag tells `cc` to name the executable the name following the `-o` flag rather than `a.out`. Run the program and you will get this:

```
# ./seg_fault
Using gdb to debug a program
More Text
Segmentation fault (core dumped)
```

Now use **gdb** to figure out where the error is:

```
# gdb seg_fault core
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
```

```
GDB is free software, covered by the GNU General Public License, and
you are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
details.
This GDB was configured as "i386-redhat-linux"...
Core was generated by `./a.out'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Reading symbols from /lib/ld-linux.so.2...done.
#0  0x804848d in main () at x.c:17
17              pointer[0]="E";
(gdb)
```

The error was at this line `pointer[0]='E';`. Use `gdb` to print the value of `pointer`, because that is the only pointer on that line that can cause a segmentation fault.

```
(gdb) print pointer
$1 = 0x0
```

meaning `pointer` is zero - its pointing to memory location `0x0`. This is bad. Quit gdb, edit the source file and remove the line `pointer=NULL;`. Recompile the code (rename to *no_seg_fault*) and use `gdb` to step through the code:

```
(gdb) quit
# vi l06a.c
# gcc -g -o no_seg_fault l06a.c
# gdb no_seg_fault
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and
you are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
details.
This GDB was configured as "i386-redhat-linux"...
(gdb) break main
Breakpoint 1 at 0x804843e: file x.c, line 9.
```

Set a break point at the entry to the program and run the program:

```
(gdb) break main
(gdb) run
Starting program: /home/sheaff/a.out

Breakpoint 1, main () at x.c:9
9               printf("Using gdb to debug a program\n");
```

Step through the code and print out variables:

```
(gdb) print pointer
$1 = 0x8049530 ""
(gdb) next
```

```
Using gdb to debug a program
10              pointer = (char *) malloc(80);
```

gdb prints the next line to be executed after each next command.

```
(gdb) next
12              strcpy(pointer,"More Text");
(gdb) print pointer
$2 = 0x80497c0 ""
```

The malloc() has be executed and pointer now has a valid value.

```
(gdb) next
14              printf("%s\n",pointer);
(gdb) next
More Text
17              pointer[0]="E";
(gdb) next
18              printf("%c\n",pointer[0]);
(gdb) next
 E
19              }
(gdb) print pointer[0]
$3 = 69 'E'
(dbx) cont
Continuing.

Program exited with code 01.
```

gdb can move up and down the stack to show arguments to functions. The stack is where function call information is stored. Use up and down to move on the stack.