ECE 3220 Lab 5

Max Houck

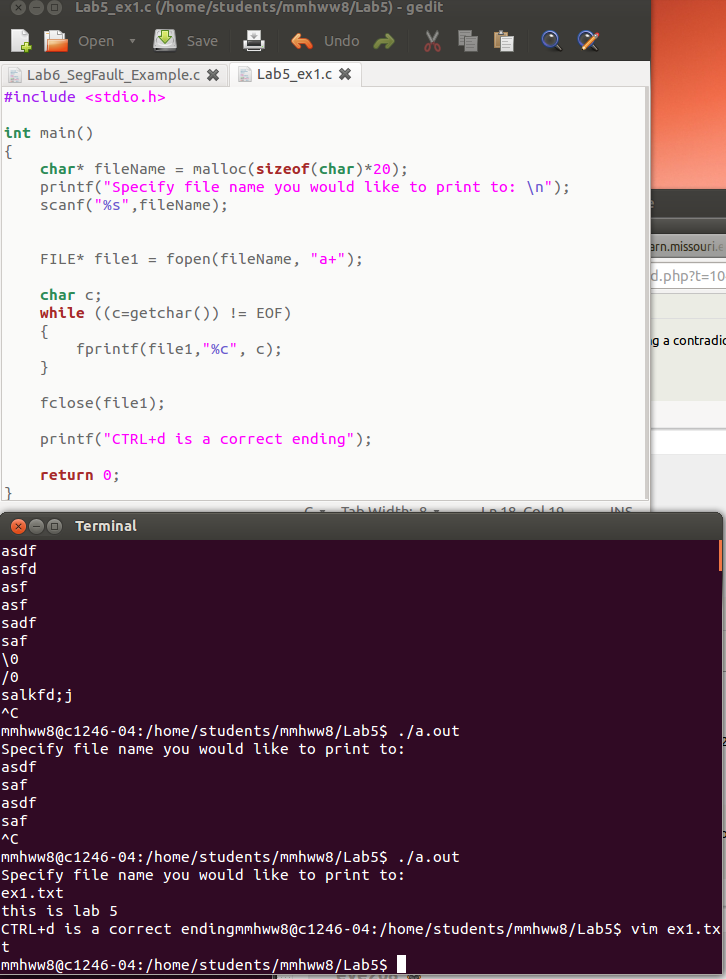
2/20/2017

Objective: The objective of this lab was to learn how to use gdb in a programming environment to debug issues with our code and fix them, specifically segmentation faults. Segmentation faults are hard to track down without using a debugger like gdb.

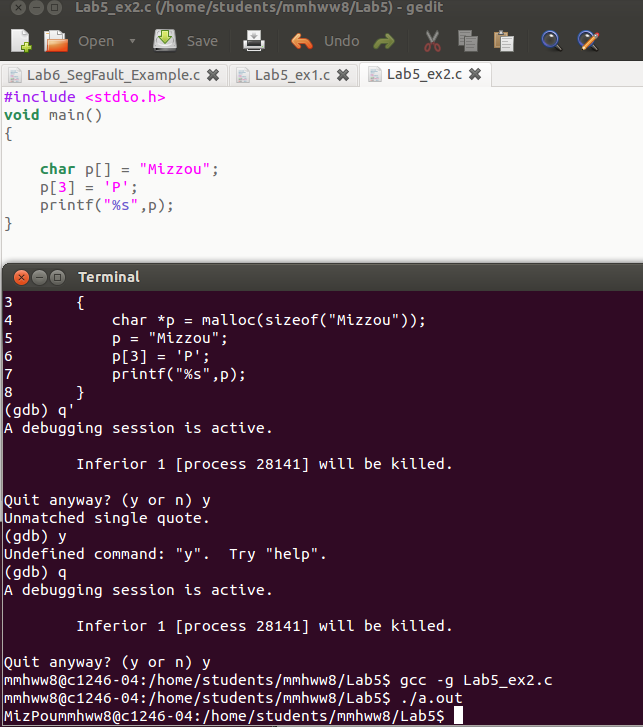
Results and Discussion:

For most examples, all I had to do was run the code and use gdb and backtrace to figure out where the issue was. Then, after simply looking at that line, most of the time I could identify what the issue was and fix it.

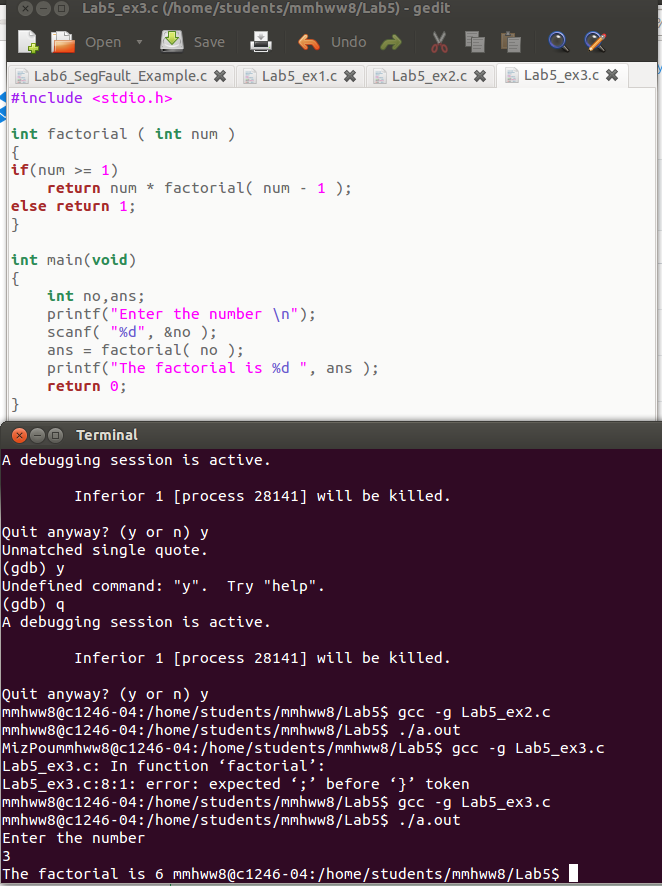
For the first example, I had to change the first line of code to malloc space for the filename string, so that the computer could properly allocate memory for it. I also had to change the syntax in the fprintf command to “%c” for the second argument so that it would write the proper data into the file.



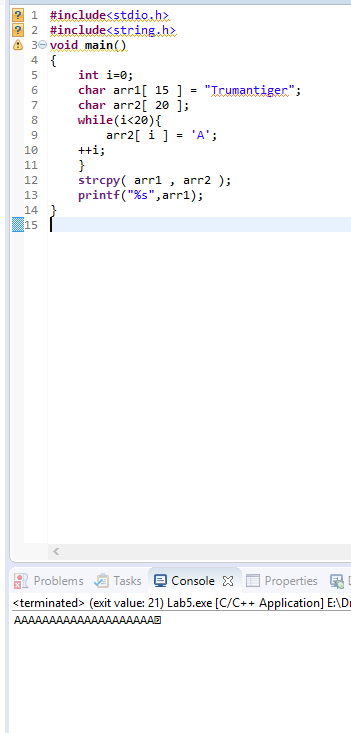
In the second example I had to change the character pointer to be declared as an array. The way it was before, it was a string literal and was seg faulting because string literals cannot be modified. After changing to an array, it is mutable and the program executed correctly.



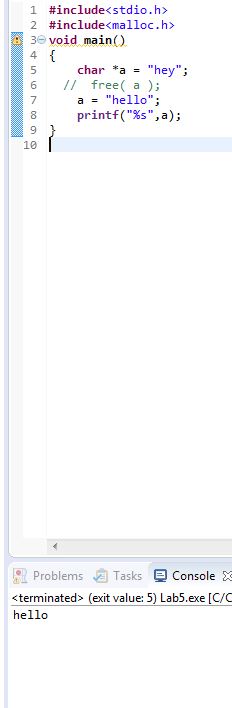
In the third example, the program never included a statement to end the recursion. I added a simple if/else statement in order to properly use recursion for the intended function, and then the program worked properly.



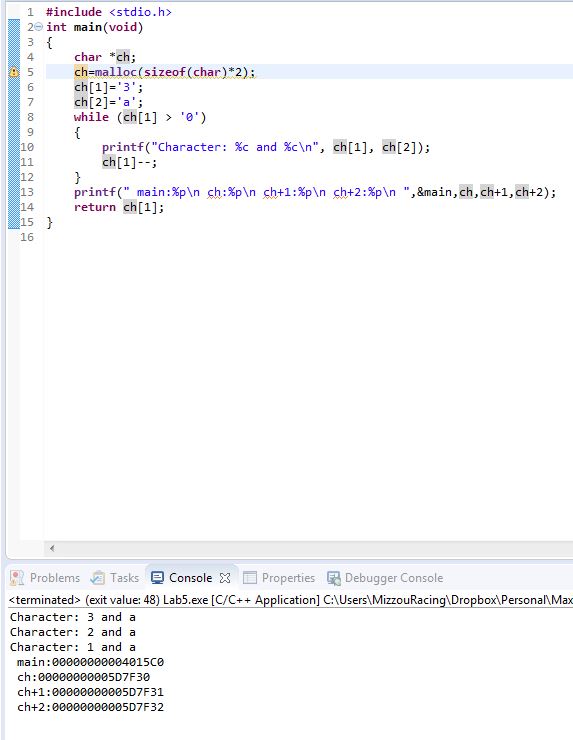
When running example 4 in eclipse and using MinGW on my Windows machine, this program executed without issuing any segmentation faults. The code seemed to do what it was intended to, so I did not make any changes to it.



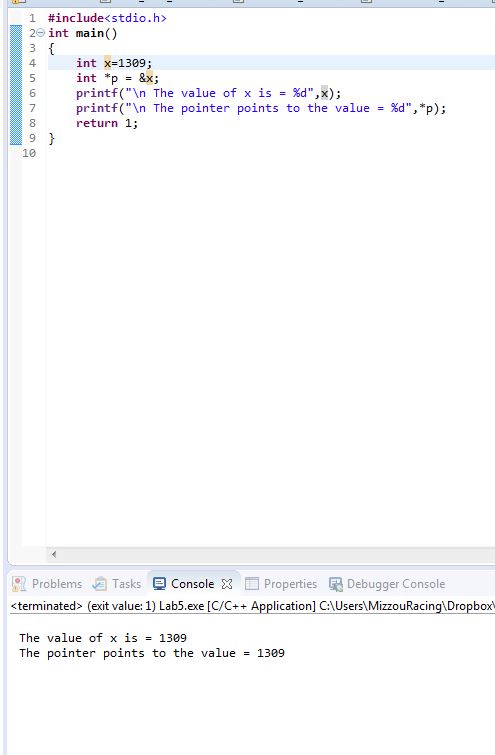
The issue with example 5 was that the code tried to free a portion of memory that was not allocated using the malloc function. To fix the code I could have either omitted the free() line or malloced the character pointer ‘a’.



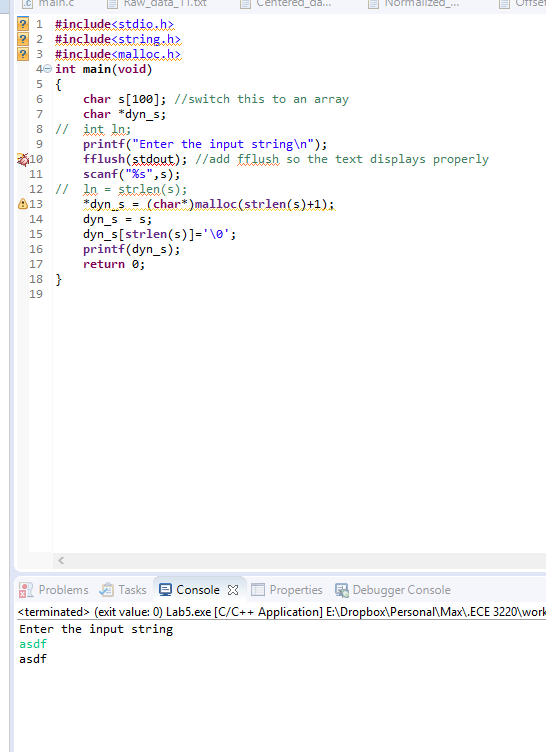
In example 6, on line 5 the original code dereferenced the character when trying to malloc space. When we call malloc, we want to set the pointer (memory) to the allocated memory, so this should not be dereferenced. After removing the asterisk, the program executed correctly.



In example 7, the issue was that you are trying to print the value at a null pointer. This causes a seg fault when the program is executed. I fixed this by simply assigning a real value to the pointer ‘p’. In this case, I used the address of the previously defined variable, x. After making this change, the printf statement on line 7 shows the value at the address pointed to by p, which is 1309 (x).



Finally, in example 8, I had to switch the first line to declare s as an array, so that it could be used without malloc. I also got rid of the int ln statements, since they weren’t actually used at all and were giving compiler warnings. Finally, I added an fflush() so that the printf would display properly in the terminal. After making these changed, the program executed as intended.



Conclusion: This lab was very helpful to refamiliarize me with GDB. The last time I had used it was in CS2050, and it was nice to brush up on my skills. It will be a very good tool to have as our code gets more complicated and harder to debug using more “primitive” methods. Using gdb in combination with Eclipse’s debugger (which is actually just a GUI on top of gdb) has been my strategy for debugging my code for the last couple labs and it was worked well for me. Knowing how to use the command line interface is also important however, as it can be very powerful and give me a better context for what Eclipse is doing with my code.

Source:

Example 1:

#include <stdio.h>

int main()

{

char\* fileName = malloc(sizeof(char)\*20); //use the malloc function here

printf("Specify file name you would like to print to: \n");

scanf("%s",fileName);

FILE\* file1 = fopen(fileName, "a+");

char c;

while ((c=getchar()) != EOF)

{

fprintf(file1,"%c", c); //fixed syntax of second argument

}

fclose(file1);

printf("CTRL+d is a correct ending");

return 0;

}

Example 2:

#include <stdio.h>

void main()

{

char p[] = "Mizzou"; //changed to be declared as an array

p[3] = 'P';

printf("%s",p);

}

Example 3:

#include <stdio.h>

int factorial ( int num )

{

if(num >= 1) //added an if else statement to use simple recursion properly

return num \* factorial( num - 1 );

else return 1;

}

int main(void)

{

int no,ans;

printf("Enter the number \n");

scanf( "%d", &no );

ans = factorial( no );

printf("The factorial is %d ", ans );

return 0;

}

Example 4:

#include<stdio.h>

#include<string.h>

void main()

{

int i=0;

char arr1[ 15 ] = "Trumantiger";

char arr2[ 20 ];

while(i<20){

arr2[ i ] = 'A';

++i;

}

strcpy( arr1 , arr2 );

printf("%s",arr1);

}

Example 5:

#include<stdio.h>

#include<malloc.h>

void main()

{

char \*a = "hey";

//free( a ); //remove this line and the code will execute

a = "hello";

printf("%s",a);

}

Example 6:

#include <stdio.h>

int main(void)

{

char \*ch;

ch=malloc(sizeof(char)\*2); //removed dereferencing operator here

ch[1]='3';

ch[2]='a';

while (ch[1] > '0')

{

printf("Character: %c and %c\n", ch[1], ch[2]);

ch[1]--;

}

printf(" main:%p\n ch:%p\n ch+1:%p\n ch+2:%p\n ",&main,ch,ch+1,ch+2);

return ch[1];

}

Example 7:

#include<stdio.h>

int main()

{

int x=1309;

int \*p = &x; //changed from null

printf("\n The value of x is = %d",x);

printf("\n The pointer points to the value = %d",\*p);

return 1;

}

Example 8:

#include<stdio.h>

#include<string.h>

#include<malloc.h>

int main(void)

{

char s[100]; //switch this to an array

char \*dyn\_s;

// int ln;

printf("Enter the input string\n");

fflush(stdout); //add fflush so the text displays properly

scanf("%s",s);

// ln = strlen(s);

\*dyn\_s = (char\*)malloc(strlen(s)+1);

dyn\_s = s;

dyn\_s[strlen(s)]='\0';

printf(dyn\_s);

return 0;

}

Github: