Brandon London

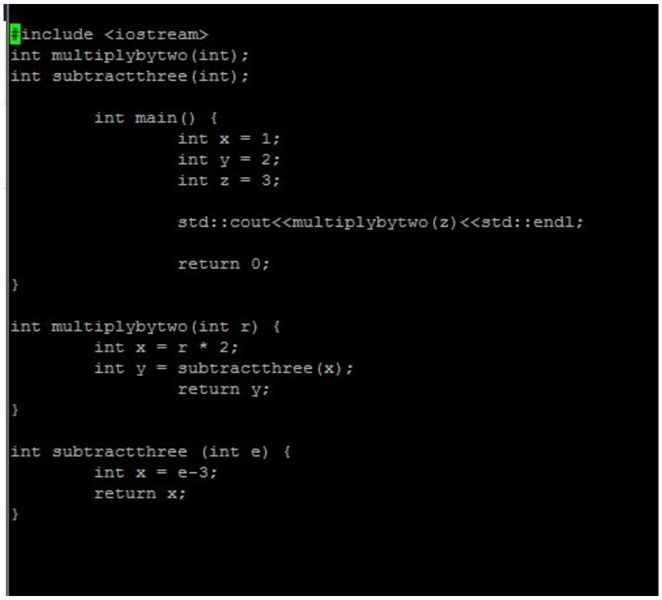
Due date: 9/17/2019

Total points possible: 65

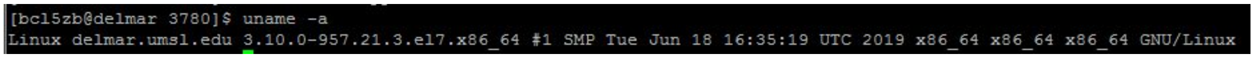
The goal of this project is to reinforce what is going on in the call stack, as well as refreshing ourselves on the unix environment and using a debugger to examine memory locations. All tasks should be done on the student unix server delmar.umsl.edu. If you have never used a debugger to examine raw memory locations, now is the time!

Task 1

Your task here is to examine the size of the activation record for a function. In particular, I want to know the size in bytes on the stack that is taken up by all the data that is not simply local variables being stored. To do this, write a C++ program that contains several functions that call each other in a sequence. These functions should take in at least one variable by value. It would also be useful to set some local variables to some specific values inside of them. Then using GDB or some other debugger, set a breakpoint at the end of the lowest-level function and examine the memory locations of the stack (probably using x in gdb). You should be able to get an understanding of how much space is taken up by non-variable data. Note that you should keep these functions fairly simple as it will make your life easier. For demonstrating your work, I want you to write up a justification for your answer, including relevant evidence. This should include screen captures of at least some display by the debugger of memory, along with a description of its interpretation.

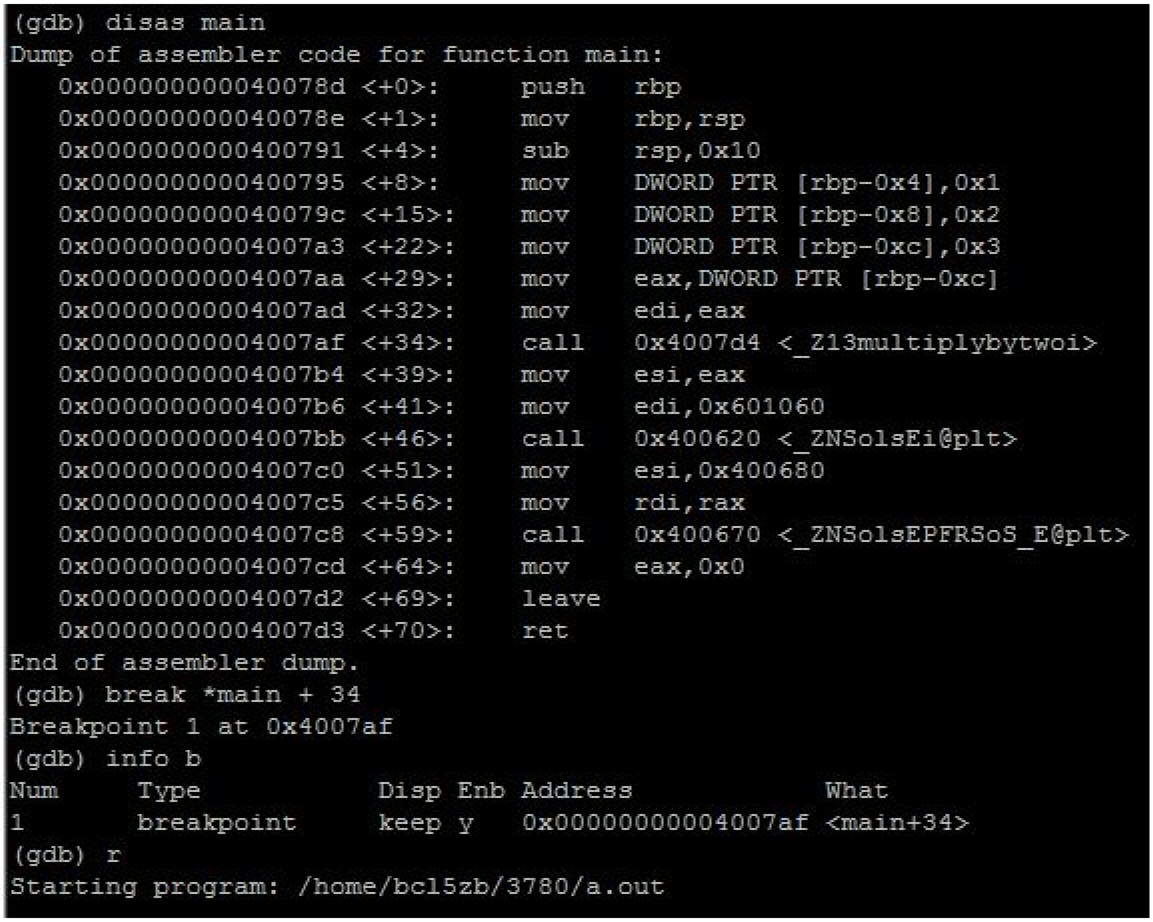


```
#include <iostream>
int multiplybytwo(int);
int subtractthree(int);

        int main() {
                int x = 1;
                int y = 2;
                int z = 3;

                std::cout<<multiplybytwo(z)<<std::endl;

                return 0;
}

int multiplybytwo(int r) {
        int x = r * 2;
        int y = subtractthree(x);
                return y;
}

int subtractthree (int e) {
        int x = e-3;
        return x;
}
```



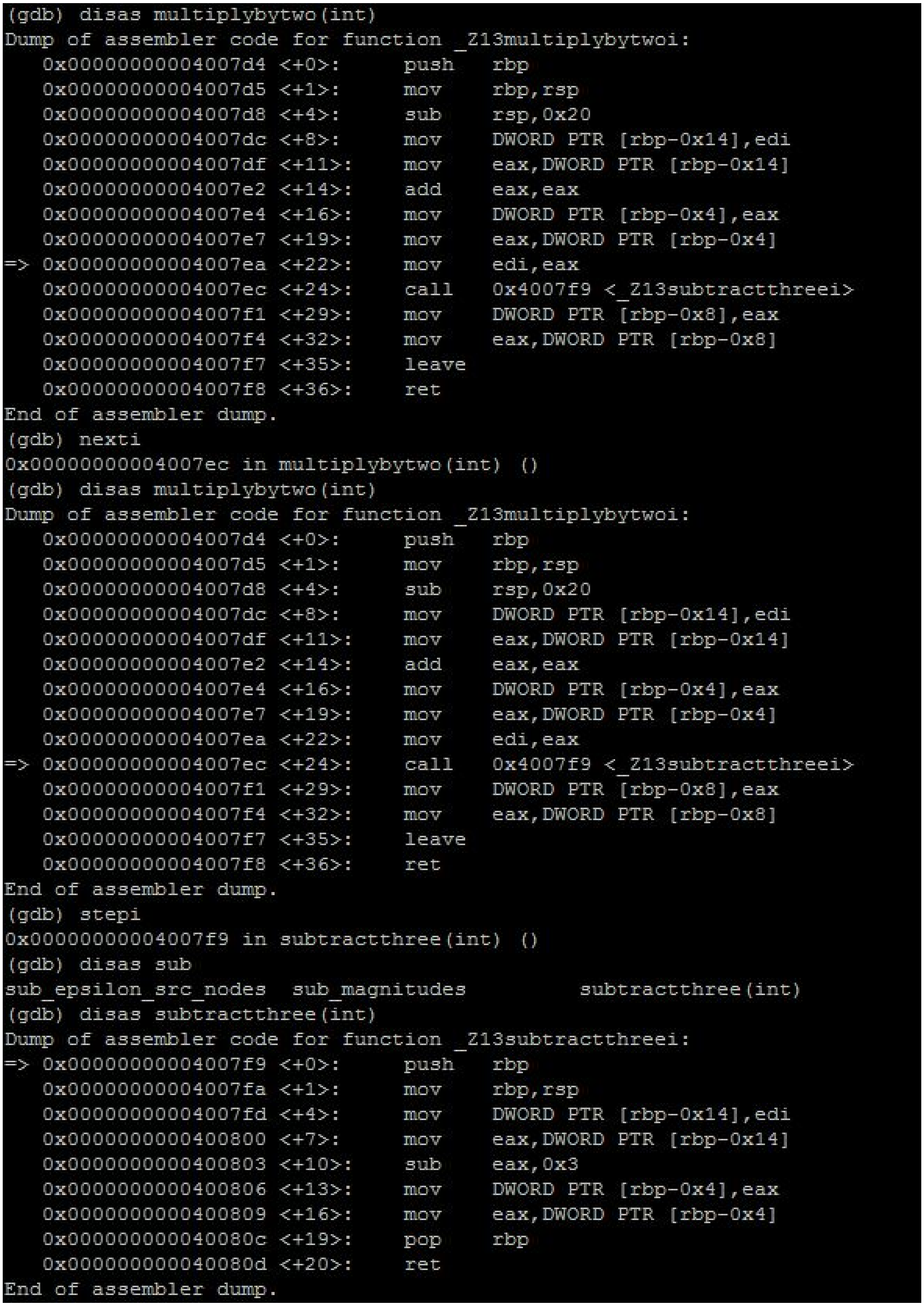
```
[bcl5zb@delmar 3780]$ uname -a
Linux delmar.umsl.edu 3.10.0-957.21.3.el7.x86_64 #1 SMP Tue Jun 18 16:35:19 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux
```

The first step should be that we need to check the amount of Bytes in the registers This can be accomplished by the command **uname -a**. From the picture above we can see that this is a 86 _64 machine that will contain 8 Byte registers. Next we can use gdb to look into the stack of my task1.cpp program. We can get the stack from the function **disas main**.



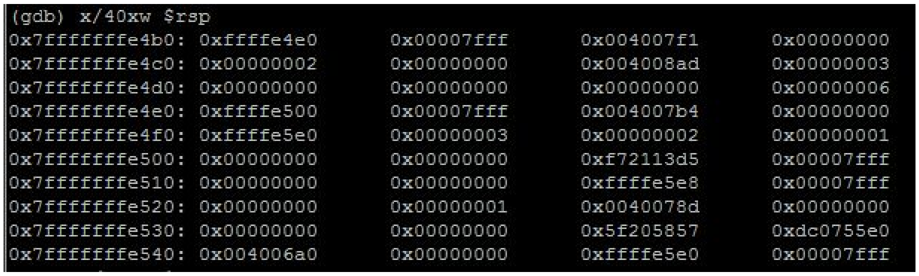
```
(gdb) disas main
Dump of assembler code for function main:
   0x000000000040078d <+0>:     push   rbp
   0x000000000040078e <+1>:     mov    rbp,rsp
   0x0000000000400791 <+4>:     sub    rsp,0x10
   0x0000000000400795 <+8>:     mov    DWORD PTR [rbp-0x4],0x1
   0x000000000040079c <+15>:    mov    DWORD PTR [rbp-0x8],0x2
   0x00000000004007a3 <+22>:    mov    DWORD PTR [rbp-0xc],0x3
   0x00000000004007aa <+29>:    mov    eax,DWORD PTR [rbp-0xc]
   0x00000000004007ad <+32>:    mov    edi,eax
   0x00000000004007af <+34>:    call   0x4007d4 <_Z13multiplybytwoi>
   0x00000000004007b4 <+39>:    mov    esi,eax
   0x00000000004007b6 <+41>:    mov    edi,0x601060
   0x00000000004007bb <+46>:    call   0x400620 <_ZNSolsEi@plt>
   0x00000000004007c0 <+51>:    mov    esi,0x400680
   0x00000000004007c5 <+56>:    mov    rdi,rax
   0x00000000004007c8 <+59>:    call   0x400670 <_ZNSolsEPFRSoS_E@plt>
   0x00000000004007cd <+64>:    mov    eax,0x0
   0x00000000004007d2 <+69>:    leave
   0x00000000004007d3 <+70>:    ret
End of assembler dump.
(gdb) break *main + 34
Breakpoint 1 at 0x4007af
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004007af <main+34>
(gdb) r
Starting program: /home/bcl5zb/3780/a.out
```

As we can see the first thing to be pushed is the rbp which is the base pointer. As you can see the rsp is being put into the rbp holding the memory address of the current position of the stack. We have to step into each function until we are at the lowest function in the stack.



```
(gdb) disas multiplybytwo(int)
Dump of assembler code for function _Z13multiplybytwoi:
   0x00000000004007d4 <+0>:     push   rbp
   0x00000000004007d5 <+1>:     mov    rbp,rsp
   0x00000000004007d8 <+4>:     sub    rsp,0x20
   0x00000000004007dc <+8>:     mov    DWORD PTR [rbp-0x14],edi
   0x00000000004007df <+11>:    mov    eax,DWORD PTR [rbp-0x14]
   0x00000000004007e2 <+14>:    add    eax,eax
   0x00000000004007e4 <+16>:    mov    DWORD PTR [rbp-0x4],eax
   0x00000000004007e7 <+19>:    mov    eax,DWORD PTR [rbp-0x4]
=> 0x00000000004007ea <+22>:    mov    edi,eax
   0x00000000004007ec <+24>:    call   0x4007f9 <_Z13subtractthreei>
   0x00000000004007f1 <+29>:    mov    DWORD PTR [rbp-0x8],eax
   0x00000000004007f4 <+32>:    mov    eax,DWORD PTR [rbp-0x8]
   0x00000000004007f7 <+35>:    leave
   0x00000000004007f8 <+36>:    ret
End of assembler dump.
(gdb) nexti
0x00000000004007ec in multiplybytwo(int) ()
(gdb) disas multiplybytwo(int)
Dump of assembler code for function _Z13multiplybytwoi:
   0x00000000004007d4 <+0>:     push   rbp
   0x00000000004007d5 <+1>:     mov    rbp,rsp
   0x00000000004007d8 <+4>:     sub    rsp,0x20
   0x00000000004007dc <+8>:     mov    DWORD PTR [rbp-0x14],edi
   0x00000000004007df <+11>:    mov    eax,DWORD PTR [rbp-0x14]
   0x00000000004007e2 <+14>:    add    eax,eax
   0x00000000004007e4 <+16>:    mov    DWORD PTR [rbp-0x4],eax
   0x00000000004007e7 <+19>:    mov    eax,DWORD PTR [rbp-0x4]
   0x00000000004007ea <+22>:    mov    edi,eax
=> 0x00000000004007ec <+24>:    call   0x4007f9 <_Z13subtractthreei>
   0x00000000004007f1 <+29>:    mov    DWORD PTR [rbp-0x8],eax
   0x00000000004007f4 <+32>:    mov    eax,DWORD PTR [rbp-0x8]
   0x00000000004007f7 <+35>:    leave
   0x00000000004007f8 <+36>:    ret
End of assembler dump.
(gdb) stepi
0x00000000004007f9 in subtractthree(int) ()
(gdb) disas sub
sub_epsilon_src_nodes  sub_magnitudes         subtractthree(int)
(gdb) disas subtractthree(int)
Dump of assembler code for function _Z13subtractthreei:
=> 0x00000000004007f9 <+0>:     push   rbp
   0x00000000004007fa <+1>:     mov    rbp,rsp
   0x00000000004007fd <+4>:     mov    DWORD PTR [rbp-0x14],edi
   0x0000000000400800 <+7>:     mov    eax,DWORD PTR [rbp-0x14]
   0x0000000000400803 <+10>:    sub    eax,0x3
   0x0000000000400806 <+13>:    mov    DWORD PTR [rbp-0x4],eax
   0x0000000000400809 <+16>:    mov    eax,DWORD PTR [rbp-0x4]
   0x000000000040080c <+19>:    pop    rbp
   0x000000000040080d <+20>:    ret
End of assembler dump.
```

In the above screenshot I use the debugger to step into each of the functions until I reach the lowest function on the stack, in this case it is the function subtract three and I keep stepping into subtract three until I reach +16. Once we are there we can look into the next 40 elements in hexadecimal, half word using the command **x/40xw $rsp**. We can see that some function had to call main. Based on the program we can see that 1, 2, 3 were called early in the main. We are at the lowest called function and as we keep and eye and know the addresses., we believe that 0x7fffffffe500 was one of the first things pushed on the stack in main. Now that we are in the lowest function the address 0x7fffffffe4b0 is where the stack pointer is currently. Then we subtract these two to find the difference that the memory takes up. When we subtract those two memory addresses we get 80 Bytes which is the size of the stack for our program.



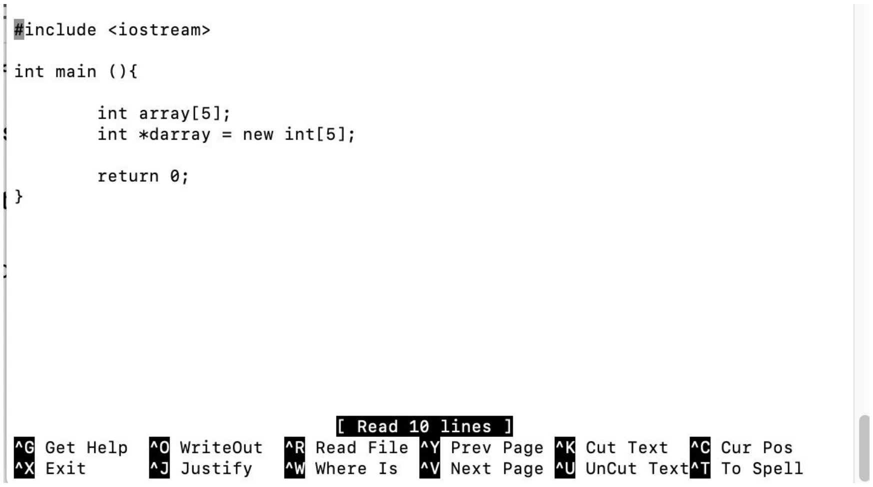
```
(gdb) x/40xw $rsp
0x7fffffffe4b0: 0xffffe4e0      0x00007fff      0x004007f1      0x00000000
0x7fffffffe4c0: 0x00000002      0x00000000      0x004008ad      0x00000003
0x7fffffffe4d0: 0x00000000      0x00000000      0x00000000      0x00000006
0x7fffffffe4e0: 0xffffe500      0x00007fff      0x004007b4      0x00000000
0x7fffffffe4f0: 0xffffe5e0      0x00000003      0x00000002      0x00000001
0x7fffffffe500: 0x00000000      0x00000000      0xf72113d5      0x00007fff
0x7fffffffe510: 0x00000000      0x00000000      0xffffe5e8      0x00007fff
0x7fffffffe520: 0x00000000      0x00000001      0x0040078d      0x00000000
0x7fffffffe530: 0x00000000      0x00000000      0x5f205857      0xdc0755e0
0x7fffffffe540: 0x004006a0      0x00000000      0xffffe5e0      0x00007fff
```

```
Code:#include <iostream>
int multiplybytwo(int);
int subtractthree(int);

    int main() {
        int x = 1;
        int y = 2;
        int z = 3;

        std::cout<<multiplybytwo(z)<<std::endl;

        return 0;
}

int multiplybytwo(int r) {
    int x = r * 2;
    int y = subtractthree(x);
        return y;
}

int subtractthree (int e) {
    int x = e-3;
```

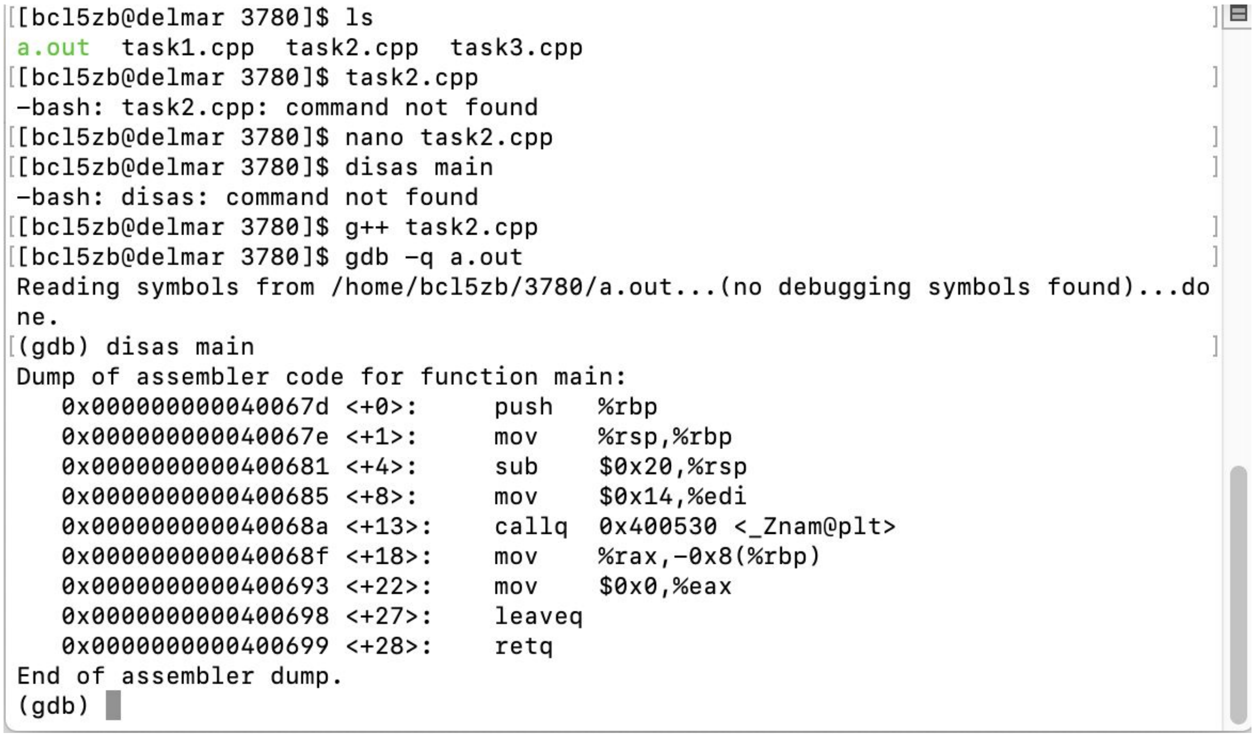
```
        return x;
}
```

Task 2:

Write another program (in C++) that will allocate a local static array of integers and then a dynamic array of integers. Are they stored next to each other? You can examine this by examining the memory addresses where they are located. As described in class, on some systems the size of a dynamic array is actually stored in the bytes previous to a dynamically allocated array. Through some experiments on your own, try to see if this is true on delmar. Is this true or not true also for the local array? As in the first part, describe the procedure you used to test for this.



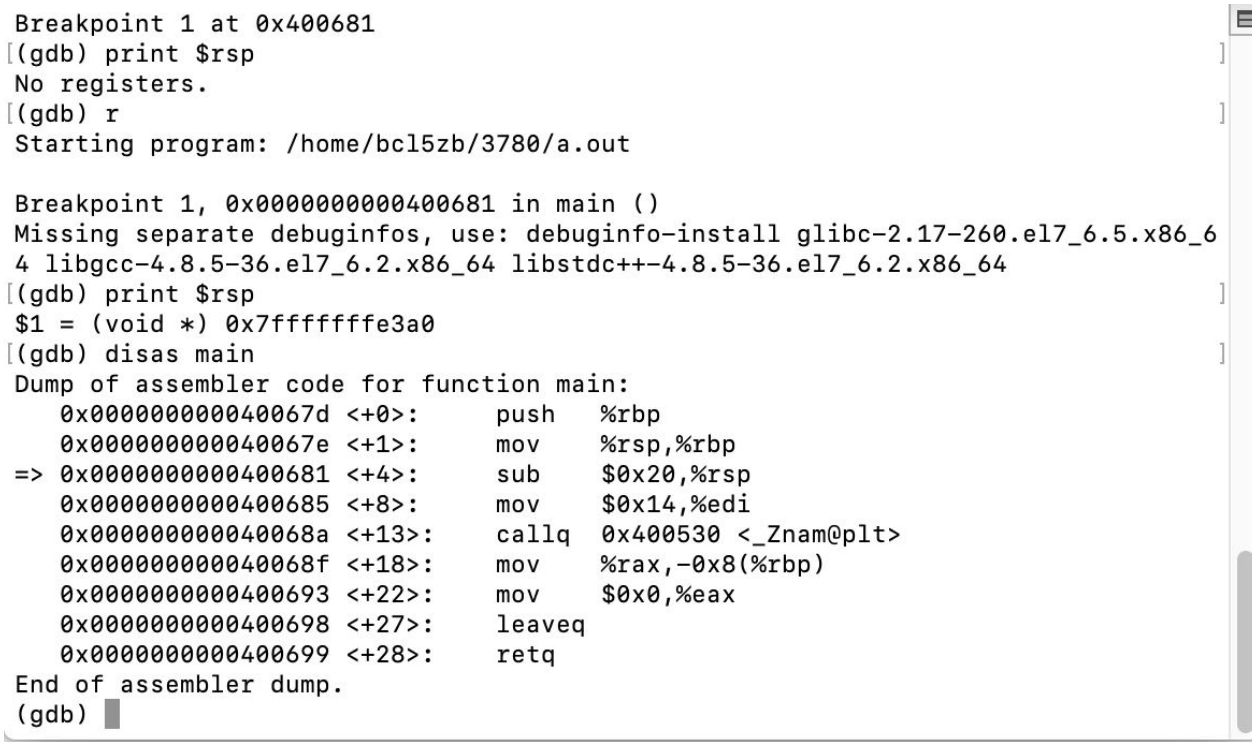
```
#include <iostream>

int main (){

        int array[5];
        int *darray = new int[5];

        return 0;
}
```



```
[bcl5zb@delmar 3780]$ ls
a.out  task1.cpp  task2.cpp  task3.cpp
[bcl5zb@delmar 3780]$ task2.cpp
-bash: task2.cpp: command not found
[bcl5zb@delmar 3780]$ nano task2.cpp
[bcl5zb@delmar 3780]$ disas main
-bash: disas: command not found
[bcl5zb@delmar 3780]$ g++ task2.cpp
[bcl5zb@delmar 3780]$ gdb -q a.out
Reading symbols from /home/bcl5zb/3780/a.out...(no debugging symbols found)...do
ne.
(gdb) disas main
Dump of assembler code for function main:
   0x000000000040067d <+0>:     push   %rbp
   0x000000000040067e <+1>:     mov    %rsp,%rbp
   0x0000000000400681 <+4>:     sub    $0x20,%rsp
   0x0000000000400685 <+8>:     mov    $0x14,%edi
   0x000000000040068a <+13>:    callq  0x400530 <_Znam@plt>
   0x000000000040068f <+18>:    mov    %rax,-0x8(%rbp)
   0x0000000000400693 <+22>:    mov    $0x0,%eax
   0x0000000000400698 <+27>:    leaveq
   0x0000000000400699 <+28>:    retq
End of assembler dump.
(gdb) 
```

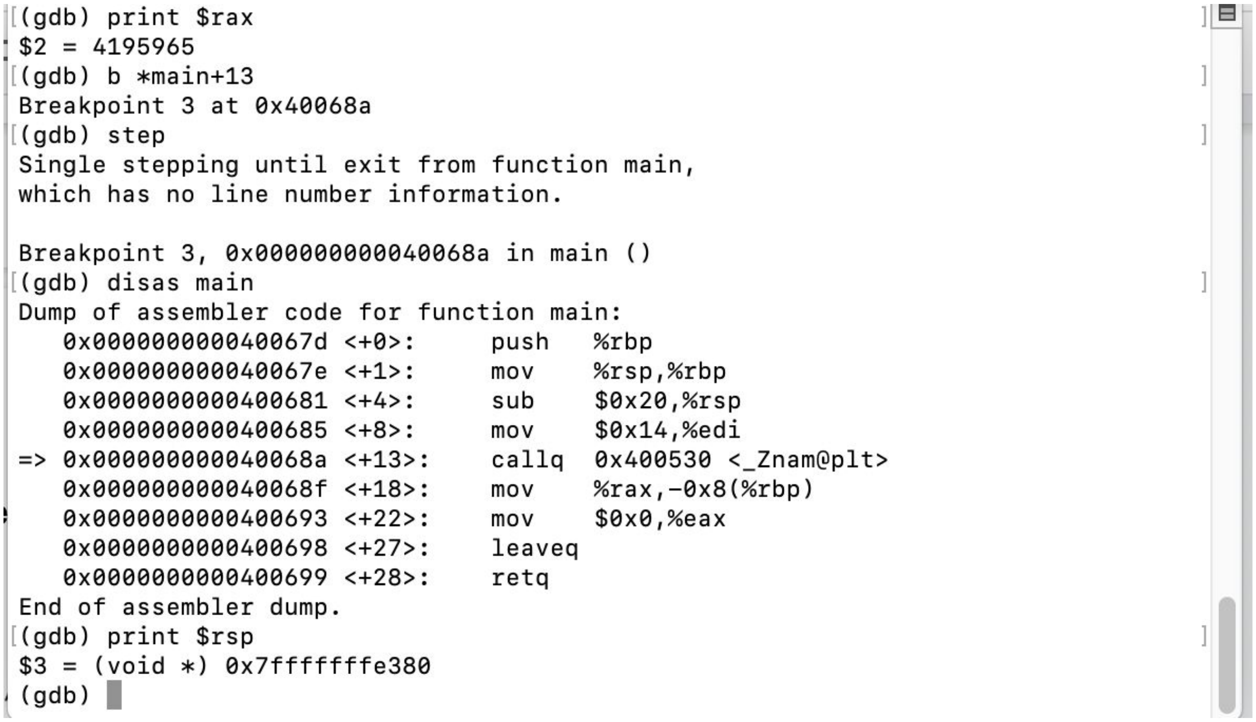
The first step is again to start up gdb. Once the gdb is started we can disassemble the main function, Note we do not have any function other than main unlike the last question.

My program only has a few lines of disassembly code. Line one is the base pointer and line two is the stack pointer. Those are going to be the same for every program as they are the configuration for the stack. Line three is going to subtract 32 from our stack pointer, this is for the local static array. I figured you could use the command **b *main +4** and **print $rsp** this will give us the beginning of the local array which would be **0xfffffffe3a0** as you can see below.



```
Breakpoint 1 at 0x400681
(gdb) print $rsp
No registers.
(gdb) r
Starting program: /home/bcl5zb/3780/a.out

Breakpoint 1, 0x0000000000400681 in main ()
Missing separate debuginfos, use: debuginfo-install glibc-2.17-260.el7_6.5.x86_6
4 libgcc-4.8.5-36.el7_6.2.x86_64 libstdc++-4.8.5-36.el7_6.2.x86_64
(gdb) print $rsp
$1 = (void *) 0x7fffffffe3a0
(gdb) disas main
Dump of assembler code for function main:
   0x000000000040067d <+0>:     push   %rbp
   0x000000000040067e <+1>:     mov    %rsp,%rbp
=> 0x0000000000400681 <+4>:     sub    $0x20,%rsp
   0x0000000000400685 <+8>:     mov    $0x14,%edi
   0x000000000040068a <+13>:    callq  0x400530 <_Znam@plt>
   0x000000000040068f <+18>:    mov    %rax,-0x8(%rbp)
   0x0000000000400693 <+22>:    mov    $0x0,%eax
   0x0000000000400698 <+27>:    leaveq
   0x0000000000400699 <+28>:    retq
End of assembler dump.
(gdb) 
```

The dynamic array is located in the heap, in line 4 of our disassembly code, it moves 20 into the EDI register and then calls a function located at **0x400530**. This is malloc however in this machine it is called **_znam@plt**, it has to be that because we are not calling anything else. EDI gets the value 20 because the number of bytes passed to new array. In the program we requested for 5 int so that would be 20 in decimal or **0x14** in hex. If we set another breakpoint on line 5 we can return the address that the array gave to our pointer. This is where the dynamic array is now on the heap.



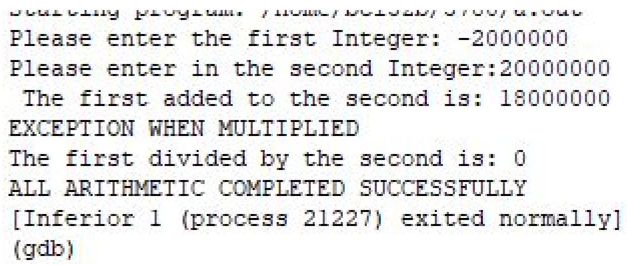
```
(gdb) print $rax
$2 = 4195965
(gdb) b *main+13
Breakpoint 3 at 0x40068a
(gdb) step
Single stepping until exit from function main,
which has no line number information.

Breakpoint 3, 0x000000000040068a in main ()
(gdb) disas main
Dump of assembler code for function main:
   0x000000000040067d <+0>:     push   %rbp
   0x000000000040067e <+1>:     mov    %rsp,%rbp
   0x0000000000400681 <+4>:     sub    $0x20,%rsp
   0x0000000000400685 <+8>:     mov    $0x14,%edi
=> 0x000000000040068a <+13>:    callq  0x400530 <_Znam@plt>
   0x000000000040068f <+18>:    mov    %rax,-0x8(%rbp)
   0x0000000000400693 <+22>:    mov    $0x0,%eax
   0x0000000000400698 <+27>:    leaveq
   0x0000000000400699 <+28>:    retq
End of assembler dump.
(gdb) print $rsp
$3 = (void *) 0x7fffffffe380
(gdb) 
```

These two arrays would obviously will not be next to each other. The static array address is at **0xfffffffe3a0** and the dynamic array array is at **0x7fffffffe380**. I could not find anything on the size of the dynamic array when examining the memory before the array.

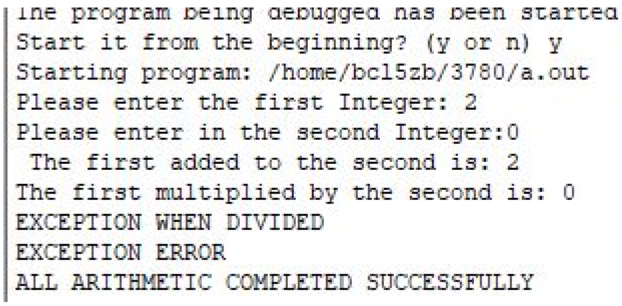
Code:

```
[bcl5zb@delmar 3780]$ ls
a.out  task1.cpp  task2.cpp  task3.cpp
[bcl5zb@delmar 3780]$ vi task2.cpp
#include <iostream>

int main (){

        int array[5];
        int *darray = new int[5];

        return 0;
}
```

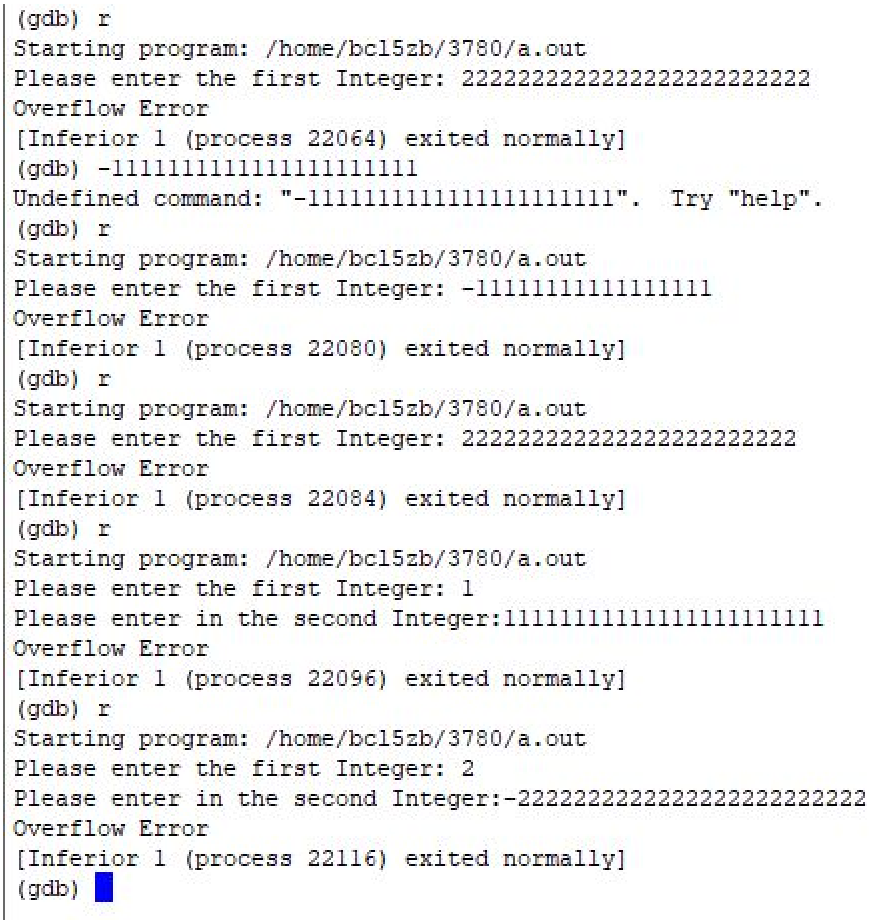
Task 3:
Write a program that prompts the user for two numbers and stores them in signed integers. The program should then add those two numbers together and store the result in a signed integer and display the result. Your program should then multiply them by each other and store the result in another b integer and display the result. Then do the same but with dividing the first number by the second. Display an error message to the screen if an operation has happened that does not result in a correct calculation. In other words, make sure to test your code for error cases. You can safely assume I will only give your program integers (I will give your program only decimal digits).



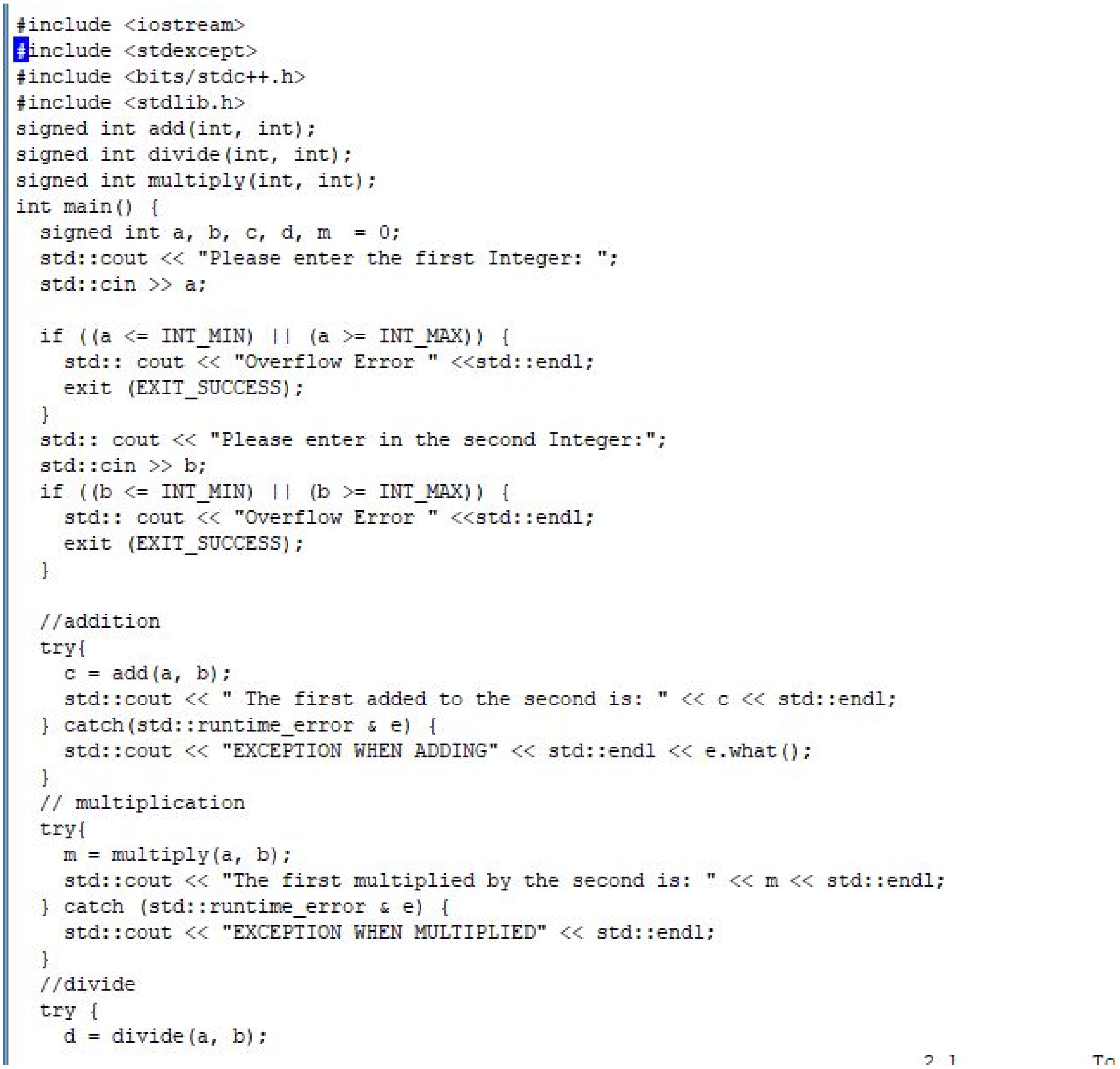
```
Starting program: /home/bcl5zb/3780/a.out
Please enter the first Integer: -2000000
Please enter in the second Integer:20000000
 The first added to the second is: 18000000
EXCEPTION WHEN MULTIPLIED
The first divided by the second is: 0
ALL ARITHMETIC COMPLETED SUCCESSFULLY
[Inferior 1 (process 21227) exited normally]
(gdb)
```



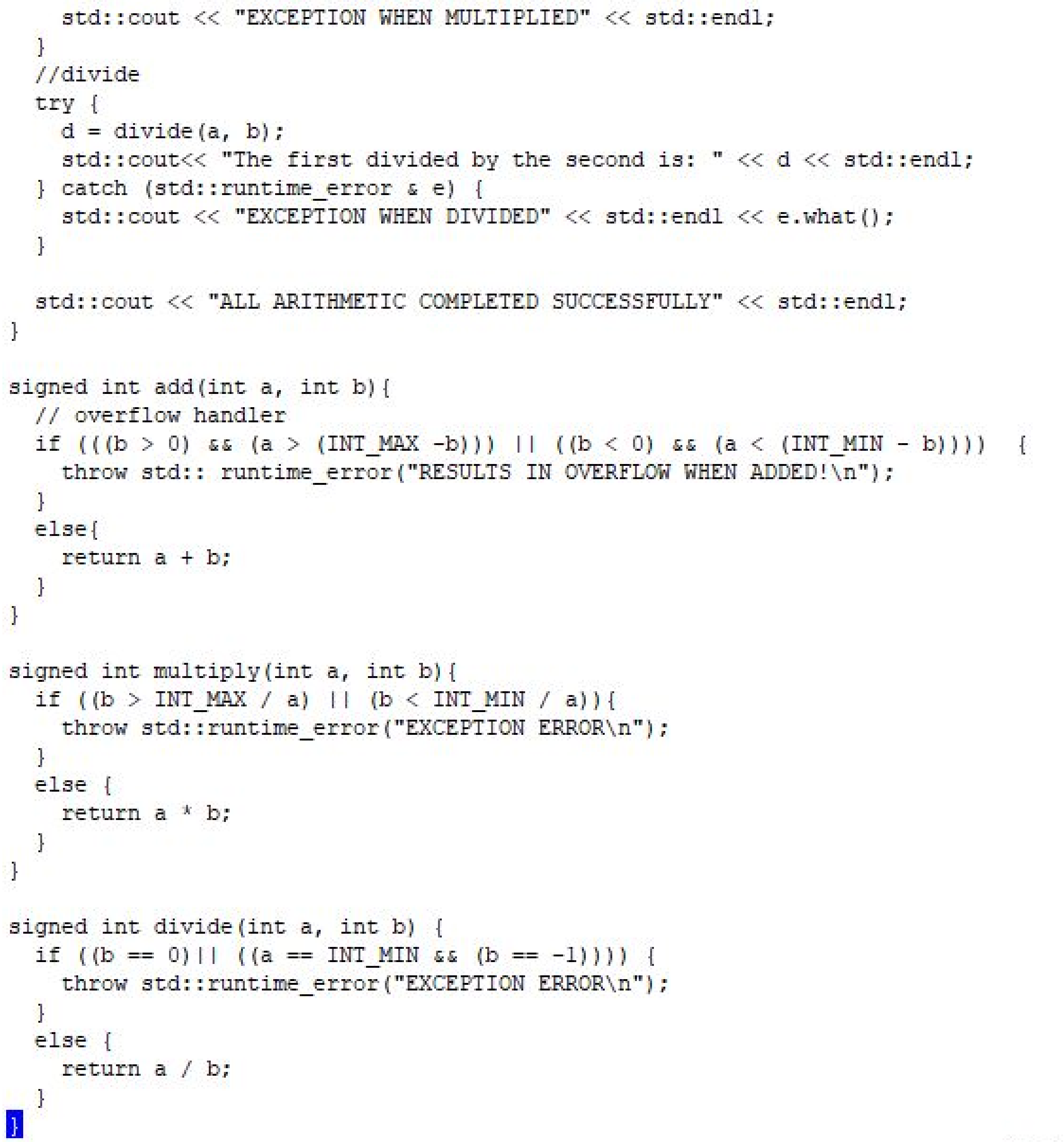
```
The program being debugged has been started
Start it from the beginning? (y or n) y
Starting program: /home/bcl5zb/3780/a.out
Please enter the first Integer: 2
Please enter in the second Integer:0
 The first added to the second is: 2
The first multiplied by the second is: 0
EXCEPTION WHEN DIVIDED
EXCEPTION ERROR
ALL ARITHMETIC COMPLETED SUCCESSFULLY
```



```
(gdb) r
Starting program: /home/bcl5zb/3780/a.out
Please enter the first Integer: 2222222222222222222222222
Overflow Error
[Inferior 1 (process 22064) exited normally]
(gdb) -1111111111111111111111
Undefined command: "-1111111111111111111111".  Try "help".
(gdb) r
Starting program: /home/bcl5zb/3780/a.out
Please enter the first Integer: -11111111111111111
Overflow Error
[Inferior 1 (process 22080) exited normally]
(gdb) r
Starting program: /home/bcl5zb/3780/a.out
Please enter the first Integer: 222222222222222222222222
Overflow Error
[Inferior 1 (process 22084) exited normally]
(gdb) r
Starting program: /home/bcl5zb/3780/a.out
Please enter the first Integer: 1
Please enter in the second Integer:11111111111111111111111
Overflow Error
[Inferior 1 (process 22096) exited normally]
(gdb) r
Starting program: /home/bcl5zb/3780/a.out
Please enter the first Integer: 2
Please enter in the second Integer:-2222222222222222222222222
Overflow Error
[Inferior 1 (process 22116) exited normally]
(gdb) 
```



```
#include <iostream>
#include <stdexcept>
#include <bits/stdc++.h>
#include <stdlib.h>
signed int add(int, int);
signed int divide(int, int);
signed int multiply(int, int);
int main() {
  signed int a, b, c, d, m  = 0;
  std::cout << "Please enter the first Integer: ";
  std::cin >> a;

  if ((a <= INT_MIN) || (a >= INT_MAX)) {
    std:: cout << "Overflow Error " <<std::endl;
    exit (EXIT_SUCCESS);
  }
  std:: cout << "Please enter in the second Integer:";
  std::cin >> b;
  if ((b <= INT_MIN) || (b >= INT_MAX)) {
    std:: cout << "Overflow Error " <<std::endl;
    exit (EXIT_SUCCESS);
  }

  //addition
  try{
    c = add(a, b);
    std::cout << " The first added to the second is: " << c << std::endl;
  } catch(std::runtime_error & e) {
    std::cout << "EXCEPTION WHEN ADDING" << std::endl << e.what();
  }
  // multiplication
  try{
    m = multiply(a, b);
    std::cout << "The first multiplied by the second is: " << m << std::endl;
  } catch (std::runtime_error & e) {
    std::cout << "EXCEPTION WHEN MULTIPLIED" << std::endl;
  }
  //divide
  try {
    d = divide(a, b);
                                                                    2,1           To
```



```
    std::cout << "EXCEPTION WHEN MULTIPLIED" << std::endl;
  }
  //divide
  try {
    d = divide(a, b);
    std::cout<< "The first divided by the second is: " << d << std::endl;
  } catch (std::runtime_error & e) {
    std::cout << "EXCEPTION WHEN DIVIDED" << std::endl << e.what();
  }

  std::cout << "ALL ARITHMETIC COMPLETED SUCCESSFULLY" << std::endl;
}

signed int add(int a, int b){
  // overflow handler
  if (((b > 0) && (a > (INT_MAX -b))) || ((b < 0) && (a < (INT_MIN - b))))  {
    throw std:: runtime_error("RESULTS IN OVERFLOW WHEN ADDED!\n");
  }
  else{
    return a + b;
  }
}

signed int multiply(int a, int b){
  if ((b > INT_MAX / a) || (b < INT_MIN / a)){
    throw std::runtime_error("EXCEPTION ERROR\n");
  }
  else {
    return a * b;
  }
}

signed int divide(int a, int b) {
  if ((b == 0)|| ((a == INT_MIN && (b == -1)))) {
    throw std::runtime_error("EXCEPTION ERROR\n");
  }
  else {
    return a / b;
  }
}
```

```
[bcl5zb@delmar 3780]$ vi task3.cpp
#include <iostream>
#include <stdexcept>
#include <bits/stdc++.h>
#include <stdlib.h>
signed int add(int, int);
signed int divide(int, int);
signed int multiply(int, int);
int main() {
  signed int a, b, c, d, m  = 0;
  std::cout << "Please enter the first Integer: ";
  std::cin >> a;
  if ((a <= INT_MIN) || (a >= INT_MAX)) {
    std:: cout << "Overflow Error " <<std::endl;
    exit (EXIT_SUCCESS);
  }
  std:: cout << "Please enter in the second Integer:";
  std::cin >> b;
  if ((b <= INT_MIN) || (b >= INT_MAX)) {
    std:: cout << "Overflow Error " <<std::endl;
    exit (EXIT_SUCCESS);
  }

  //addition
  try{
    c = add(a, b);
    std::cout << " The first added to the second is: " << c << std::endl;
  } catch(std::runtime_error & e) {
    std::cout << "EXCEPTION WHEN ADDING" << std::endl << e.what();
  }
  // multiplication
  try{
    m = multiply(a, b);
    std::cout << "The first multiplied by the second is: " << m << std::endl;
  } catch (std::runtime_error & e) {
    std::cout << "EXCEPTION WHEN MULTIPLIED" << std::endl;
  }
  //divide
  try {
    d = divide(a, b);
    std::cout<< "The first divided by the second is: " << d << std::endl;
  } catch (std::runtime_error & e) {
    std::cout << "EXCEPTION WHEN DIVIDED" << std::endl << e.what();
  }

  std::cout << "ALL ARITHMETIC COMPLETED SUCCESSFULLY" << std::endl;
}

signed int add(int a, int b){
  // overflow handler
  if (((b > 0) && (a > (INT_MAX -b))) || ((b < 0) && (a < (INT_MIN - b))))  {
    throw std:: runtime_error("RESULTS IN OVERFLOW WHEN ADDED!\n");
  }
  else{
    return a + b;
  }
}

signed int multiply(int a, int b){
  if ((b > INT_MAX / a) || (b < INT_MIN / a)){
    throw std::runtime_error("EXCEPTION ERROR\n");
  }
  else {
    return a * b;
  }
}

signed int divide(int a, int b) {
  if ((b == 0)|| ((a == INT_MIN && (b == -1)))) {
    throw std::runtime_error("EXCEPTION ERROR\n");
  }
  else {
    return a / b;
  }
}
```