Specification

```
Lab10B Specification:
The purpose of this lab is to continue explorations of
communications with web server and with other programs we may
have access to on our systems.
```

HTML Page

```
<html>
        <head>
                <H1>Lab10C - Slope</H1>
        </head>
        <body>
                <form action="/cgi-bin/Lab10.cgi
                    ">
                        <label>Point 1:
                                <input name="X1"
                                    size="20">
                                <input name="Y1"
                                    size="20">
                        </label>
                        <br>
                        <label>Point 2:
                                <input name="X2"
                                    size="20">
                                <input name="Y2"
                                    size="20">
                        </label>
                        <input type="submit">
                </form>
        </body>
</html>
```

Start

```
        .data
        .global _argc_
        .global _argv_
        .global _envp_

_argc_: .long 0
_argv_: .quad 0
_envp_: .quad 0

        .text
        .global _start
```

```
_start:
        movl (%rsp), %edi
        lea 8(%rsp), %rsi
        lea 16(%rsp, %rdi, 8), %rdx
        movl %edi, _argc_
        movq %rsi, _argv_
        movq %rdx, _envp_
        call main
        movq %rax, %rdi
        movq $60, %rax
        syscall
```

The purpose of this implementation of _start is to grab argc, argv, and envp off of the stack, place them in global variables, and pass them to main as arguments. After main has exited, _start will call sys_exit.

Main

```
        .section .rodata

QUERY_STRING:
        .string "QUERY_STRING"

X1:
        .string "X1"
Y1:
        .string "Y1"
X2:
        .string "X2"
Y2:
        .string "Y2"
SLOPE:
        .string "Slope: "

        .text
        .global main
```

The read-only data section of the main file consists of a few variables.
QUERY_STRING is just a hardcoded string to search for in the environment variables.
X1 is a hardcoded string to search for within the query string, as well as Y1, X2, and Y2.
SLOPE is a hardcoded message.

```
.equ QueryString, -8
.equ X1F, -12
.equ Y1F, -16
.equ X2F, -20
.equ Y2F, -24
.equ Buffer, -88
.equ EquationBuffer, -152
.equ Slope, -156
```

The main function uses multiple local variables.
QueryString holds a pointer to the QUERY_STRING.
X1F will contain the float value of the X1 argument.
Y1F will contain the float value of the Y1 argument.
X2F will contain the float value of the X2 argument.
Y2F will contain the float value of the Y2 argument.
Buffer is a general usage string buffer.
EquationBuffer is a buffer that will hold the generated line equation.
Slope will hold the float value of the slope.

```
main:
        enter $256, $0
        movq %rdx, %rsi
        lea QUERY_STRING, %rdi
        call GetENV
        movq %rax, QueryString(%rbp)
```

First, the function enters, and creates the stack frame.
It also calls GetENV and stores the QUERY_STRING pointer on the stack.

Address	Name	Type	Value
RBP-8	QueryString	char*	"QUERY_STRI..."
RBP-12	X1F	float	?
RBP-16	Y1F	float	?
RBP-20	X2F	float	?
RBP-24	Y2F	float	?
RBP-88	Buffer	char[64]	?
RBP-152	Equation	char[64]	?
RBP-156	Slope	float	?

```
movq QueryString(%rbp), %rsi
lea X1, %rdi
lea Buffer(%rbp), %rdx
call GetQueryStringValue
lea Buffer(%rbp), %rdi
call StringToFloat
movss %xmm0, X1F(%rbp)
```

Now, the function will translate the first float argument into a float value.
It uses GetQueryStringValue to copy the string into the general purpose buffer, then it uses StringToFloat to convert it to a float.
After that, it copies the float onto the stack.

Address	Name	Type	Value
RBP-8	QueryString	char*	"QUERY_STRI..."
RBP-12	X1F	float	1
RBP-16	Y1F	float	?
RBP-20	X2F	float	?
RBP-24	Y2F	float	?
RBP-88	Buffer	char[64]	"1.0"
RBP-152	Equation	char[64]	?
RBP-156	Slope	float	?

```
movq QueryString(%rbp), %rsi
lea Y1, %rdi
lea Buffer(%rbp), %rdx
call GetQueryStringValue
lea Buffer(%rbp), %rdi
call StringToFloat
movss %xmm0, Y1F(%rbp)
```

Now, the function will translate the first float argument into a float value.
It uses GetQueryStringValue to copy the string into the general purpose buffer, then it uses StringToFloat to convert it to a float.
After that, it copies the float onto the stack.

Address	Name	Type	Value
RBP-8	QueryString	char*	"QUERY_STRI..."
RBP-12	X1F	float	1
RBP-16	Y1F	float	2
RBP-20	X2F	float	?
RBP-24	Y2F	float	?
RBP-88	Buffer	char[64]	"2.0"
RBP-152	Equation	char[64]	?
RBP-156	Slope	float	?

```
movq QueryString(%rbp), %rsi
lea X2, %rdi
lea Buffer(%rbp), %rdx
call GetQueryStringValue
lea Buffer(%rbp), %rdi
call StringToFloat
movss %xmm0, X2F(%rbp)
```

Now, the function will translate the first float argument into a float value.
It uses GetQueryStringValue to copy the string into the general purpose buffer, then it uses StringToFloat to convert it to a float.
After that, it copies the float onto the stack.

Address	Name	Type	Value
RBP-8	QueryString	char*	"QUERY_STRI..."
RBP-12	X1F	float	1
RBP-16	Y1F	float	2
RBP-20	X2F	float	3
RBP-24	Y2F	float	?
RBP-88	Buffer	char[64]	"3.0"
RBP-152	Equation	char[64]	?
RBP-156	Slope	float	?

```
movq QueryString(%rbp), %rsi
lea Y2, %rdi
lea Buffer(%rbp), %rdx
call GetQueryStringValue
lea Buffer(%rbp), %rdi
call StringToFloat
movss %xmm0, Y2F(%rbp)
```

Now, the function will translate the first float argument into a float value.
It uses GetQueryStringValue to copy the string into the general purpose buffer, then it uses StringToFloat to convert it to a float.
After that, it copies the float onto the stack.

Address	Name	Type	Value
RBP-8	QueryString	char*	"QUERY_STRI..."
RBP-12	X1F	float	1
RBP-16	Y1F	float	2
RBP-20	X2F	float	3
RBP-24	Y2F	float	4
RBP-88	Buffer	char[64]	"4.0"
RBP-152	Equation	char[64]	?
RBP-156	Slope	float	?

```
movss Y2F(%rbp), %xmm0
movss Y1F(%rbp), %xmm1
subss %xmm1, %xmm0
movss X2F(%rbp), %xmm2
movss X1F(%rbp), %xmm3
subss %xmm3, %xmm2
divss %xmm2, %xmm0
movss %xmm0, Slope(%rbp)
lea Buffer(%rbp), %rdi
call FloatToString
```

Now, the function will take the converted float values, and use them to calculate the slope of the line between them.
It subtracts them from each other, and calculates rise over run.
Then, it uses the FloatToString function to convert it back into a string.
It also stores the converted float value on the stack in the Slope variable.

Address	Name	Type	Value
RBP-8	QueryString	char*	"QUERY_STRI..."
RBP-12	X1F	float	1
RBP-16	Y1F	float	2
RBP-20	X2F	float	3
RBP-24	Y2F	float	4
RBP-88	Buffer	char[64]	"1.0"
RBP-152	Equation	char[64]	?
RBP-156	Slope	float	1

```
call PrintHTMLHeader
lea SLOPE, %rdi
call Print
lea Buffer(%rbp), %rdi
call PrintLine
call PrintHTMLBreak
```

Next, the function will print the HTML header, as well as the slope.
It will then print a line break to make the graph show up on a new line.

```
lea EquationBuffer(%rbp), %rdi
movss X1F(%rbp), %xmm0
movss Y1F(%rbp), %xmm1
movss Slope(%rbp), %xmm2
call Equation
lea EquationBuffer(%rbp), %rdi
call Plot
```

Next, the function will generate the equation to graph.
It loads the values of X1F, Y1F, and Slope into floating point argument registers, and a pointer to the Equation buffer in rdi.
Then, it calls the Equation function to generate the equation, and subsequently calls Plot, to create the graph of the line.

Address	Name	Type	Value
RBP-8	QueryString	char*	"QUERY_STRI..."
RBP-12	X1F	float	1
RBP-16	Y1F	float	2
RBP-20	X2F	float	3
RBP-24	Y2F	float	4
RBP-88	Buffer	char[64]	"1.0"
RBP-152	Equation	char[64]	"1*(x-1)-2"
RBP-156	Slope	float	1

```
lea PLOT_OUTPUT_FILE, %rdi
call PrintHTMLImage
xorq %rax, %rax
leave
ret
```

Finally, the function displays an HTML image tag containing the path to the image generated.
Then, the function returns zero.

StringToFloat

```
        .section .rodata

Radix:
        .float 10.0
_1:
        .float 1.0
_0:
        .float 0.0

        .text
        .global StringToFloat
```

In the stof file, the read-only data section contains a few entries.
Radix is just a constant, a float that equals 10, used for conversion.
_1 is a constant, equaling 1, for utility purposes.
_0 is a constant, equaling 0, for utility purposes.

```
StringToFloat:
        xorq %rcx, %rcx
stof_For_1:
        movb (%rdi, %rcx, 1), %al
        cmp $'.', %al
        je stof_For_2
        test %al, %al
        jz stof_For_2
        incq %rcx
        jmp stof_For_1
stof_For_2:
```

When StringToFloat first enters, it sets rcx to zero, because it will be used as a for loop iterator.
Then, the function gets the index of the first decimal point in the string, and if there is none, the index is equal to the null terminator.

```
        movss _1, %xmm1
stof_For_3:
        decq %rcx
        jz stof_For_4
        mulss Radix, %xmm1
        jmp stof_For_3
stof_For_4:
```

Then, the function sets xmm1 to 1, and multiplies it by the Radix based on how many characters were present before the index of the decimal point.

```
        xorq %rcx, %rcx
        xorps %xmm0, %xmm0
stof_For_5:
        movzbq (%rdi, %rcx, 1), %rax
        cmp $'.', %rax
        je stof_Skip
        test %rax, %rax
        jz stof_For_6
        subq $'0', %rax
        cvtsi2ss %rax, %xmm2
        mulss %xmm1, %xmm2
        addss %xmm2, %xmm0
        divss Radix, %xmm1
stof_Skip:
        incq %rcx
        jmp stof_For_5
stof_For_6:
```

The function then sets rcx back to zero, because it will be used as a loop iterator. It also sets xmm0 to zero because it will be used to accumulate the value of the string. The function then loops through all numeric characters in the string, skipping decimal points, and dividing the base by 10 each time.
Each character is translated into a float and multiplied by the current base, and added onto the accumulator.

Then, the function returns.

```
ret
```

FloatToString

```
        .section .rodata

RADIXD:
        .double 10.0
RADIXF:
        .float 10.0
NEGATIVED:
        .double -1.0
NEGATIVEF:
        .float -1.0
ZEROF:
        .float 0.0
```

In the read-only data section of the ftos file, there are a few important constants to be noted.
RADIXD is a double equaling 10, used in conversion.
RADIXF is a float equaling 10, used in conversion.
NEGATIVED is a double equaling -1, used for flipping the sign of doubles.
NEGATIVEF is a float equaling -1, used for flipping the sign of floats.
ZEROF is a float equaling zero, used for checking if other floats are zero.

```
.text
.global FloatToString

.equ Precision, 6
.equ Radix, 10
.equ Buffer, -64
```

The FloatToString function uses one local variable on its stack frame, Buffer. Buffer is a string buffer, that is used during the function to contain an intermediate form of the final output, before it is copied over and a null-terminator and decimal point are added. Precision is a constant, defining how many decimal places the string should preserve. Radix is a constant, because float strings are generally in base 10.

```
FloatToString:
        comiss ZEROF, %xmm0
        jne ftos_Start
        movb $'0', (%rdi)
        movb $0, 1(%rdi)
        ret
```

The objective of the FloatToString function is to convert a 32 bit float to a string. First, it checks if the given float is zero, and if so, it writes a zero into the output buffer and returns.

```
ftos_Start:
        enter $64, $0
        xorq %rax, %rax
        cvtss2sd %xmm0, %xmm2
        movsd RADIXD, %xmm3
ftos_While_1:
        cmp $Precision, %rax
        je ftos_While_2
        mulsd %xmm3, %xmm2
        incq %rax
        jmp ftos_While_1
ftos_While_2:
```

If the given float was not zero, the function will begin translating the float to a string. First, the function converts the float to a double, to increase the range of numbers that can be represented.
Second, the function loops through and multiplies the given double by Radix, however many times the Precision constant requires.

```
        cvtsd2si %xmm2, %rax
        xorq %rcx, %rcx
        xorq %r8, %r8
        cmp $0, %rax
        jnl ftos_If_1
        movb $1, %r8b
        negq %rax
ftos_If_1:
```

After the function has finished multiplying by Radix, it converts the freshly multiplied double into an integer, to ease conversion. It then checks if the integer is negative, and if it is, it sets a flag and negates the value.

```
ftos_While_3:
        test %rax, %rax
        jz ftos_While_4
        movq $Radix, %r9
        xorq %rdx, %rdx
        divq %r9
        addb $'0', %dl
        movb %dl, Buffer(%rbp, %rcx, 1)
        incq %rcx
        jmp ftos_While_3
ftos_While_4:
```

Subsequently, the function loops through the integer, converting it into its string representation
and storing it in the intermediate output buffer on the stack.

```
        test %r8, %r8
        jz ftos_If_2
        movb $'-', Buffer(%rbp, %rcx, 1)
        incq %rcx
ftos_If_2:
        xorq %r8, %r8
        movq %rcx, %r9
        decq %r9
```

After conversion, the function checks if the afformentioned negative flag was set, and if so, it appends a negative sign. Subsequently, it sets r8 to zero, because it will be used as an index for the final output buffer, and r9 to rcx, which is the number of characters in the intermediate buffer. It will be used to count down through the reversed intermediate buffer.

```
ftos_While_5:
        cmp $0, %r9
        jl ftos_While_6
        movq %rcx, %r11
        movq $Precision, %rsi
        subq %rsi, %r11
        cmp %r11, %r8
        jne ftos_If_3
        movb $'.', (%rdi, %r8, 1)
        incq %r8
ftos_If_3:
        movb Buffer(%rbp, %r9, 1), %r10b
        movb %r10b, (%rdi, %r8, 1)
        incq %r8
        decq %r9
        jmp ftos_While_5
ftos_While_6:
```

The function now has to copy the reversed characters from the intermediate buffer into the final buffer, and add a decimal point and null-terminator.
All this loop does is go through the intermediate buffer backwards, copying characters, and if the required position of a decimal place is met, a decimal point is also appended.

Then, the function returns.

```
movb $0, (%rdi, %r8, 1)
leave
ret
```

Equation

```
        .section .rodata

EQ1:
        .string "*(x-"
EQ2:
        .string ")+"

        .text
        .global Equation
```

In the Equation file, there are some important read-only data variables.
EQ1 is the first hardcoded segment of the equation format,
and EQ2 is the second hardcoded segment of the equation format.
The afformentioned format is a line equation format called point-slope.
It allows you to create a line equation from just a point and the slope, which is perfect for our needs.

```
Concatenate:
        xorq %rax, %rax
Concatenate_While_1:
        movb (%rsi, %rax, 1), %cl
        test %cl, %cl
        jz Concatenate_While_2
        incq %rax
        jmp Concatenate_While_1
Concatenate_While_2:
```

The Concatenate function is merely a duplication of strcat, which is used by the Equation function to append strings together. First, it loops through the destination string, and finds the null-terminator.

```
        xorq %rcx, %rcx
Concatenate_While_3:
        movb (%rdi, %rcx, 1), %dl
        test %dl, %dl
        jz Concatenate_While_4
        movb %dl, (%rsi, %rax, 1)
        incq %rax
        incq %rcx
        jmp Concatenate_While_3
Concatenate_While_4:
        movb $0, (%rsi, %rax, 1)
        ret
```

Then, it loops through the source string, copying it over into the destination string until a null-terminator is reached.

```
.equ X, -4
.equ Y, -8
.equ M, -12
.equ XString, -44
.equ YString, -76
.equ MString, -108
.equ Output, -116
```

The Equation function uses several local variables.
X is a storage location for the float value of X.
Y is a storage location for the float value of Y.
M is a storage location for the float value of the Slope.
XString is a string buffer, used to store the string representation of the float X.
YString is a string buffer, used to store the string representation of the float Y.
MString is a string buffer, used to store the string representation of the float M.
Output is a storage location for the pointer value of the output buffer.

```
Equation:
        enter $128, $0
        movq %rdi, Output(%rbp)
        movb $0, (%rdi)
        movss %xmm0, X(%rbp)
        movss %xmm1, Y(%rbp)
        movss %xmm2, M(%rbp)
```

First, the function stores the arguments into local variables.

```
lea XString(%rbp), %rdi
call FloatToString
movss Y(%rbp), %xmm0
lea YString(%rbp), %rdi
call FloatToString
movss M(%rbp), %xmm0
lea MString(%rbp), %rdi
call FloatToString
```

Second, the function converts the three floating arguments into strings, and stores them into their respective string buffers.

```
movq Output(%rbp), %rsi
lea MString(%rbp), %rdi
call Concatenate
movq Output(%rbp), %rsi
lea EQ1, %rdi
call Concatenate
movq Output(%rbp), %rsi
lea XString(%rbp), %rdi
call Concatenate
movq Output(%rbp), %rsi
lea EQ2, %rdi
call Concatenate
movq Output(%rbp), %rsi
lea YString(%rbp), %rdi
call Concatenate
leave
ret
```

The string then concatenates the strings, along with a few hardcoded strings, to make a point-slope equation that gnuplot will accept.

GetENV

```
.text
.global GetENV
.global GetENVValue
```

```
GetENV:
        xorq %rcx, %rcx
```

The GetENV function is meant to parse through a list of environment pointers, and return a pointer to the one bearing the specified key. Our use case would be only for QUERY_STRING, but this function can be used to retrieve any environment pointer.
First, the function sets rcx to zero, because it will later be used as an iterator variable for a loop.

```
GetENV_While_1:
        movq (%rsi, %rcx, 8), %rax
        test %rax, %rax
        jz GetENV_Fail
        xorq %rdx, %rdx
```

This is the beginning of the outer loop in the GetENV function. It basically iterates through every single envp entry in the envp array until it reaches a null pointer.

```
GetENV_For_1:
        movb (%rdi, %rdx, 1), %r8b
        movb (%rax, %rdx, 1), %r9b
        test %r8b, %r8b
        jnz GetENV_No_Success
        cmp $'=', %r9b
        jne GetENV_No_Success
        jmp GetENV_Success
GetENV_No_Success:
        test %r9b, %r9b
        jz GetENV_For_2
        cmp %r8b, %r9b
        jne GetENV_For_2
        incq %rdx
        jmp GetENV_For_1
```

This is the inner loop, its function is to take the current envp that the outer loop has provided it, and perform a simple string matching operation to determine whether or not the key is the one we are searching for. It just goes through the environment variable until it either finds a non-matching character or a null pointer, and if it hits an equals sign before that, it will indicate success by returning a pointer to the environment variable.

```
GetENV_For_2:
        incq %rcx
        jmp GetENV_While_1
```

This is the code that is executed whenever the inner loop finishes execution without finding a confirmed match.
All it does is increment the outer loop iterator variable rcx, and jump back to the start of the outer loop.

```
GetENV_Fail:
        xorq %rax, %rax
GetENV_Success:
        ret
```

These are the labels that are jumped to to indicate either success or failiure.
If the failiure label is jumped to, rax is set to zero, and the function returns a null pointer.
If the success label is jumped to, the function just returns, because the current env variable is already in rax.

```
GetENVValue:
        call GetENV
GetENVValue_While_1:
        movb (%rax), %cl
        cmp $'=', %cl
        je GetENVValue_Success
        incq %rax
        jmp GetENVValue_While_1
GetENVValue_Success:
        incq %rax
        ret
```

The GetENVValue function is essentially just a wrapper around the GetENV function.
All it does is call GetENV to get the start of the matchig environment pointer, and subsequently increments the pointer until the first equals sign in the environment string has been passed.

Query

```
.text
.global GetQueryString
.global GetQueryStringValueAddress
.global GetQueryStringValue
```

```
GetQueryString:
GetQueryString_While_1:
        movb (%rsi), %al
        cmp $0, %al
        jne GetQueryString_If_1
        movq $0, %rax
        ret
GetQueryString_If_1:
```

The GetQueryString function is meant to parse the QUERY_STRING environment variable for a specified variable.
This is essentially the same as a strstr function.
First the function enters an outer while loop, that will iterate through each character.
The loop first checks if the current character is equal to a null terminator, and if it is, it will return a null pointer.

```
        xorq %r8, %r8
GetQueryString_For_1:
        movb (%rdi, %r8, 1), %al
        cmp $0, %al
        jne GetQueryString_If_2
        movq %rsi, %rax
        ret
GetQueryString_If_2:
        movb (%rsi, %r8, 1), %al
        cmp $0, %al
        jne GetQueryString_If_3
        xorq %rax, %rax
        ret
GetQueryString_If_3:
        movb (%rdi, %r8, 1), %al
        movb (%rsi, %r8, 1), %cl
        cmp %al, %cl
        jne GetQueryString_For_2
        incq %r8
        jmp GetQueryString_For_1
GetQueryString_For_2:
```

Here, the function is entering its inner loop, the function of which is to match the key we are looking for to the current string. r8 is set to zero, because it will be used as the iterator for the inner loop.
The inner loop iterates through the string until it finds either a null character or a non matching character.
If the end of the string is reached before an unmatching character is found, the function will return the pointer to the specified variable within QUERY_STRING.

```
incq %rsi
jmp GetQueryString_While_1
```

This code merely increments the string pointer for the outer loop, and jumps back to the beginning of the outer loop.

```
GetQueryStringValueAddress:
        call GetQueryString
GetQueryStringValueAddress_While_1:
        movb (%rax), %cl
        cmp $'=', %cl
        je GetQueryStringValueAddress_While_2
        incq %rax
        jmp GetQueryStringValueAddress_While_1
GetQueryStringValueAddress_While_2:
        incq %rax
        ret
```

The purpose of the GetQueryStringValueAddress function is to call GetQueryString, and increment the returned pointer until the first equals sign in the string has been passed. It's meant to help isolate the variable from the key.

```
GetQueryStringValue:
        push %rdx
        call GetQueryStringValueAddress
        pop %rdx
        movq %rax, %rdi
        movq %rdx, %rsi
        call QueryTranslate
        ret
```

The objective of the GetQueryStringValue function is to call GetQueryStringValueAddress, take the returned pointer, and copy every subsequent character in the string until it reaches either an ampersand or a null chaacter.

The objective of the QueryHex function is to translate HTML hex codes into characters.

```
QueryHex:
        cmp $'0', %dil
        jl QueryHex_Else_1
        cmp $'9', %dil
        jg QueryHex_Else_1
        subb $'0', %dil
        movb %dil, %al
        ret
QueryHex_Else_1:
        andb $0b11011111, %dil
        cmp $'A', %dil
        jl QueryHex_Else_2
        cmp $'F', %dil
        jg QueryHex_Else_2
        subb $'A', %dil
        addb $10, %dil
        movb %dil, %al
        ret
QueryHex_Else_2:
        movb $-1, %al
        ret
```

The objective of the QueryTranslate function is to normalize HTML strings.
If a string contains html hex codes for special characters, or it contains plus signs in place of spaces,
then this function will translate the string into a format that gnuplot will accept.

```
        .equ QueryTranslate_Input_Index , -8
        .equ QueryTranslate_Output_Index , -16
        .equ QueryTranslate_Input , -24
        .equ QueryTranslate_Output , -32

QueryTranslate:
        enter $32 , $0
        push %r12
        movq %rdi , QueryTranslate_Input(%rbp)
        movq %rsi , QueryTranslate_Output(%rbp)
        movq $0 , QueryTranslate_Input_Index(%rbp
            )
        movq $0 , QueryTranslate_Output_Index(
            %rbp)
```

```
QueryTranslate_While_1:
        movq QueryTranslate_Input(%rbp), %rax
        movq QueryTranslate_Input_Index(%rbp),
            %rcx
        movb (%rax, %rcx, 1), %al
        test %al, %al
        jz QueryTranslate_While_2
        cmp $'&', %al
        je QueryTranslate_While_2
        cmp $'%', %al
        je QueryTranslate_Switch_Case_Percent
        cmp $'+', %al
        je QueryTranslate_Switch_Case_Plus
        jmp QueryTranslate_Switch_Default
```

After initializing its local variables, QueryTranslate begins its first while loop, the purpose of which is to iterate through all the characters in the input string.
It will only stop iterating if it reaches either a null terminator or an ampersand.
Inside of the loop, it goes through a switch statement that checks for percent signs and plus signs.
Percent signs denote the presence of a literal hex character in the following two bytes.
Plus signs, in html, are replacements for spaces.

```
QueryTranslate_Switch_Case_Percent:
        xorb %r12b, %r12b
        incq QueryTranslate_Input_Index(%rbp)
```

If the character was a percent sign, the function will set r12b to zero, because it will be used to accumulate the character onto.
It will also increment the input index to bypass the percent sign.

```
QueryTranslate_For_1:
        movq QueryTranslate_Input(%rbp), %rcx
        movq QueryTranslate_Input_Index(%rbp),
            %rdx
        movb (%rcx, %rdx, 1), %dil
        call QueryHex
        cmp $-1, %al
        jz QueryTranslate_For_2
        movb %al, %r8b
        movb %r12b, %al
        movb $16, %cl
        imulb %cl
        movb %al, %r12b
        addb %r8b, %r12b
        incq QueryTranslate_Input_Index(%rbp)
        jmp QueryTranslate_For_1
QueryTranslate_For_2:
```

```
This is the for loop through which the
function iterates until it finds a non-hex
character.
For every character, it calls QueryHex, which
will check if the character is a valid hex
code.
If so, it will accumulate it onto r12b, in
order to translate the hex code into an
actual character.
```

```
movq QueryTranslate_Output(%rbp), %rcx
movq QueryTranslate_Output_Index(%rbp),
    %rdx
movb %r12b, (%rcx, %rdx, 1)
incq QueryTranslate_Output_Index(%rbp)
jmp QueryTranslate_Switch_End
```

Once the translation loop has exited, the function writes the character onto the output string, and jumps to the end of the switch statement.

```
QueryTranslate_Switch_Case_Plus:
        movq QueryTranslate_Output(%rbp), %rcx
        movq QueryTranslate_Output_Index(%rbp),
            %rdx
        movb $' ', (%rcx, %rdx, 1)
        incq QueryTranslate_Input_Index(%rbp)
        incq QueryTranslate_Output_Index(%rbp)
        jmp QueryTranslate_Switch_End
```

If the character was a plus, the function substitutes a space for the character in the output string, and jumps to the end of the switch statement.

If the character has no special meaning, just write it into the output string with no changes.

```
QueryTranslate_Switch_Default:
        movq QueryTranslate_Output(%rbp), %rcx
        movq QueryTranslate_Output_Index(%rbp),
            %rdx
        movb %al, (%rcx, %rdx, 1)
        incq QueryTranslate_Input_Index(%rbp)
        incq QueryTranslate_Output_Index(%rbp)
```

```
QueryTranslate_Switch_End:
        jmp QueryTranslate_While_1
```

At the end of the switch statement, all that takes place is a jump back to the beginning of the first while loop.

When the while loop ends, the function writes a null terminator to the end of the output string, collapses its stack frame, and returns.

```
QueryTranslate_While_2:
        movq QueryTranslate_Output(%rbp), %rcx
        movq QueryTranslate_Output_Index(%rbp),
            %rdx
        movb $0, (%rcx, %rdx, 1)
        pop %r12
        leave
        ret
```

HTMLHeader

```
        .section .rodata

HTMLHeader: .string "Content-type: text/html\n\n
   "

        .text
        .global PrintHTMLHeader
```

```
PrintHTMLHeader:
        lea HTMLHeader, %rdi
        call PrintLine
        ret
```

The purpose of the PrintHTMLHeader function is to print a hardcoded HTML header.
It is used because CGI applications are meant to disclose what type of file they are trying to produce, in our case, HTML.

PrintHTMLImage

```
        .section .rodata

TAG_1:
        .string "<img src=\""
TAG_2:
        .string "\">"

        .text
        .global PrintHTMLImage
```

```
PrintHTMLImage:
        push %rdi
        lea TAG_1, %rdi
        call Print
        pop %rdi
        call Print
        lea TAG_2, %rdi
        call Print
        call NewLine
        ret
```

The objective of the PrintHTMLImage function is to provide a simple way to display an image.
All it does is wrap the given string, passed in rdi, inside of a html image tag.

PrintHTMLBreak

The objective of the PrintHTMLBreak function is to print an html break tag.

```
        .section .rodata

BREAK:
        .string "<br>"

        .text
        .global PrintHTMLBreak

PrintHTMLBreak:
        lea BREAK, %rdi
        call PrintLine
        ret
```

Process

```
.text
.global Fork
.global Execute
.global Wait
.global Spawn

.equ SYS_FORK, 57
.equ SYS_EXECVE, 59
.equ SYS_WAIT4, 61

.equ WAIT_STAT_LOC, -4
.equ WAIT_OPTION, 0
.equ WAIT_RUSAGE, -64
```

```
Fork:
        movq $SYS_FORK, %rax
        syscall
        ret
```

The objective of this function, Fork, is to act as a wrapper around the SYS_FORK syscall. All it does is pass the given arguments to the operating system.

```
Execute:
        movq $SYS_EXECVE, %rax
        syscall
```

The objective of this function, Execute, is to act as a wrapper around the SYS_EXECVE syscall.
All it does is pass the given arguments to the operating system.

```
Wait:
        enter $128, $0
        lea WAIT_STAT_LOC(%rbp), %rsi
        movl $WAIT_OPTION, %edx
        lea WAIT_RUSAGE(%rbp), %rcx
        movq $SYS_WAIT4, %rax
        syscall
        movl WAIT_STAT_LOC(%rbp), %eax
        leave
        ret
```

The objective of this function, Wait, is to simplify the usage of the SYS_WAIT4 syscall. It takes in the process ID of the forked process, and creates a memory location for the return value of the process to be stored. It then passes the process id and a pointer to the memory location to the system. Once the system call finishes, the function returns the return value stored in the memory location by SYS_WAIT4.

Plot

```
        .section .rodata
        .global PLOT_OUTPUT_FILE

PROGRAM:
        .string "/usr/bin/gnuplot"
COMMAND:
        .string "set terminal png; set output '/
           home/debian/CS118-Lab-10-C-Output.png
           '; plot [-5:5] "
ARGUMENT:
        .string "-e"
ARGUMENT_ENVP:
        .quad 0
PLOT_OUTPUT_FILE:
        .string "/home/CS118-Lab-10-C-Output.png
           "
```

This is the read-only data section of the Plot file, and it contains some important things.
First, the PROGRAM variable contains the path to the gnuplot program.
Second, the COMMAND variable contains a template argument for the gnuplot program.
Third, the ARGUMENT variable contains a required argument for the gnuplot program.
Fourth, the ARGUMENT_ENVP variable is the environment pointers that gnuplot will be called with, as you can see, there is only one entry, which is the null-terminator.
Fifth, the PLOT_OUTPUT_FILE variable is the path to where the web server can find the image created by gnuplot.

```
.text
.global Plot
```

```
Command:
        lea COMMAND, %rax
        xorq %rcx, %rcx
```

The objective of the Command function is simply to store a copy of the COMMAND string, above, into a buffer, with a given string appended onto it.
The string that will be appended onto the output should be passed in rdi, while the output buffer should be passed in rsi.
First, the function stores a pointer to the COMMAND variable, which is used as a template, into rax.
It also sets rcx to zero, because it will be used as a loop iterator variable.

```
Command_While_1:
        movb (%rax, %rcx, 1), %r8b
        test %r8b, %r8b
        jz Command_While_2
        movb %r8b, (%rsi, %rcx, 1)
        incq %rcx
        jmp Command_While_1
Command_While_2:
```

Next, the function enters its first loop, the objective of which is to copy the COMMAND template string into the output buffer. The function just iterates through each character of the COMMAND string until it reaches a null pointer, at which point it stops copying and exits the loop.

```
        xorq %r9, %r9
Command_While_3:
        movb (%rdi, %r9, 1), %r8b
        test %r8b, %r8b
        jz Command_While_4
        movb %r8b, (%rsi, %rcx, 1)
        incq %rcx
        incq %r9
        jmp Command_While_3
Command_While_4:
```

After the first loop has ended, the function sets r9 to zero, because it will be used as the index that is currently being copied from the string contained in rdi.
Now, the function enters the second loop, in which it appends the string contained in rdi onto the output buffer.
Once the end of the string contained in rdi is reached, the loop ends.

```
movb $0, (%rsi, %rcx, 1)
ret
```

Once the second loop has exited, the function writes a null terminator to the end of the output buffer.
Subsequently, the function returns.

```
.equ Plot_ARGV3, -8
.equ Plot_ARGV2, -16
.equ Plot_ARGV1, -24
.equ Plot_ARGV0, -32
.equ Plot_Command, -256
```

These are the stack variables used by the PlotInternal function.
the Plot_ARGVx variables are entries in gnuplot's arguments, and the Plot_Command variable is a string buffer for the formatted gnuplot command to be stored in.

```
PlotInternal:
        enter $256, $0
        lea Plot_Command(%rbp), %rsi
        call Command
```

The purpose of the PlotInternal function is to simply take in a single string, that represents a mathematical function, and append the given string onto the COMMAND string above using the Command function. It will then package that command, along with a few other required commands, into a two dimensional array that will be passed to Execute as gnuplot's argv. First, the function calls the Command function, which places the command string onto the stack, in Plot_Command.

Address	Name	Type	Value
RBP-8	argv[3]	char*	?
RBP-16	argv[2]	char*	?
RBP-24	argv[1]	char*	?
RBP-32	argv[0]	char*	?
RBP-256	command	char[224]	"set term..."

```
lea Plot_Command(%rbp), %rax
movq %rax, Plot_ARGV2(%rbp)
lea ARGUMENT, %rax
movq %rax, Plot_ARGV1(%rbp)
lea PROGRAM, %rax
movq %rax, Plot_ARGV0(%rbp)
xorq %rax, %rax
movq %rax, Plot_ARGV3(%rbp)
```

After the command has been formatted and stored on the stack, PlotInternal has to build the argv for gnuplot.
It will consist of four things, gnuplot's path, a command flag, the command itself, and a null terminator.
After building the argument list, this is what the stack looks like.

Address	Name	Type	Value
RBP-8	argv[3]	char*	NULL
RBP-16	argv[2]	char*	&command
RBP-24	argv[1]	char*	"-e"
RBP-32	argv[0]	char*	"gnuplot"
RBP-256	command	char[224]	"set term..."

```
lea PROGRAM, %rdi
lea Plot_ARGV0(%rbp), %rsi
lea ARGUMENT_ENVP, %rdx
call Execute
#Exection does not continue
```

After setting up the argv for gnuplot, PlotInternal has to pass a pointer to the argv array, as well as an envp array, to Execute.
First, it loads the address of the argv array into rsi, then it loads the path to gnuplot into rdi, and subsequently loads the address of the empty envp array into rdx.
After that, it calls execute. There's no need to return after it, because execute will never return.

```
Plot:
        push %r12
        movq %rdi, %r12
        call Fork
        test %rax, %rax
        jnz Plot_Parent
Plot_Child:
        movq %r12, %rdi
        call PlotInternal
        #Execution does not continue
Plot_Parent:
        movq %rax, %rdi
        call Wait
Plot_End:
        pop %r12
        ret
```

The objective of the Plot function is to provide a wrapper around Fork and PlotInternal. First, the function first calls Fork, which creates a child process, then, it checks if it is the child process or not, by comparing the value returned by Fork to zero, if it is zero, it is the child, if not, it is the parent.
If it is the child, it calls PlotInternal, which will call gnuplot.
If it is the parent, it calls Wait, which is a wrapper around SYS_WAIT4.
After the parent's call to Wait is finished, the function returns.

Output

Lab10C - Slope

Point 1: 10 12

Point 2: 20 32 Submit

Slope: 2.000000