CSE 320 Spring 2016 - Homework 1

Due 02/05/2016 @ 11:59pm

Introduction

This assignment is going to introduce you to Unix philosophy. Basically, the idea is that software should be created in small modular pieces that do a simple task and do that task well. Combining small modular pieces we can then create complex powerful programs.

In this assignment we will create a tool which accepts hexadecimal values representing MIPS instructions on `stdin` and produce some information about these instructions on `stdout`. The output should be formatted in such a way that we can pipe it to tools such as `grep`, `sort`, etc. This program will also accept command line arguments that will adjust the type of information that is printed to `stdout`. Like any C program you write, your tool should return a exit status such as `EXIT_FAILURE` or `EXIT_SUCCESS` (found in `stdlib.h`).

In this assignment, you will learn about the C compiler and compilation tools such as `make`. You should be aware of C's system-to-system dependency. For example, data type's size and endianness change depending on your working operating system or architecture.

The goal of this assignment is to write a basic but useful C program. It should help you become familiar with piping and redirection as well as other basic command line tools.

> ⚠ C is a **spartan** language in comparison to Java or Python. Meaning it won't do you any favors by warning you about mistakes in your code, you must use the tools at your disposal to find errors and debug your code.

Compiling with GCC

Let’s start off by making a simple hello world program and explaining what each part means.

```
1. #include <stdio.h>
2. #include <stdlib.h>

3. int main(int argc, char *argv[]) {
4.       printf("Hello, World!\n");
5.       return EXIT_SUCCESS;
6. }
```

Lines 1 and 2

Lines **1** and **2** are the C preprocessor statements which include function prototypes for some of the functions in the C standard library (aka libc). For now you can just vaguely relate these to the import statements you might find at the top of a java file.

```
import java.util.List;
```

The C preprocessor is a very powerful tool and we will learn about it in future assignments. For now, just accept this basic explanation of what these two lines do. The `#include` directive takes the contents of the `.h` file and copies it into the `.c` file before the C compiler actually translates the `C` code.

> Files that end in `.h` are called header files. They typically contain preprocessor macros, function prototypes, struct information, and typedefs.
>
> You can view the source for stdlib.h and stdio.h in the freebsd repository.

Line 3

Line **3** is how you describe the `main` function of a C program. In C, if you are creating an executable program it must have **one** and **ONLY one** main function. Any main function you write in this course **MUST** return an integer value (in older textbooks/documentation they might return void; watch out).

This is sort of similar to the main declaration in Java (ignoring the return type and fancy class stuff Java gives you). In Java, your arrays have a field which contain just how many elements are in the array. In C, arrays contain no such information. So to remedy this issue two arguments are passed, `argc`, which contains how many elements are in the array and `argv`, which is an array of strings which contains each of the arguments passed on the command line. Even if no arguments are passed by the user, `argv` will typically contain at least one argument which is usually the name of the binary being executed.

> ℹ If you look through other C programs, you might see that there are quite a few different ways to declare `main`. In this course you should declare `main` just as it is in this hello world example unless specified otherwise in the homework assignment.
>
> ⚠ When we say that only one `main` function should exist in your whole program we really mean it. This is not like java where you can have a different `main` in every file and then choose which `main` you want to run. If you have more than one `main` when you try to compile it will give you an error. For example, assume you had two files `main1.c` and `main2.c` and you tried to compile them both into one program (reasonable thing to do). One thing you forgot is that both `main1.c` and `main2.c` both have a `main` function defined in them. So when you try to compile it you get the following linker error.
>
> ```
> /tmp/cc8eYGEA.o: In function 'main':
> main2.c:(.text+0x0): multiple definition of 'main'
> /tmp/ccaaqneq.o:main1.c:(.text+0x0): first defined here
> collect2: error: ld returned 1 exit status
> ```
>
> This error means that this function is defined twice within your program. This concept also extends to all functions. Two functions **CAN NOT** have the same name under normal conditions. If they do you will receive an error similar to this.
>
> In addition, function overloading is not allowed in C.
> Example: Assume you had the file `func.c` with the following function declarations.
>
> ```
> void func(int a);
> void func(int a, int b);
> ```
>
> This will result in the following error:
>
> ```
> func.c:5:6:error: conflicting types for ‘func’
> void func(int a, int b) {
>      ^
> func.c:1:6: note: previous definition of ‘func’ was here
> void func(int a) {
> ```

Line 4

Line **4** is how this program is printing out its values to standard out. The `printf` function can be compared to the `System.out.print()` or `System.out.printf()` functions in Java. This function accepts a `char*` argument known as the format string (assume for now char* is equivalent to the Java String type). This will work fine for when you know ahead of time what you want to print, but what if you want to print a variable?

If you just assumed C is like Java, you may try to concatenate strings in the following form:

```
int i = 5;
printf("The value of i is " + i + "\n");
```

If you try to compile this code, GCC may give you some of the following cryptic error messages:

```
error: invalid operands to binary + (have ‘char *’ and ‘char *’)
```

or

```
warning: format not a string literal and no format arguments [-Wformat-security]
```

Unfortunately in C, we do not have string concatenation via the `+` operator. Luckily for you the `printf` function also takes a variable number of arguments after the format string. So if you put a special value inside of the format string and then provide a variable of the correct type as an argument after the format string you can print out the value of variables using `printf`.

```
#include <stdio.h> // If you want to use printf include this header file.
#include <stdlib.h> // if you want to use EXIT_SUCCESS

int main(int argc, char *argv) {
    int i = 5;
    // Print out that the value of i is 5.
    printf("The value of i is %d\n", i);
    return EXIT_SUCCESS;
}
```

> You can view a list of all printf formats by viewing this. Alternatively you can use the command

> `man 3 printf` in your terminal to view the documentation for `printf` as well.

Line 5

Line **5** is the end of the main function. The value returned in main is the value that represents the return code of the program. In *nix when a program exits successfully, the value returned is usually zero. When it has some sort of an error, the value is usually an non-zero number. Since these values are defined by programmers and they may be different depending on the system you are using, it is usually best to use the constants **EXIT_SUCCESS** and **EXIT_FAILURE** which are defined in `stdlib.h` for simple cases.

> ℹ The term *nix is used for describing operating systems that are derived from the Unix operating system (ex. BSD, Solaris) or clones of it (ex. Linux).

Compiling the program

Let's compile the program which was located at the top of this section.

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

int main(int argc, char *argv[]) {
    printf("Hello, World!\n");
    return EXIT_SUCCESS;
}
```

To do that you should navigate on the command line to where the `.c` file is located. If the file was called `helloworld.c`, type the following command to compile the program.

```
$ gcc helloworld.c
$
```

> ℹ The '$' is the command line prompt.

If no messages print, that means there were no errors and the executable was produced. To double check that your program produced a binary you can type the `ls` command to list all items in the directory.

```
$ ls
a.out helloworld.c
$
```

The file **a.out** is your executable program. To run this program you should put a `./` infront of the binary name.

```
$ ./a.out
Hello, World!
$
```

> The `./` has a special meaning. The `.` translates to the path of the current directory. So if your file was in the user `cse320`'s home directory then when you type `./a.out` this would really translate to the path `/home/cse320/a.out`.

Compilation flags

Let's modify the helloworld program to sum up the values from 0 to 5.

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

int main(int argc, char *argv[]) {
    int i, sum;
    for(i = 0; i < 6; i++) {
        sum += i;
    }
    printf("The sum of all integers from 0-5 is: %d\n", sum);
    return EXIT_SUCCESS;
}
```

Compile this program just like you did in part 1 and run it.

```
$ gcc helloworld2.c
$ ./a.out
The sum of all integers from 0-5 is: 15
$
```

This program compiled with no errors and even produced the correct result. Shockingly, there is actually a subtle, but potentially dangerous bug in this code. Since there are many bugs like this in C, the developers of our C compiler have built in some flags to help us find them.

Let's compile the same piece of code again but with a slightly different command. Add the flags `-Wall` and `-Werror` to the `gcc` command.

```
$ gcc -Wall -Werror helloworld2.c
helloworld2.c:7:3: error: variable 'sum' is uninitialized when used here [-
Werror,-Wuninitialized]
                sum += i;
                ^~~
helloworld2.c:5:12: note: initialize the variable 'sum' to silence this
warning
        int i, sum;
                  ^
                   = 0
```

> ℹ Depending on your compiler (gcc vs. clang vs. gcc versions vs. etc) the above error and message may differ. This is a key point of this assignment.
> ℹ The flag `-Wall` enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.
> ℹ The flag `-Werror` makes all warnings into hard errors. Source code which triggers warnings will be rejected.

This error is telling us that we used the variable sum without initializing it. Why does this matter? The C language does not actually specify how the compiler should treat uninitialized variables. Most implementations of the C compiler will hopefully be nice and zero them out for you, but really you have no idea the behavior of how this situation will be handled. This can lead to undefined behavior and the way that this program works on another computer may not be the same way it works on yours (you may of heard some horror stories about students homework working on their computer but it didn't work on the graders). How do you fix this potential error? Simply assigning the variable `sum` to zero is sufficient enough.

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

```c
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int i, sum = 0;
    for(i = 0; i < 6; i++) {
        sum += i;
    }
    printf("The sum of all integers from 0-5 is: %d\n", sum);
    return EXIT_SUCCESS;
}
```

Compile the program again and you should no longer see any errors.

```
$ gcc -Wall -Werror helloworld2.c
$ ./a.out
The sum of all integers from 0-5 is: 15
$
```

In this class, you **MUST ALWAYS** compile your assignments with the flags `-Wall -Werror`. This will help you locate mistakes in your program and the grader will compile your assignment with these flags as well.

> ⚠ Consider this your warning, `-Wall -Werror` are necessary, please do not progress through your assignment without using these flags and attempt to fix the errors they highlight last minute.

Header files

There are some coding practices that you should become familiar with in C from the beginning, in order to give you a better chance at writing clear and correct programs. As we learned in class, the C compiler reads through your code once and only once. This means all functions and variables you use must be declared in advance of their usage or the compiler will not know how to compile and exit with errors. This is why we have header files, we declare all of our function prototypes in a `.h` file and `#include` it in our `.c` file. This is so we can write the body of our functions in any order and call them in any order we please.

A header file is just a file which ends in the `.h` extension. Typically you declare function prototypes, define struct and union types, #include other header files, #define constants and macros, and typedef. Some header files also expose global variables, but this is strongly discouraged unless you know what your doing.

When you define function prototypes in a `.h` file, you can then define the body of the function inside of any `.c` file. Typically though, if the header file was called `example.h`, we would define the functions in `example.c`. If we were producing a massive library like stdlibc, you may instead declare all the function prototypes in a single header file but put each function definition in its own file. Its all a preference, but these are two common practices. You should never be defining function bodies in the header though, this will just cause you issues later.

There are two ways to specify where the include directive looks for header files. If you use `<>`, when the preprocessor encounters the include statement it will look for the file in a predefined location on your system (usually `/usr/include`). If you use `""`, the preprocessor will look in the current directory of the file being processed. Typically system and library headers are included using `<>`, and custom headers that you have made for your program are included using `""`.

Header file example

```c
// example.h
#include <stdio.h>
#include <stdlib.h>
#include <stdlib.h>

#define TRUE 1
#define FALSE 0

struct student {
    char *first_name;
    char *last_name;
    int age;
    float gpa;
};

int foo(int a, int b);
void bar(void);
```

```c
// example.c
#include "example.h"

int main(int argc, char *argv[]) {
    bar();
    return EXIT_SUCCESS;
}
```

```
void bar(void) {
    printf("foo: %d", foo(2, 3));
}

int foo(int a, int b) {
    return a * b;
}
```

Header Guard

While using header files solves one issue, they create issues of their own. What if multiple files include the same header file? What if header file A includes header file B, and header file B includes header file A? If we keep including the same header file multiple times, this will make our source files larger and slow down the compilation process. It may also cause errors if there are variables declared in the code. If two files keep including each other how does the compiler know when to stop? In the following examples we demonstrate what is known as a header guard. The header guard is used to prevent double and cyclic inclusion of a header file.

Header Guard example

For example:
`grandfather.h`:

```
struct foo {
    int member;
};
```

`father.h`:

```
#include "grandfather.h"
```

`child.c`:

```
#include "grandfather.h"
#include "father.h"
```

The linker will create a temporary file that has literal copies of the `foo` definition twice and this will create a compiler error since the compiler does not know which definition takes precedent (and to its credit, neither do we). The fix:
`grandfather.h`:

```
#ifndef GRANDFATHER_H
#define GRANDFATHER_H
struct foo {
    int member;
};
#endif
```

`father.h`:

```
#include "grandfather.h"
```

`child.c`:

```
#include "grandfather.h"
#include "father.h"
```

`#ifndef`, `#define`, `#endif` are preprocessor macros that prevent the double inclusion. This is because when the `father.h` file includes `grandfather.h` for the second time the `#ifndef` macro returns false so the second definition for `foo` is never included.

Read here for more information.

> ℹ You should always use header files and guards in your assignments.
> ℹ You may see header guards also called include guards; include guards and header guards the same.
> ℹ Newer compilers now support what is known as #pragma once. This directive performs the same operation as the header guard, but it may not be a cross platform solution when considering older machines.

GNU Make

As you program more in C, you will continue to add more flags and more files to your programs. To type these commands over and over again will eventually become an error laden chore. Also as you add more files, if you rebuild every file every time, even if it didn't change, it will take a long time to compile your program. To help alleviate this issue build tools were created. One such tool is GNU Make (you will be required to use make in this class). Make itself has lots of options and things that can be configured. While we will try to guide you in what you need to know for this class, to truly use `make` to its full potential you will need to read and understand the make manual.

Start by creating a file called `Makefile` in the same directory as your `helloworld.c` and `helloworld2.c` files.

Now let's add our first command to the `Makefile`.

```
# Attempt 1
helloworld2:
	gcc helloworld2.c
```

> ⚠ Spacing is important in makefiles. The space between the left side and gcc must be a **TAB** character, and the file should always have a newline at the bottom with nothing else.

Now let's run the make file. Simply type `make` in the terminal and watch what happens.

```
$ make
gcc helloworld2.c
$
```

After typing `make`, the make program searched the current directory for the `Makefile` and executed the command to build `helloworld2.c`. Now add a second command for building `helloworld.c`.

```
# Attempt 2
helloworld:
	gcc helloworld.c

helloworld2:
	gcc helloworld2.c
```

Save the file and type `make` again.

```
$ make
gcc helloworld.c
```

```
$
```

This time make created the first program but not the second. The names `helloworld` and `helloworld2` are called targets. If we provide a target to the `make` command we can tell it which program we want to build.

```
$ make helloworld2
gcc helloworld2.c
$ make helloworld
gcc helloworld.c
$
```

OK this sort of works. What about all the flags you are required to use?

```
# Attempt 3
helloworld:
    gcc -Wall -Werror helloworld.c

helloworld2:
    gcc -Wall -Werror helloworld2.c
```

Great! We added the C compile flags to both commands. If you rerun the previous commands you should now see:

```
$ make helloworld
gcc -Wall -Werror helloworld.c
$ make helloworld2
gcc -Wall -Werror helloworld2.c
$
```

This is certainly better than having to type them out each time. The flags used between `helloworld` and `helloworld2` are identical. Why not group that value in one place and then share it between the two commands.

```
# Attempt 4
CFLAGS = -Wall -Werror
helloworld:
    gcc $(CFLAGS) helloworld.c

helloworld2:
    gcc $(CFLAGS) helloworld2.c
```

We made a variable called `CFLAGS` and then accessed the contents of that variable by using `$()`. If you run the above command again you should get identical output as before.

```
$ make helloworld
gcc -Wall -Werror helloworld.c
$ make helloworld2
gcc -Wall -Werror helloworld2.c
$
```

What if you wanted to build both programs at once? You can do that by adding a directive called `all:` and then just calling `make`.

```
# Attempt 5
CFLAGS = -Wall -Werror

all: helloworld helloworld2

helloworld:
    gcc $(CFLAGS) helloworld.c

helloworld2:
    gcc $(CFLAGS) helloworld2.c
```

Type `make` and you should see the following output.

```
$ make
gcc -Wall -Werror helloworld.c
gcc -Wall -Werror helloworld2.c
$
```

Now type the `ls` command.

```
$ ls
a.out helloworld.c helloworld2.c
```

As you can see from the make command, it built both programs but there's only one final binary! This happens because by default gcc produces an executable called `a.out`. The binary produced by `helloworld2.c` overwrote the binary produced by `helloworld.c`. To fix this, gcc has a flag `-o` which allows you to change the name of the executable produced.

```
# Attempt 6
```

```
CFLAGS = -Wall -Werror

all: helloworld helloworld2

helloworld:
    gcc $(CFLAGS) helloworld.c -o helloworld

helloworld2:
    gcc $(CFLAGS) helloworld2.c -o helloworld2
```

Now if you type make again, it should build both programs, but this time you should have 2 new binary programs called `helloworld` and `helloworld2`.

```
$ make
gcc -Wall -Werror helloworld.c -o helloworld
gcc -Wall -Werror helloworld2.c -o helloworld2
$ ls
a.out helloworld helloworld.c helloworld2 helloworld2.c
$
```

> ⚠ Be careful with what name you give to `-o`. Students in previous semesters have overwritten their source files because they named their executable the same name and extension as the source. **This is why you should use git! It allows you to save your progress, and then it can be recovered if needed.**.

Make has all of these special values which you can use. So if you always wanted to name your executable the same as your target you could replace the values of the name with the following automatic variable:

```
# Attempt 7
CFLAGS = -Wall -Werror

all: helloworld helloworld2

helloworld:
    gcc $(CFLAGS) helloworld.c -o $@

helloworld2:
    gcc $(CFLAGS) helloworld2.c -o $@
```

> ℹ The `$@` is a special value that gets replaced by the name of the target. In the above example `$@`

> gets replaced by the values `helloworld` and `helloworld2` .

You should get the same exact output as before.

```
$ make
gcc -Wall -Werror helloworld.c -o helloworld
gcc -Wall -Werror helloworld2.c -o helloworld2
$ ls
a.out helloworld helloworld.c helloworld2 helloworld2.c
$
```

Finally, to sum up our simple introduction to makefiles, typically you also create what is known as a `clean` target. This target simply removes all the binaries and artifacts left over from the build process. In this example, we will create another variable to store the names of all the different binary programs so we can remove them all.

```
# Attempt 8
CFLAGS = -Wall -Werror
BINS = helloworld helloworld2

all: $(BINS)

helloworld:
    gcc $(CFLAGS) helloworld.c -o $@

helloworld2:
    gcc $(CFLAGS) helloworld2.c -o $@

clean:
    rm -f $(BINS)
```

Try running the command make clean to remove all the binaries produced.

```
$ make clean
rm -f helloworld helloworld2
$ ls
helloworld.c helloworld2.c
$
```

You can also use `make` for combining various files together. Let’s assume that the `main` function is inside the file `main.c` , and the function `function` is defined in `functions.c` . Also, you must have the function prototypes which you wish to use in other files defined in `functions.h` (that way you can include them in `main.c`). You want to create separate object files and then link those object files together. When gcc creates object files, they have the extension `.o`. If we pass gcc the `-c` flag it will create these object files. The object files contain binary code and data in a special format with extra information to combine with other object files to create a full executable or library during the linking stage of compilation.

```
# Attempt 9
CFLAGS = -Wall -Werror
BINS = helloworld helloworld2 assignment

all: $(BINS)

helloworld:
	gcc $(CFLAGS) helloworld.c -o $@

helloworld2:
	gcc $(CFLAGS) helloworld2.c -o $@

assignment: main.o functions.o
	gcc $(CFLAGS) $^ -o $@

main.o: main.c
	gcc $(CFLAGS) -c $^

functions.o: functions.c
	gcc $(CFLAGS) -c $^

clean:
	rm -f *.o $(BINS)
```

There are more advanced and complex ways to create makefiles but for this assignment this is sufficient. It should be similar to the file described above except it should only build your assignment.

Datatype sizes

Perhaps you were tasked with building a fancy number converter which relied heavily on the size of data types in C. Then you were asked to make the program work not only on computer architecture but on a set of different ones with different word sizes, etc. Would your program still work?

In a language like Java, much of these issues are hidden from the programmer. The JVM creates another layer of abstraction which can allow the programmer to believe all datatypes are of same size no matter the underlying architecture. When you program in C, you don't have this luxury. You have to consider everything about the systems you are working on. If you need it to work across multiple systems, you need to test and compare values and alter your code and logic accordingly.

C lacks the ability to add new datatypes to its specification. Instead, we work with models known as LP64, ILP64, LLP64, ILP32, and LP32. The `I` stands for **INT**, the `L` stands for **LONG** and the `P` stands for **POINTER**. The number after the letters describes the maximum bit size of the data types.

The typical sizes of these models are described below in the following table (the numbers are how many bits):

Datatype	LP64	ILP64	LLP64	ILP32	LP32
char	8	8	8	8	8
short	16	16	16	16	16
_int32		32			
int	32	64	32	32	16
long	64	64	32	32	32
long long			64		
pointer	64	64	64	32	32

After taking a quick look at this table, you should notice that this causes an issue. An integer on one machine is not the same size as an integer on another machine. To prove this to yourself, you may want to use a special operator in the C language known as `sizeof`. The operator `sizeof` will tell you the size in bytes of a specific datatype. As an exercise, you should create the following program and run it in your development environment and on Sparky and compare the results.

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

int main(int argc, char *argv[]) {
    /* Basic data types */
    printf("=== Basic Data Types ===\n");
    printf("short: %lu bytes\n", sizeof(short));
    printf("int: %lu bytes\n", sizeof(int));
    printf("long: %lu bytes\n", sizeof(long));
    printf("long long: %lu bytes\n", sizeof(long long));
```

```
        printf("char: %lu byte(s)\n", sizeof(char));
        printf("double: %lu bytes\n", sizeof(double));
        /* Pointers */
        printf("=== Pointers ===\n");
        printf("char*: %lu bytes\n", sizeof(char*));
        printf("int*: %lu bytes\n", sizeof(int*));
        printf("long*: %lu bytes\n", sizeof(long*));
        printf("void*: %lu bytes\n", sizeof(void*));
        printf("double*: %lu bytes\n", sizeof(double*));
        /* Special value - This may have undefined results... why? */
        printf("=== Special Data Types ===\n");
        printf("void: %lu byte(s)\n", sizeof(void));
        return EXIT_SUCCESS;
    }
```

To further illustrate why this is a problem, consider the following program.

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

    int main(int argc, char *argv[]) {
        // 0x200000000 -> 8589934592 in decimal
        long value = strtol("200000000", NULL, 16);
        printf("value: %ld\n", value);
        return EXIT_SUCCESS;
    }
```

Run this program on both your Linux environment and Sparky and look at the results. Why does this happen?

How do you handle this? In libc, there exists a header `stdint.h` which has special types defined to make sure that if you use them, no matter what system you are on it can guarantee that they are the correct size.

> You can find the source for `stdint.h` and all other C header files that are included using <> in the `/usr/include` directory.

Endianness

Since you will be working with multi byte values and different architectures, the endianness of each architecture should also be taken into account. There are many ways to detect what endianness your machine is, but the easiest way which does not involve any fancy preprocessor magic or libraries is performed as follows:

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

int main(int argc, char *argv[]) {
    unsigned int i = 1;
    char *c = (char*)&i; // Convert the LSB into a character
    if(*c) {
        printf("little endian\n");
    } else {
        printf("big endian\n");
    }
    return EXIT_SUCCESS;
}
```

Can you think of why this works? Could you explain it if asked on an exam?

Assembly

When you compile a program in C, it is translated to an assembly source file during the compilation process. We will now illustrate how a C file is compiled differently depending on the architecture, and how to inspect the intermediate file generated during the compilation process. You should be aware of this because its possible that something which has great performance in one system could have terrible performance in another with the exact same C implementation. To figure out what is going wrong you may need to inspect the assembly code.

We will compile the file `asm.c` for x86, x86-64, and 32-bit SPARC. To compile a program to the `x86` instruction set on an `x86-64` machine, you will need to install `gcc-multilib`.

> $ sudo apt-get install gcc-multilib

After installing `gcc-multilib` you will be able to compile to 32-bit binaries with the `-m32` flag when using gcc on a 64-bit platform.

```
// asm.c
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>

int main(int argc, char *argv[]) {
        char buffer[1024];
        // Get user input
        fgets(buffer, 1024, stdin);
        int64_t value = strtoll(buffer, NULL, 10);
        printf("You entered %" PRId64 "\n", value);
        return EXIT_SUCCESS;
}
```

Compile each program respectively:

```
$ gcc -Wall -Werror -m64 asm.c -o 64.out
$ gcc -Wall -Werror -m32 asm.c -o 32.out
```

Run each program and you should see this output.

```
$ ./64.out
75
You entered 75
$ ./32.out
75
You entered 75
```

As you can see, even though both programs are compiled for different architectures, they still produce the same results. These programs are assembled using different instruction sets though. To see this we will compile our programs with the `-S`. This flag will store the intermediate assembly of the program in a `.s` file.

Start by generating the assembly for `x86-64` by using the following command

- `$ gcc -Wall -Werror -m64 -S asm.c`

Take a look at `asm.s` which was just generated in current working directory.

```
# x86-64 assembly for asm.c
    .file   "asm.c"
    .section    .rodata
```

```
.LC0:
    .string "You entered %ld\n"
    .text
    .globl  main
    .type   main, @function
main:
.LFB2:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $1072, %rsp
    movl    %edi, -1060(%rbp)
    movq    %rsi, -1072(%rbp)
    movq    %fs:40, %rax
    movq    %rax, -8(%rbp)
    xorl    %eax, %eax
    movq    stdin(%rip), %rdx
    leaq    -1040(%rbp), %rax
    movl    $1024, %esi
    movq    %rax, %rdi
    call    fgets
    leaq    -1040(%rbp), %rax
    movl    $10, %edx
    movl    $0, %esi
    movq    %rax, %rdi
    call    strtoll
    movq    %rax, -1048(%rbp)
    movq    -1048(%rbp), %rax
    movq    %rax, %rsi
    movl    $.LC0, %edi
    movl    $0, %eax
    call    printf
    movl    $0, %eax
    movq    -8(%rbp), %rcx
    xorq    %fs:40, %rcx
    je  .L3
    call    __stack_chk_fail
.L3:
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

```
.LFE2:
	.size	main, .-main
	.ident	"GCC: (Ubuntu 5.2.1-22ubuntu2) 5.2.1 20151010"
	.section	.note.GNU-stack,"",@progbits
```

Now compile it for `x86` using the following command:

- `$ gcc -Wall -Werror -m32 -S asm.c`

Again, take a look at `asm.s` which was just generated in current working directory.

```
# x86 assembly for asm.c
	.file	"asm.c"
	.section	.rodata
.LC0:
	.string "You entered %lld\n"
	.text
	.globl	main
	.type	main, @function
main:
.LFB2:
	.cfi_startproc
	leal	4(%esp), %ecx
	.cfi_def_cfa 1, 0
	andl	$-16, %esp
	pushl	-4(%ecx)
	pushl	%ebp
	.cfi_escape 0x10,0x5,0x2,0x75,0
	movl	%esp, %ebp
	pushl	%ecx
	.cfi_escape 0xf,0x3,0x75,0x7c,0x6
	subl	$1060, %esp
	movl	%ecx, %eax
	movl	4(%eax), %eax
	movl	%eax, -1052(%ebp)
	movl	%gs:20, %eax
	movl	%eax, -12(%ebp)
	xorl	%eax, %eax
	movl	stdin, %eax
	subl	$4, %esp
	pushl	%eax
	pushl	$1024
	leal	-1036(%ebp), %eax
	pushl	%eax
```

```
        call    fgets
        addl    $16, %esp
        subl    $4, %esp
        pushl   $10
        pushl   $0
        leal    -1036(%ebp), %eax
        pushl   %eax
        call    strtoll
        addl    $16, %esp
        movl    %eax, -1048(%ebp)
        movl    %edx, -1044(%ebp)
        subl    $4, %esp
        pushl   -1044(%ebp)
        pushl   -1048(%ebp)
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        movl    $0, %eax
        movl    -12(%ebp), %edx
        xorl    %gs:20, %edx
        je  .L3
        call    __stack_chk_fail
    .L3:
        movl    -4(%ebp), %ecx
        .cfi_def_cfa 1, 0
        leave
        .cfi_restore 5
        leal    -4(%ecx), %esp
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
    .LFE2:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 5.2.1-22ubuntu2) 5.2.1 20151010"
        .section    .note.GNU-stack,"",@progbits
```

Additionally you can log into sparky, and use the C compiler on that machine. It will generate 32-bit SPARC assembly

- gcc -Wall -Werror -S asm.c

```
    # 32-bit SPARC assembly
            .file   "asm.c"
            .section        ".rodata"
```

```
        .align 8
.LLC0:
        .asciz  "You entered %lld\n"
        .section        ".text"
        .align 4
        .global main
        .type   main, #function
        .proc   04
main:
        save    %sp, -1128, %sp
        st      %i0, [%fp+68]
        st      %i1, [%fp+72]
        add     %fp, -1032, %g1
        mov     %g1, %o0
        mov     1024, %o1
        sethi   %hi(__iob), %g1
        or      %g1, %lo(__iob), %o2
        call    fgets, 0
         nop
        add     %fp, -1032, %g1
        mov     %g1, %o0
        mov     0, %o1
        mov     10, %o2
        call    strtoll, 0
         nop
        std     %o0, [%fp-8]
        sethi   %hi(.LLC0), %g1
        or      %g1, %lo(.LLC0), %o0
        ld      [%fp-8], %o1
        ld      [%fp-4], %o2
        call    printf, 0
         nop
        mov     0, %g1
        mov     %g1, %i0
        return  %i7+8
         nop
        .size   main, .-main
        .ident  "GCC: (GNU) 4.9.1"
```

Assembly Analysis

As you can see the assembly generated for a particular architecture varies greatly even though it all accomplishes the exact same task on each system. The first thing you may notice is that the SPARC assembly is shorter than the other two (40 lines for SPARC, 67 lines for x86, and 51 lines for x86-64) or that the registers used are different in all three examples.

Next lets take a look at how the format string in the `printf` call got translated:

```
printf("You entered %" PRId64 "\n", value);
```

```
.string "You entered %ld\n"  # x86-64; 64-bits
.string "You entered %lld\n" # x86; 32-bits
.asciz  "You entered %lld\n" # SPARC; 32-bits
```

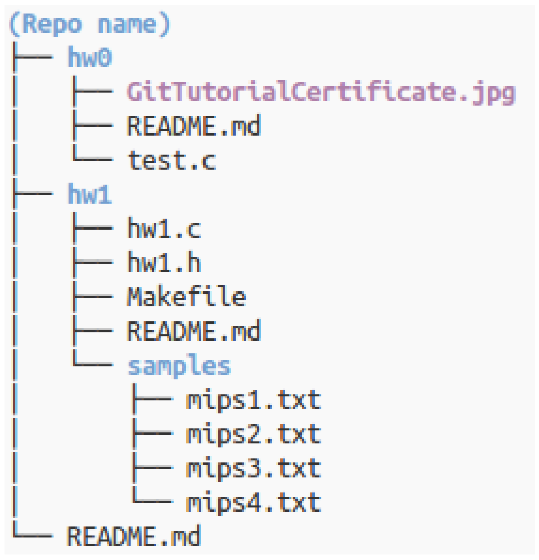
If you look at the strings you will see that PRId64 got translated to different formats `%ld` and `%lld`. This is because the `int64_t` is translated to different types depending on the platform to guarantee that it is atleast 64-bits wide. If you look at the SPARC code, you may notice that there are `nop` instructions after the call to `printf`, `strtoll`, `fgets`, and `return`. This is is because of a technique known as delayed branching used in the SPARC architecture.

In the x86 assembly you can see `subl` and `pushl` instructions which are used to manipulate the stack before calling functions. These instructions are absent from the x86-64 example. This is because x86 architecture has half the amount of registers as x86-64 architectures so the convention is to push arguments for a function call to the stack to compensate for this. At the core, the Application Binary Interface differs between the systems. There are also various other differences that can't be seen by looking at the assembly such as variable sized instruction formats, but in general you should just be aware that these 5 lines of code within the `main` function get translated very differently depending on the machine.

Part I - Getting started

- Begin by cloning the CSE320 git repository to your working directory.
  - `git clone https://gitlab01.cs.stonybrook.edu/cse320/hw1.git`
  - Follow the directions from HW0 to pull the code for this remote repository into your repository.
- Inside the `hw1` directory you should see the `sample` directory. This directory contains sample input files to test your program with.

- Create `hw1.c` , `hw1.h` , `Makefile` , and `README` in the `hw1` directory.



```
(Repo name)
├── hw0
│   ├── GitTutorialCertificate.jpg
│   ├── README.md
│   └── test.c
├── hw1
│   ├── hw1.c
│   ├── hw1.h
│   ├── Makefile
│   ├── README.md
│   └── samples
│       ├── mips1.txt
│       ├── mips2.txt
│       ├── mips3.txt
│       └── mips4.txt
└── README.md
```

Part II - Program description and operation

As you become increasing familiar with the *nix environment, most programs will have some sort of usage statement if you just run the program with no arguments. Here, we present you the usage statement of your program, and the conditions you must satisfy to complete this assignment.

```
$ ./mstat -h
Usage:  ./mstat [OPTION]
        ./mstat -h          Displays this help menu.
        ./mstat -i [-u]     Displays statistics about instruction types.
        ./mstat -r [-u]     Displays information about the registers.
        ./mstat -o [-u]     Displays number and percentage of opcodes used.

    Optional flags:
    -u           Displays human readable headers for the different outputs.
```

> The `-u` argument is optional and even redundant if paired with `-h` as the help menu is already in a human readable format.

Your program will have three different output formats. In addition to these outputs you must implement a user help menu or *usage* similar to the standard help menu found in programs like `ls` or `grep`. Your help menu and several output formats will be denoted by command line flags.

> ℹ Try typing `ls --help` to see what an ideal help menu looks like.
> ⚠ You are not permitted to use an argument parsing library such as `getopt` for this assignment you must practice parsing the arguments yourself.

These command arguments will be accessible in your code via the `argv` array of size `argc`. For ease of this first assignment, you can assume that all the program arguments will always appear in the same order.

Stdin, Stdout, and Stderr

Your program will read input from `stdin` and write output to `stdout` and write descriptive error strings to `stderr`. You should become familiar with what these three files represent. These are known as the standard streams. Prior to Unix, computer programs needed to specify and be connected to a particular I/O device such as magnetic tapes. This made portability nearly impossible. Later in the course we will delve deeper into "files" and how they represent abstract devices in Unix-like operating systems. For now understand that they work much like your typical `.txt` file they can written to and read from.

Here are some methods of reading and writing to the standard I/O streams.

```c
#include <unistd.h>
ssize_t read (int filedes, void *buf, size_t count);
ssize_t write(int filedes, void *buf, size_t count);
#include <stdio.h>
int fscanf (FILE *stream, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

> ℹ Note the `read` and `write` calls take the same arguments as MIPS read and write syscalls, this is an example of a high-level interface to a low-level syscall.

Typically, this is the arrangement of file descriptors to standard streams.

I/O	File descriptor
FILE* stdin	0

FILE* stdout	1
FILE* stderr	2

A file descriptor is simply an integer that is an indicator of a particular file. For example, if we were to open a file with

```
#include <sys/stat.h>
#include <fcntl.h>
int fd;
char *buffer[100];
if( (fd = open("test.txt", O_RDONLY) < 0){
    exit(EXIT_FAILURE);
}
```

we could then read from it using

```
#include <unistd.h>
ssize_t bytes = read(fd, buffer, 100);
```

There are many ways to access and handle files. Different approaches suit the needs of different programs.

- Read this page on approaches for handling files in C.

Input to your program

Your program will accept input via `stdin`. The input will consist of strings which are the hexadecimal representations of MIPS instructions (one instruction per line).

Input Example File:

```
0x3c011001
0x3424024f
0x24020004
0x0000000c
0x24020005
0x0000000c
```

```
0x24080001
0x24090002
0x240a0003
0x1048000f
0x1049000e
0x104a017e
0x00404025
0x3c011001
```

> ℹ You should use fgets, scanf, read, or fread to get input from the user. Then use the function strtoull to parse and error check your input.
>
> ℹ You can create your own input file with your own MIPS instructions using MARS. Take any Mips program, and assemble it. After assembling press the command `ctrl-d`. This should pop up a menu about dumping a section of your program. For the **memory segment** select the *.text* section and for the **dump format** select *Hexadecimal Text*. Finally select *Dump to File…* which will ask you were you want to save your instruction dump. This file won't have `0x` at the beginning of each hex number you can easily add this though. Use `$ sed -i.bak 's/^/0x/' filename` in the command line. Put the name of your file where "filename" is in the command. The argument `-i.bak` creates a backup of your file in case of an error.
>
> ℹ You will have to use C bitwise operations after parsing the hex values to figure out what type of instructions they are.

Your program will then parse the instructions to gather the statistical data required to create the data tables shown with each flag option.

To provide input to your program you will need to use the I/O redirection operators to take the content from the sample files and provide it to `stdin` of your program.

> ℹ You can find I/O redirection explanations and tutorials at the source. If the tldp explanation is too dense for you to understand, you can find a more beginner friendly tutorial here.

To provide the input to your program, you could redirect the instructions from the sample file to the `stdin` of your program using the `<` operator. This can be done like: `$ ./mstat -r < instructions.txt`

Alternatively you can use the pipe `|` operator to send the output of another program (such as `cat`) to the `stdin` of your program. This can be done like: `$ cat instructions.txt | ./mstat -r`

`-i` Instruction Information

When given the command line flag `-i` your program will produce some simple statistics in the following format. Each line will contain the instruction type, the count of this instruction type given in the input, and the percentage of this instruction type (based on the total number of instructions provided in the input). Upon successful completion the program should return `EXIT_SUCCESS`.

Example output:

```
$ cat instructions.txt | ./mstat -i
I-Type  333     69.1%
J-Type  28      5.8%
R-Type  121     25.1%
$ echo $?
0
```

When additionally given the command line flag `-u` the printout will display a header labeling what the columns for the output means. This flag can ONLY be used in conjunction with another flag.

> This is only useful for humans, the `-u` flag should not be provided when giving your output to another program.

Example output of `-i` with `-u`:

```
$ cat instructions.txt | ./mstat -i -u
TYPE    COUNT   PERCENT
I-Type  333     69.1%
J-Type  28      5.8%
R-Type  121     25.1%
$ echo $?
0
```

`-r` Register Information

When given the command line flag `-r` your program should list out the statistics of each register. There is a column for how many times it was used in total, how many R-type instructions used it, how many I-type instructions used it, how many J-type instructions used it, and the percentage of times the register was being used based on all the other registers. Upon successful completion the program it should return `EXIT_SUCCESS`.

Example output:

```
$ cat instructions.txt | ./mstat -r
$0      362     199     163     0       75.1%
$1      180     17      163     0       37.3%
$2      62      15      47      0       12.9%
$3      0       0       0       0       0%
$4      101     17      84      0       21%
... Removed rest of output for brevity
$31     26      15      11      0       5.4%
$ echo $?
0
```

When your program receives the `-u` flag then it should add column headers as the previous section described. Furthermore, you should print out the human readable names of the registers that you are familiar coding with in MIPS.

Example output of `-r` with `-u`:

```
$ cat instructions.txt | ./mstat -r -u
REG         USE    R-TYPE I-TYPE  J-TYPE  PERCENT
$zero       362     199     163     0       75.1%
$at         180     17      163     0       37.3%
$v0         62      15      47      0       12.9%
$v1         0       0       0       0       0%
$a0         101     17      84      0       21%
... Removed rest of output for brevity
$ra         26      15      11      0       5.4%
$ echo $?
0
```

> ℹ The `$zero` register has an inflated number of uses. This is because in certain R-type instructions RS, RT, and RD registers may not be used depending on the function field. The assembler places the zero register in the unused RS, RT, and RD fields of the instruction as a placeholder. By using this register, the assembler is assured there will be no impact on the program, as the zero register is always 0. A "better" implementation may filter out these unused registers, but this **NOT PART OF THIS** assignment. Think of the `-r` statistics more as an occurrence count representing how many times the register has appeared in the instructions.

`-o` Opcode Information

When given the command line flag `-o` your program should list out the statistics of each opcode used. There are 64 (2^6) different opcodes. You should keep track of the number of times an opcode is used and the percentage used in the entire program. For the instructions with opcode `0x00`, also known as R-type instructions, the function field is additionally used to specify the instruction. The function field is 6-bits, enabling 64 (2^6) additional instructions. You should keep track of the function field and its usage percentage with respect to the total number of instructions with opcode `0x00`. If done correctly the percentage of instructions with opcode `0x00` should equal the percentage of R-type instructions you see when you run your program with `-i`. Note the single line break between opcode and function information.

Example output:

```
$ cat instructions.txt | ./mstat -o
0x0     121     25.1%
0x1     20      4.1%
0x2     1       0.2%
0x3     27      5.6%
0x4     16      3.3%
... Removed rest of output for brevity
0x3E    0       0%
0x3F    0       0%

0x0     1       0.8%
0x1     0       0%
0x2     1       0.8%
... Removed rest of output for brevity
0x3F    0       0%
```

Note with the `-u` flag there is a single line break between opcode and function header information.

Example output of `-o` with `-u`:

```
$ cat instructions.txt | ./mstat -o -u
OPCODE  COUNT   PERCENTAGE
0x0     121     25.1%
0x1     20      4.1%
0x2     1       0.2%
0x3     27      5.6%
0x4     16      3.3%
... Removed rest of output for brevity
```

```
0x3F    0       0%

FUNC    COUNT   PERCENTAGE
0x0     1       0.8%
0x1     0       0%
0x2     1       0.8%
... Removed rest of output for brevity
0x3F    0       0%
```

Fail conditions

- Your program must be given one of the arguments `-h`, `-i`, `-r`, or `-o`. If your program is run without any of these arguments it should exit gracefully but with an `EXIT_FAILURE` status. Also, whenever a program is run improperly it usually prints out its *usage* statement, yours should do the same.
- The `-u` argument is for use in tandem with the `-i`, `-r`, or `-o` argument. If `-u` is passed to your program it should format the output into a human readable format, so if `-u` is given without `-i`, `-r`, or `-o` then you should print the *usage* statement and return `EXIT_FAILURE`.
- If your program cannot parse a hexadecimal value in the input file (e.g, `0xzbq3`) then it should return `EXIT_FAILURE` and print the improper hex value to `stderr`.
- This is not an exhaustive list of errors, use good judgement to handle other errors you may encounter. Unless otherwise specified use `EXIT_SUCCESS` or `EXIT_FAILURE` whenever your program exit. If there is a failure, print a descriptive string as to why it failed to `stderr` along with the usage of your program. Essentially your program should never crash it should always exit gracefully.

> ℹ There is an environment variable `?` that holds the exit value of the previously run program. So you can check you're program's exit code by typing `$ echo $?` in the command line.

Combining your program with other UNIX tools

If you make a good Unix tool, then you should be able to use it in tandem with other existing tools. You should become familiar with using some of the following tools. When we grade your assignment we will be combining it with other Unix tools. You should be testing your program this way as well.

Sample program combinations

Assume that your program produces the following output:

```
$ ./mstat -i < instructions.txt
I-Type  333 69.1%
J-Type  28  5.8%
R-Type  121 25.1%
```

Now lets chain this together with the `sort` program. We will sort the rows based on the **numeric** values in the second column.

```
$ ./mstat -i < instructions.txt |  sort -k2n
J-Type  28  5.8%
R-Type  121 25.1%
I-Type  333 69.1%
```

Now let’s use `grep` to search for a value. We will sort the rows based on the **numeric** values in the second column in descending order and then use grep to look for any row that contains the char 2.

```
$ ./mstat -i < instructions.txt |  sort -k2nr | grep 2
J-Type  28  5.8%
R-Type  121 25.1%
```

Now lets use `wc` to count how many results we have:

```
$ ./mstat -i < instructions.txt |  sort -k2nr | grep 2 | wc -l
2
```

> ℹ The programs wc, grep, sort, head , tail are some common tools which are used to help format and examine the output of programs. They usually come installed with most Linux and BSD distributions.
> ⚠ Just because we showed these commands, it does not mean this is the only way we will test your program. This is just a good way to gauge that your program is working correctly but it is **NOT** the end

> all be all. Test your program appropriately.

MIPS Instruction Format Reference

For reference, the MIPS instruction formats are as follows:

R-type:

OPCODE	RS	RT	RD	SHAMT	FUNC
000000	5-BITS	5-BITS	5-BITS	5-BITS	6-BITS

> ℹ In this assignment, RS, RT, and RD field should be counted regardless of the function field.

I-type:

OPCODE	RS	RT	IMMEDIATE
6-BITS	5-BITS	5-BITS	16-BITS

> ℹ In this assignment, opcodes for Immediate instructions are any values other than 0, 2, or 3, which are used for the other instruction formats.

J-type:

OPCODE	ADDRESS
000010	26-BITS
000011	26-BITS

> ℹ Registers can be specified in the RS, RT and RD fields of the instructions. You will need to mask out the values of each of these fields to collect the statistics about register usage.
>
> ℹ A sheet containing all the MIPS reference data has been provided for you on piazza in the

> resources section.

Hand-in instructions

You are expected to hand in at minimum the following files in the hw1 folder of your git submission:

1. All `*.c` and `*.h` files you have created for the assignment
2. Makefile - It must include the following:
    - all target
    - mstat target
        - The `mstat` target must produce a binary called `mstat`
    - clean target
3. README - It must contain the following information:
    - name
    - SBUID #
    - Partners (none for this assignment)
    - Anything relevant to grading your project when errors may occur

Your assignment is expected to work on the following platforms:

1. Ubuntu Desktop x86_64 15.10

Submitting your assignment

REMEMBER Do not submit at the last minute. We will be using the time stamp of the commit associated with the tag to determine if your homework assignment is late or not. On top of that we will **NOT GRADE late assignments**

1. Make sure all files for this assignment are in a `hw1` directory.
2. To tag your submission, make sure first that you have pulled from the remote and all code changes are merged. Once everything is merged, push it back to the remote server. **Be sure you are done making any and all changes before proceeding. Tags cannot be deleted.**
3. Log on to your gitlab account and navigate to the tags page from your repository's main page.

4. Here you can click the green **New Tag** button.
5. Enter `hw1` for the tag name, the name of your current branch (typically `master`) and an optional completion message.
    - **Do not** put important information in the message as we will not necessarily see it. Any relevant information to your submission should be in your `hw1/README`

6. To check if you submitted properly, you should now see a `hw1` tag in the list of tags for your repository.

> ℹ When writing your program try to comment as much as possible. Try to stay consistent with your formatting. It is much easier for your TA and the professor to help you if we can figure out what your code does quickly.