Lab 03

In this lab, we shall work directly from the Khoury Server. The idea is to learn how to work with different tools that are important for our programming in using the C language as well as practice creating arrays using dynamic memory allocation.

The intended learning outcomes are as follows:

1) Understand how to use Make Utility
2) Understand how to use Valgrind for memory management
3) Use pointers
4) Use Dynamic memory allocation to create arrays.

ssh to the Khoury Server and pull the Lab03 pdf from the class git repository

copy the Lab03 folder from the class repository to your working repository folder.

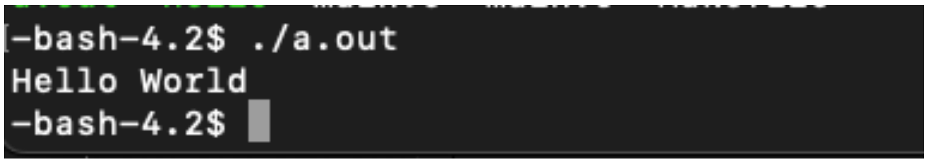
You are now ready to start.

The make utility:

In this portion of the lab, we will begin setting up our environment so that we can quickly compile, clean up, test, and run our lab project. We will begin by making a simple hello world program. Use your favorite editor (vi or emacs or any other) to create and edit a file named "main.c".

```
#include <stdio.h>

int main(int argc, char* argv[])
{
   printf("Hello world!\n");
   return 0;
}
```

We could compile the program from the command line using the command "gcc main.c" which would create an executable named a.out in the current working directory. Do this now and you should be able to execute the file simply by typing its name with the current directory before it as in "./a.out". The resulting output should look like:



```
-bash-4.2$ ./a.out
Hello World
-bash-4.2$
```

Remove the executable file using the command "rm a.out".

For most of the semester we will want to run gcc with the -Wall and --std=c99 options at a minimum. The following command line will create an executable named hello instead of one named a.out.

```
gcc -Wall --std=c99 -o hello main.c
```

We can use a utility program called make to make the compilation process much easier than typing everything out every time. It will especially help as we start moving towards projects that use multi-file compilation.

Use your favorite editor to create and edit a file named "Makefile". Be sure that the name of your file begins with a capital letter M. Edit your Makefile so that it looks like the following:

```
hello: main.c
	gcc -Wall --std=c99 main.c -o hello
```

This forms what the make utility calls a rule. The following is an edited excerpt from the gnu make utility manual:

A simple makefile consists of "rules" with the following shape:

```
target … : prerequisites …
        recipe
        …
        …
```

A *target* is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as '`clean`'

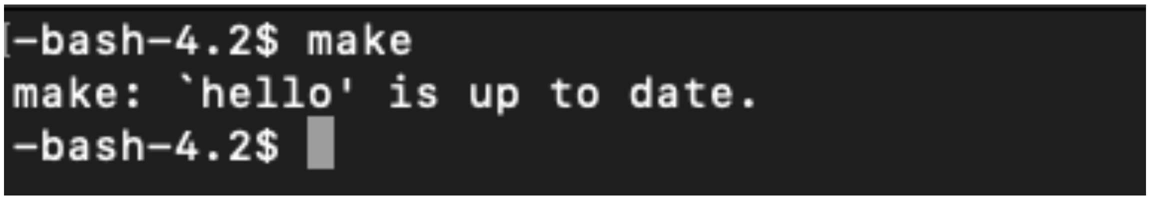
A *prerequisite* is a file that is used as input to create the target. A target often depends on several files.

A *recipe* is an action that `make` carries out. A recipe may have more than one command, either on the same line or each on its own line. **Please note:** you need to put a tab character at the beginning of every recipe line! This is an obscurity that catches the unwary.

Usually a recipe is in a rule with prerequisites and serves to create a target file if any of the prerequisites change. However, the rule that specifies a recipe for the target need not have prerequisites.

If you have created and saved your Makefile correctly you can now type “make” at the command line and it will build the executable named hello which you can run by typing “./hello”.

One of the great features of having a Makefile is that it allows you to only compile the parts of the project that need to be compiled because changes have been made since the last compilation. For large projects this can be a significant savings in time. You can witness this effect by typing make again after you have already built the project and it will tell you that hello is up to date.

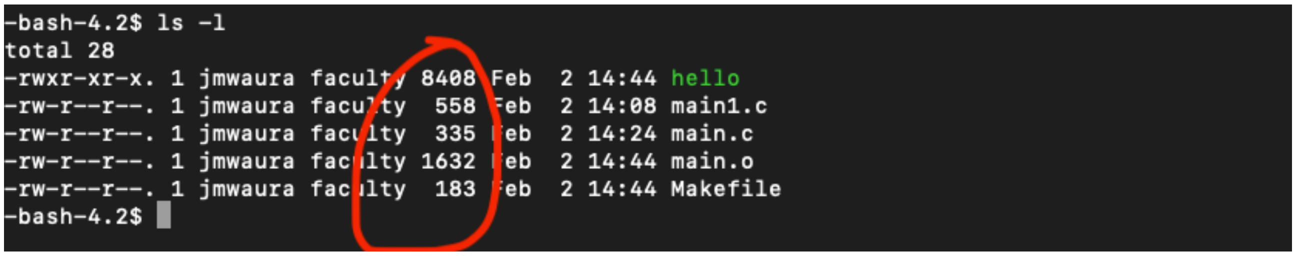


```
-bash-4.2$ make
make: `hello' is up to date.
-bash-4.2$ 
```

Let’s modify the Makefile so that it can split the compilation phase of a file from the linking phase of making an executable. We can do this by asking that the make utility first compile our source code into object code and then make a new rule to turn that object code into an executable. Modify your Makefile so that it looks like the following but remember to use tabs before all of your recipe lines.

```
hello: main.o
	gcc -Wall --std=c99 -o hello main.o
main.o: main.c
	gcc -Wall --std=c99 -c main.c -o main.o
```

The make utitlity, by default, will attempt to make the first target in the Makefile. In our case it will attempt to build the target hello. Since we have changed the rule to say that hello relies on a file named main.o (an object file, not an executable) then it will attempt to build it but since main.o does not yet exist it will look for a rule that describes how to make the target file main.o first. The main difference in the rule logic is that we use the compiler flag -c to tell the compiler that we want to create only object code from the main.c file. The object code for main.c can be compiled independently of all other source files and then linked together at some later time as needed. Type make and give the new Makefile a test run. Once you validate that it is working, let’s clean up the space a little. Right now if I type “ls -l", where the l is a lower case ‘L’, then I get the following output:



```
-bash-4.2$ ls -l
total 28
-rwxr-xr-x. 1 jmwaura faculty 8408 Feb  2 14:44 hello
-rw-r--r--. 1 jmwaura faculty  558 Feb  2 14:08 main1.c
-rw-r--r--. 1 jmwaura faculty  335 Feb  2 14:24 main.c
-rw-r--r--. 1 jmwaura faculty 1632 Feb  2 14:44 main.o
-rw-r--r--. 1 jmwaura faculty  183 Feb  2 14:44 Makefile
-bash-4.2$ 
```

The important information shown is the number in the 5th column. It tells me how many bytes that file is taking up on our file system. Notice that my main.c file is fairly small but the executable, hello, is quite a bit larger and, in practice, can be huge for large projects. If we are not currently testing or running our code then we should do the rest of the users on our system a favor and clean up the files until we need to rebuild. We could do this by typing “rm hello main.o”. This will remove the executable named hello and the object file main.o to save space.

Instead of typing the command manually though each time, let’s make a rule in our Makefile called clean that will allow us to do this any time we want. Modify your Makefile so that it looks like the following:

```
hello: main.o
	gcc -Wall --std=c99 -o hello main.o
main.o: main.c
	gcc -Wall --std=c99 -c main.c -o main.o
clean:
	rm hello main.o
```

Since the make utility will, by default, try to make the first target in the file each time we can ask it to select a specific target by simply naming it. Type “make clean” at the command prompt now and it will automatically remove the appropriate files. You can type make followed by any of your target names and it will try to make just that.

We can use variables in a Makefile to make writing the rules easier. It is common practice to make a variable for your compiler, your compiler flags, and the object files used in your executable. Modify your Makefile so that it looks like the following:

```
CC = gcc
CFLAGS = -Wall --std=c99
OBJECTS = main.o

hello: $(OBJECTS)
	$(CC) $(CFLAGS) -o hello $(OBJECTS)
main.o: main.c
	$(CC) $(CFLAGS) -c main.c -o main.o
clean:
	rm hello $(OBJECTS)
```

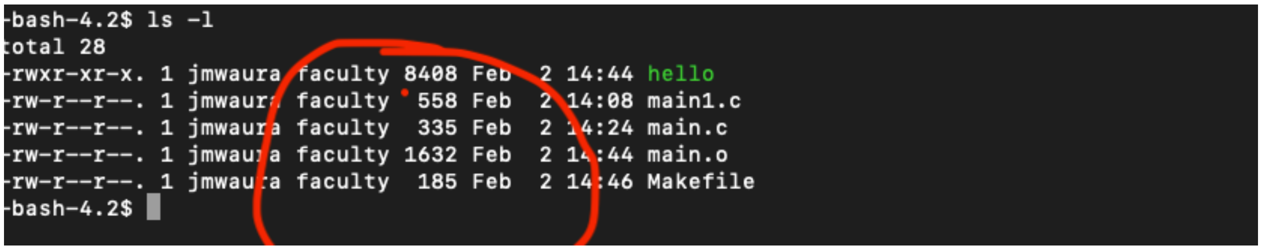
Notice how creating the variable is as simple as giving it a name, an equals sign, and assigning a value. Later in the Makefile you can replace the variable name with its value simply by putting the name in parenthesis and putting a dollar sign in front of it as shown.

CHECKPOINT 1: Demonstrate to your TA the following:

A. Display your Makefile for your TA.
B. Demonstrate that you can build the project using make with no arguments and then run the executable.
C. Demonstrate that you can clean up the directory by typing make clean and then show the directory using “ls -l”.

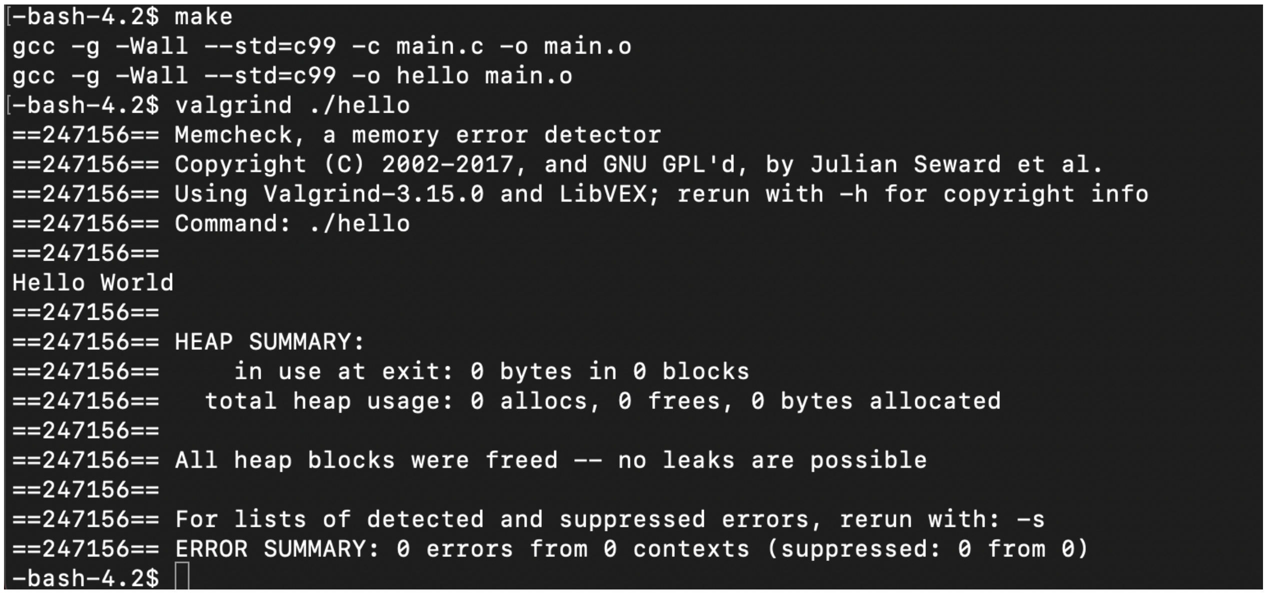
Valgrind:val

We will be using a tool called valgrind throughout the term to help us test for memory leaks and understand memory management better in C. To make use of the tool we have to turn on debugging options for our compiler. We can do this simply by adding the -g option to our CFLAGS variable in our Makefile. Notice that after adding the -g flag to the compiler options that the object code and the executable are slightly bigger.



```
-bash-4.2$ ls -l
total 28
-rwxr-xr-x. 1 jmwaura faculty 8408 Feb  2 14:44 hello
-rw-r--r--. 1 jmwaura faculty  558 Feb  2 14:08 main1.c
-rw-r--r--. 1 jmwaura faculty  335 Feb  2 14:24 main.c
-rw-r--r--. 1 jmwaura faculty 1632 Feb  2 14:44 main.o
-rw-r--r--. 1 jmwaura faculty  185 Feb  2 14:46 Makefile
-bash-4.2$
```

After you build the project using the make utility run it using valgrind by typing “valgrind ./hello”. Your output should look something like the following:

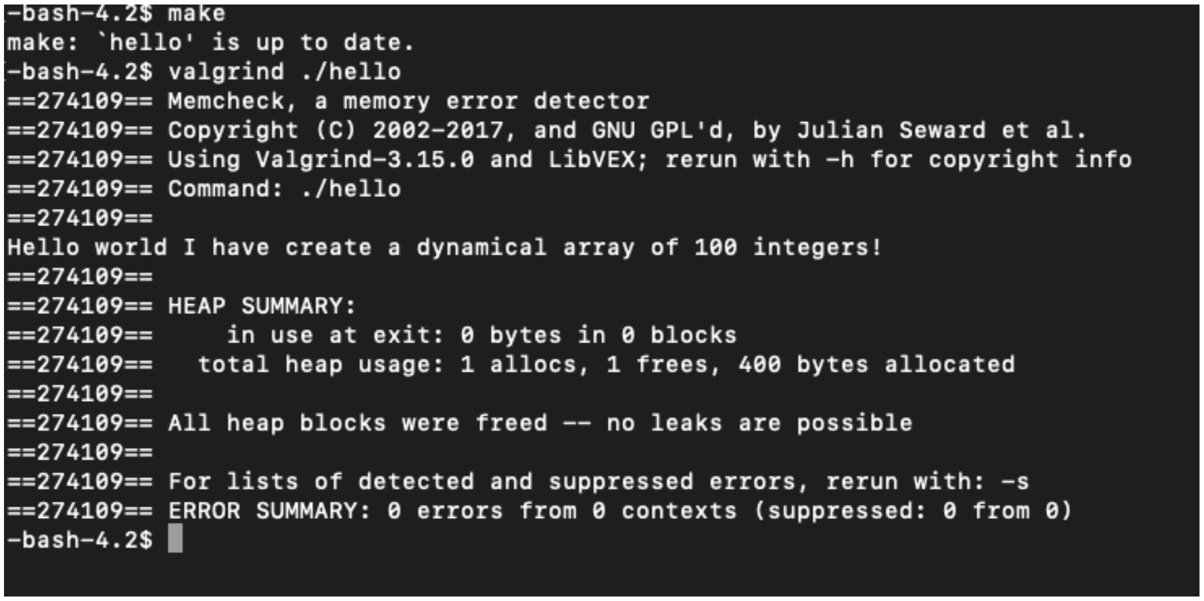


```
-bash-4.2$ make
gcc -g -Wall --std=c99 -c main.c -o main.o
gcc -g -Wall --std=c99 -o hello main.o
-bash-4.2$ valgrind ./hello
==247156== Memcheck, a memory error detector
==247156== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==247156== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==247156== Command: ./hello
==247156==
Hello World
==247156==
==247156== HEAP SUMMARY:
==247156==     in use at exit: 0 bytes in 0 blocks
==247156==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==247156==
==247156== All heap blocks were freed -- no leaks are possible
==247156==
==247156== For lists of detected and suppressed errors, rerun with: -s
==247156== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
-bash-4.2$
```

The important part of the output comes after the HEAP SUMMARY: You will note that no heap space was in use at the time of exit and the program did not use any heap at all. The most important line is the one that says --no leaks are possible. This is your mantra for the entire semester. You want all of your programs to be able to run and give you the result that no leaks are possible. Valgrind will be your friend. Use it. Use it often. Let's try to write a program with an intentional memory leak to demonstrate. Modify your main.c file so that it looks like the following:

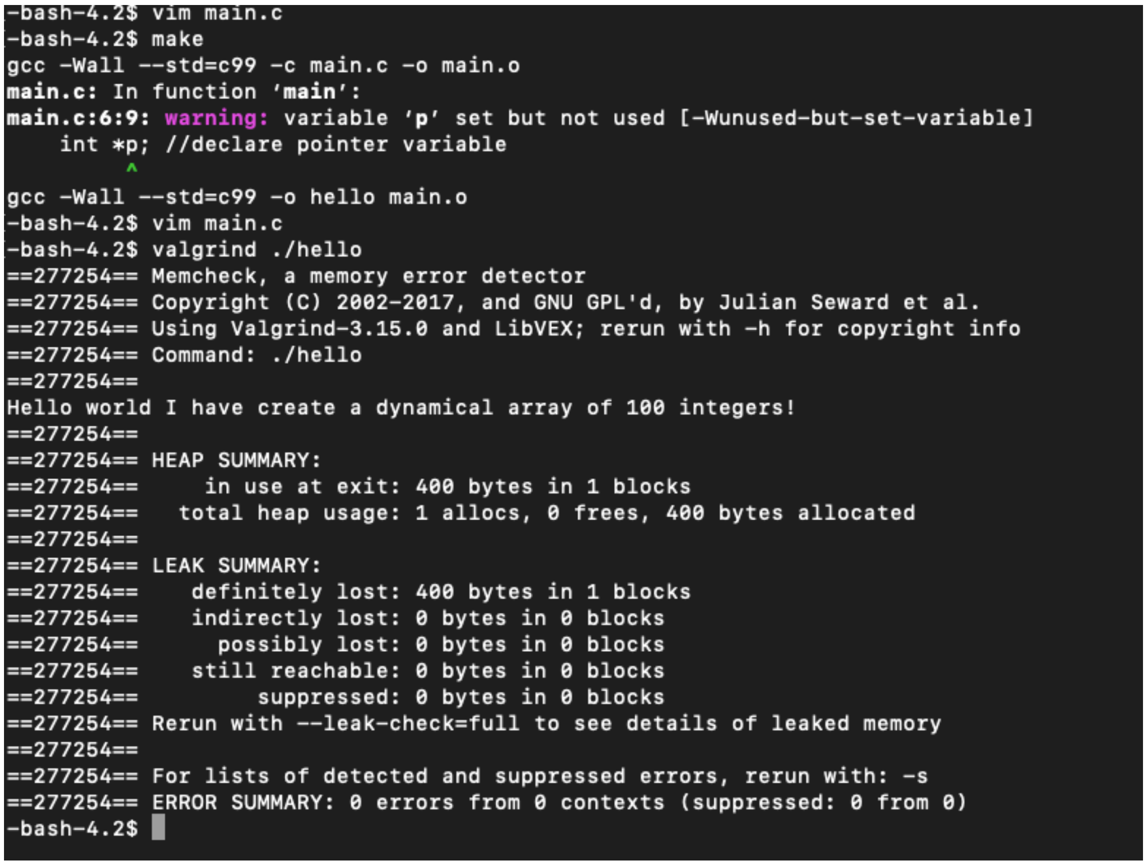
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
  int *p; //Declaration of a pointer variable.

  p = (int*) malloc(sizeof(int)*100clear //First "bookend" allocates space
  printf("Hello world! I have created a dynamic array of 100 integers!\n");

  free(p); //Second "bookend" cleans up the space
  return 0;
}
```

Use the make utility to build your project using the command make and then use valgrind to test it by typing "valgrind ./hello". Your output should look something like the following:



```
-bash-4.2$ make
make: `hello' is up to date.
-bash-4.2$ valgrind ./hello
==274109== Memcheck, a memory error detector
==274109== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==274109== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==274109== Command: ./hello
==274109==
Hello world I have create a dynamical array of 100 integers!
==274109==
==274109== HEAP SUMMARY:
==274109==     in use at exit: 0 bytes in 0 blocks
==274109==   total heap usage: 1 allocs, 1 frees, 400 bytes allocated
==274109==
==274109== All heap blocks were freed -- no leaks are possible
==274109==
==274109== For lists of detected and suppressed errors, rerun with: -s
==274109== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
-bash-4.2$
```

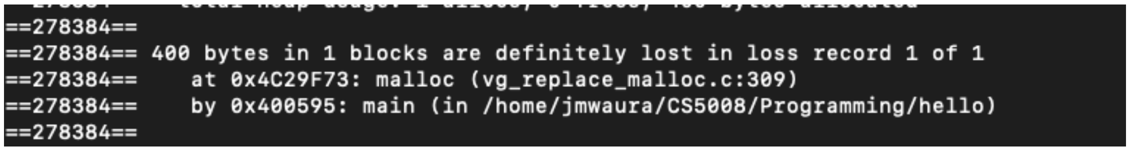
Now comment out the free(p); line in your main.c program and run valgrind again. Your result should now look something like the following:



```
-bash-4.2$ vim main.c
-bash-4.2$ make
gcc -Wall --std=c99 -c main.c -o main.o
main.c: In function ‘main’:
main.c:6:9: warning: variable ‘p’ set but not used [-Wunused-but-set-variable]
    int *p; //declare pointer variable
         ^
gcc -Wall --std=c99 -o hello main.o
-bash-4.2$ vim main.c
-bash-4.2$ valgrind ./hello
==277254== Memcheck, a memory error detector
==277254== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==277254== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==277254== Command: ./hello
==277254==
Hello world I have create a dynamical array of 100 integers!
==277254==
==277254== HEAP SUMMARY:
==277254==     in use at exit: 400 bytes in 1 blocks
==277254==   total heap usage: 1 allocs, 0 frees, 400 bytes allocated
==277254==
==277254== LEAK SUMMARY:
==277254==    definitely lost: 400 bytes in 1 blocks
==277254==    indirectly lost: 0 bytes in 0 blocks
==277254==      possibly lost: 0 bytes in 0 blocks
==277254==    still reachable: 0 bytes in 0 blocks
==277254==         suppressed: 0 bytes in 0 blocks
==277254== Rerun with --leak-check=full to see details of leaked memory
==277254==
==277254== For lists of detected and suppressed errors, rerun with: -s
==277254== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
-bash-4.2$
```

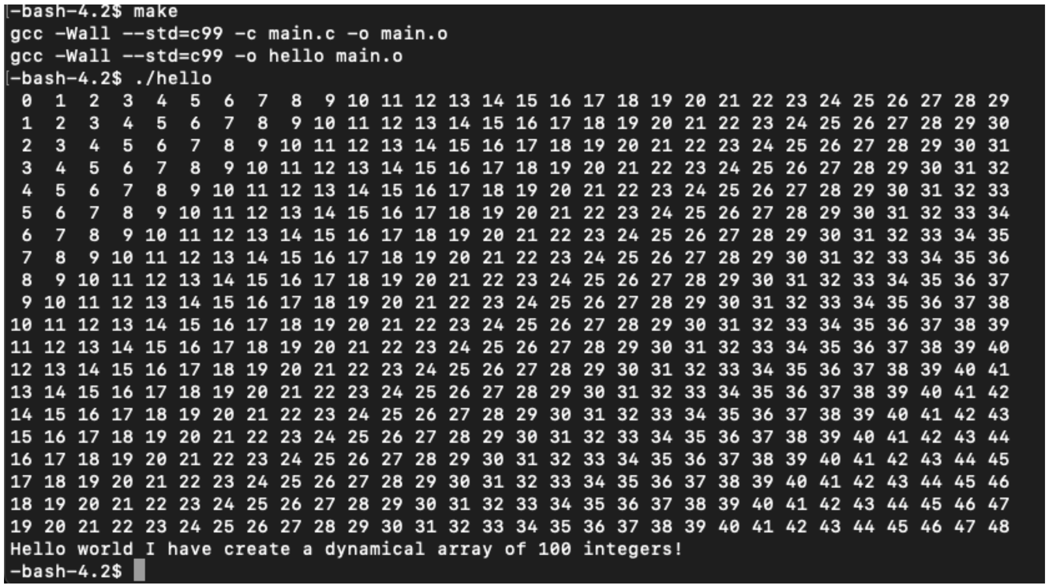
This shows me that there is a loss but it is not clear where the loss is. Sometimes the size of the loss can help you determine what it is and in our case we only have one allocation and 0 frees meaning we didn't try to free it at all but you can get even more information by simply following the instructions to rerun valgrind with the --leak-check=full option. You can do this by typing "valgrind --leak-check=full ./hello".

I just want to draw your attention to these lines:



```
==278384==
==278384== 400 bytes in 1 blocks are definitely lost in loss record 1 of 1
==278384==    at 0x4C29F73: malloc (vg_replace_malloc.c:309)
==278384==    by 0x400595: main (in /home/jmwaura/CS5008/Programming/hello)
==278384==
```

Modify your main.c file so that it allocates a two dimentional array of integers so that the array has 20 rows of 30 integers in each row. Fill up the entire array so that the first row contains 0-29 the second row contains 1-30, the third row contains 2-31 and so on until the last row contains 19-48. Print the results in a grid so that your output looks like:



```
[-bash-4.2$ make
gcc -Wall --std=c99 -c main.c -o main.o
gcc -Wall --std=c99 -o hello main.o
[-bash-4.2$ ./hello
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38
 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
Hello world I have create a dynamical array of 100 integers!
-bash-4.2$
```

CHECKPOINT 2 (50 points): Demonstrate to your TA the following:

Please note to get all the 50 points, you need to make sure that the 2D array; the entire array must be made using dynamic memory allocation. Also, the free function should demonstrate a clear understanding of how dynamic allocation was done.

A. Run your program and show the appropriate output.
B. Show your program to your TA so that they can see you have a dynamic two-dimensional array using malloc.
C. Demonstrate, using valgrind, that your program has no memory leaks. (No leaks are possible)

Submission: Push all your files to your Github repo (10 points)