THE UNIVERSITY *of* NEW ORLEANS

DEPARTMENT OF COMPUTER SCIENCE

CSCI 2467, Spring 2020
Lab 0: Introductions to C and Unix
Due: Wednesday, January 22, 11:59PM

2467 Instructors: M. Toups & C. Boyce
Assistants: D. McDonald & S. Duwal
staff@2467.cs.uno.edu

1 Introductions

The purpose of this assignment is to perform two introductions while getting you started in your exploration of Computers from a Systems perspective. Chapter 1 of Bryant & O'Hallaron's CS:APP textbook will accompany this lab conceptually, and the tools we use will be explained further in the two PDF documents from Stanford's CS Library (*Essential C* and *Unix Programming Tools*).

Please keep both the textbook and these two supplements handy as we work through this lab. The two introductions we will accomplish are:

1.1 Introducing you to the lab tools

This lab will introduce you to the tools and processes we will use for all labs this semester. This includes how to access the initial files to work on and how to turn in your work using our AutoLab system.

It is *very important* that you become comfortable with the process of starting, working on, and turning in a lab using the instructions given here. The labs after this one will be much more challenging, so you cannot afford to get held up on logistical details with those labs. Use this opportunity to learn the system with this lab, which mainly requires you to follow directions.

1.2 Introducing us to the students

Your instructor and course staff want to know a little bit about each of the students in the course. Rather than pass out a survey or ask you to introduce yourselves in class, instead you are given the opportunity to introduce yourself while completing this simple lab assignment.

2 Pre-requisites

2.1 What you will hand in

This is an individual lab assignment. You must turn in only your own work, including comments. We will discuss much of this in class, but it is your responsibility to work on and complete the lab on your own as homework. You may do your work in the lab (Math 209/212) or remotely using `ssh`.

The instructions in this (and all subsequent) assignments assume that you are logged in to our class server. You will either open a terminal in room 209/212 and type commands, or use the web client at `https://systems-lab-web.cs.uno.edu`, or use ssh.

You will hand in your work electronically using AutoLab when you complete the steps given in this document. When you are finished, you will create a `tar`[1] file containing your work to submit which should contain the following files (including the bonus answers *if* you choose to attempt them):

```
part1/part1.c
part1/student.c
part1/student.h

part2/answers2.txt
part2/a.out
part2/bonus-answers2.txt
part2/hello.c
part2/hello.i
part2/hello.s
part2/hello
part2/hello-att.s
part2/hello-bits
part2/hello-xxd

part3/answers3.txt
part3/bonus-answers3.txt
part3/copyij_result.txt
part3/copyji_result.txt
```

2.2 Get the initial *handout files* and set up a working directory

The lab files are all contained in a `tar` file. You will download a copy of the `tar` file and then extract its contents in your home directory.

Each user has their own space on the system for files, called a *home directory*. The path for the directory is `/home/<USERID>` where USERID is your UNO user ID. When you first log in and open a terminal, your home directory is the default starting point. The files on your home directory are accessible from all of the thin clients in rooms 209 and 212, as well as via ssh, so you should always be able to access them inside or outside the lab.

[1]This originally stood for *(t)ape (ar)chive* but the name persists decades later. `tar` is the most common archive format on Unix systems, analogous to `zip` files on MS Windows systems.

2.2.1 Creating your directory

We will make a *protected working directory* to use for all class assignments in your home directory, using the following commands:

```
  $ cd ~
  $ mkdir 2467
  $ chmod go-rwx 2467
```

Above, the first line changes our *current directory* to your home directory. Normally when you first log in this will already be the current directory, but in case it isn't, it doesn't hurt to do this. Then the second line creates a directory called `2467` for your lab work.

The third line is important. This changes the permissions on the directory, ensuring that no other users (such as your classmates) can read your files. `chmod` is the Unix command for changing permissions. The characters `go-rwx` tell it to remove (`-`) the read (`r`), write (`w`), and execute (`x`) bits from both group (`g`) and other (`o`) users. We will describe these things more in class, but it is important that your working directory is fully private.

Verify that the permissions look right by using the command `ls -ld 2467`. It should look like this (only the names and dates should be different):

```
drwx------ 2 matoups2 matoups2 4096 Aug 17 12:13 2467
```

2.2.2 Downloading `introlab-handout.tar` from autolab

Now you can get the assignment files by downloading the file `introlab-handout.tar` from Autolab. In a web browser, navigate to `autolab.cs.uno.edu`, select the "Sign in with UNO" option near the bottom of the page and sign in with your UNO student credentials. Select the 2467 course, then **Intro Lab** from the list of assignments. Under 'Options', click on 'Download handout' to download `introlab-handout.tar`. Move it from `Downloads` to your `2467` directory. Then extract the `tar` archive using the tar command. See the example below:

```
  $ cd ~/2467
  $ tar -xvf introlab-handout.tar
```

The first line changes your current directory, so you are now "in" the `2467` directory where `introlab-handout.tar` was downloaded (the tilde symbol (~) is an abbreviation for your home directory, so it is equivalent to `/home/<UserID>`).

Then the second line extracts the files, using the tar command. This will cause a number of files to be unpacked.

The letters `xvf` are given as an argument to the `tar` program.[2] In fact, each letter denotes something specific we are telling the program. `x` is the main operation, short for `extract`, as opposed to `create` which does the opposite. `v` is an option, short for `verbose`, which says that we want to see a list of each file extracted. Without this option, `tar` will only report errors, which can be confusing if you

[2] The dash before `xvf` is actually optional, but we have included it here to make it clear that those letters are not the filename being operated on.

don't know what files to expect. Finally, `f` says we are going to extract from a file, and is followed by the filename to operate on.

Warning: The `tar` command can be powerful and unforgiving. It does not ask you "are you sure?" before doing what you tell it to do. If you have already extracted these files, if you tell it to extract again it will happily do so **and overwrite any changes you may have made!** Once that happens, any changes you have made will be lost and no one can recover the file.

Be **very careful** in the future with the `tar` command. You will need to use it throughout the semester, but be aware that it can overwrite files you may have already worked on. If you are in doubt, you can always create a temporary directory and extract the files there to be sure it doesn't overwrite anything or do something else you aren't expecting.

3 Lab manual

Now that you have your initial handout files and working directory set up you can proceed with the lab. The following sections will be graded with points given for each section completed.

Part 1 (20 points): Who are you?

In this section, you will simply introduce yourself to me and tell me a little about yourself. I don't want you to provide more personal information than you're comfortable with, so for this section there will be *no wrong answers*. You can answer with as much detail as you like, the idea is for me to get an idea of who you are, both personally and in terms of your computer science background.

In addition to introducing yourself, you will also get familiar with creating a very simple C program.

As you read the *Essential C* document from Stanford CS Library, you should notice that many of the things described in section 1 and 2 seem similar to Java. We will assume you already know how to use control structures like *if* statements, *for* loops, etc and you know the difference between simple datatypes like *int* and a *float*.

Section 3 of *Essential C* and Chapter 6 of *K&R* describe something that does not exist in Java: a *struct*. This term is short for a *structure* [3], a complex data type which can contain various data types within it.[4] You will be using a *struct* to store various types of information about yourself.

Big hint: To help you out, and because dealing with strings in C is notoriously awkward, I have provided you with some example code to start from. Look in the `part1` directory that you extracted in the previous section.

You can complete this part by editing the following files:

- `part1.c` This contains the `main()` function, the entry point for all C programs.
- `student.c` This contains the `make_student()` function, described below.
- `student.h` This contains the definition of the `struct student` datatype. This filename ends in `.h`, denoting a *header* file, because it only contains a struct definition but no functions.

[3] Some languages may refer to this kind of heterogeneous data type as a *record*.

[4] This may remind you of a *Class* in Java, but a C *struct* is much simpler, without the notion of methods or members being *public/private*.

You will edit each of these files using a text editor, for example the lightweight `nano` or `vim`, or the more modern and complex `notepad++` or `atom` or `sublime_text`. You may choose whatever editor you are comfortable with, feel free to try a few, they have advantages and disadvantages. Regardless, you will need to be comfortable with at least one (preferably more) of these tools in your career as a student and especially professionally.

If you are using nano, your command will look like: (for another editor, replace `nano` with `vim, atom, sublime_text`, etc)

```
$ nano student.h
```

Once you've made edits, save the file (in `nano` use `CTRL-O` [5] and then exit with `CTRL-X`). Any time you make changes you should compile the code and make sure it does not contain errors.

How to compile the program: Use the following command to compile using `gcc`[6]

```
$ gcc part1.c student.c -o part1
```

How to run the program: (Note the use of `./` before the program name)

```
$ ./part1
```

Lab task (10 points): Make a struct with appropriate datatypes to contain the following pieces of information about you. Then, write a function to populate an instance of that struct with the relevant information. (Both of these things are already started in the sample files.)

Most of this information will be stored in either `char` arrays or `ints`. [7]

First we must *define* the struct as a datatype. This has been started in `student.h` but you must complete it by defining more fields in the struct.

Here is some information I would like you to provide:

- Student ID number (from your ID card)
- Student Name
- *(optional)* Name you prefer, if different
- UNO email address
- *(optional)* preferred email address
- *(optional)* City or place of birth/origin (or where you consider your hometown to be)
- Number of semesters you have studied at UNO prior to this semester (0=this is your first semester here)
- CS Classes taken (course numbers, as an array of `ints`, see the example code)

[5] Often the control key is denoted with the carrot, so ^O is another way of writing CTRL-O

[6] Described in Section 1 of *Unix programming tools.*

[7] If you can think of ways to use other C datatypes, feel free to be creative, but be aware that strings and pointers in C can quickly become difficult. Try not to get in over your head, at least not until you've finished the easy parts.

- What you most want to get out of this class
- Why you are studying Computer Science
- *(optional)* Describe the most complicated or difficult program you have written
- *(optional)* Describe something other than programming you've learned in CS
- Describe a non-CS interest of yours (such as music, art, literature, cuisine, sports, movies, etc)
- *(optional)* Anything else you think I should know about you

In `student.h` you are only defining the *types* for these fields within the `student_info` struct. Once the struct is defined, add the actual information within the `make_student()` function in `student.c`[8].

Be sure the arrays you create in `student.h` are large enough to hold the values you are putting in them. You can always increase the size if you decide to write more than you thought. Refer to *Essential C* and *K&R* to make sure you are using array notation properly.

After you define the types in `student.h`, you will actually set values for an instance of the struct in `student.c`. This is where you'll actually type up the information requested above.

Lab task (10 points): Modify the `main()` function to print out your struct in a nice, readable fashion with each member of the struct on its own line, with a heading. This will require use of the library function `printf`, which is described on page 42 of *Essential C* or section 7.2 of *K&R*.

So your program output should look similar to this:

```
$ ./part1
ID number: 22222222

Name: My Name

...

Why I am studying Computer Science:
All the cool kids are doing it.

...
```

You can see a starting point on how to do this in `part1.c`. Using the example code, finishing this program should be quite easy.

Extra credit (1 point): Create a file called `Makefile` so that we can compile this program by simply typing `make`. See Section 2 of *Unix Programming Tools* for more information on makefiles. It can be a simple makefile, you do not need to worry about `CFLAGS` and other various options. Be warned that the makefile format has this annoying feature where it requires you to use tabs instead of whitespace. If you attempt this part, add it to your handin (as described below).

Be sure any code you turn in will compile using the `gcc` line given above!

[8]Pay attention to the difference between `student.c` and `student.h`. Having two files with the same name and different endings is a common convention in C programming.

If `gcc` returns with no output, that means there were no errors during compilation. Run your program one more time (as described above) to make sure it looks right. If there are any problems you can fix them and turn in again.

Thank you for introducing yourself in a C program!

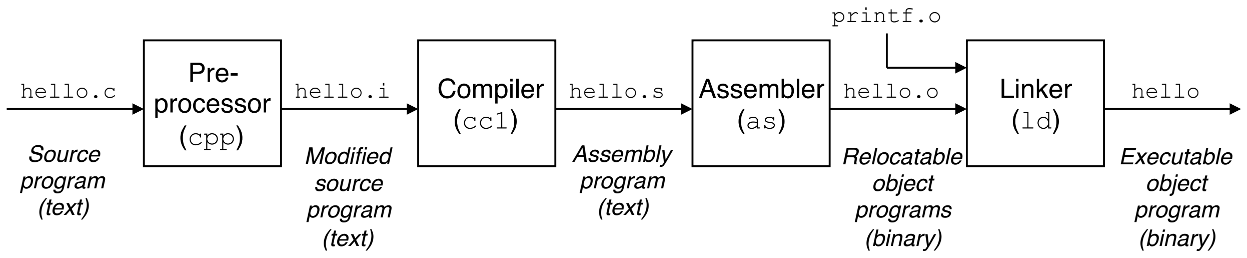
Part 2 (10 points): The lifecycle of a C program

In this section we will trace the life cycle of the famous first C program called `hello.c`. This program prints the words `hello, world` and exits. You have probably seen similar programs written in other languages; this program has become the classic example which most texts use as a starting point.

What you will hand in:

- `answers2.txt bonus-answers2.txt` containing your answers to questions in this section.
- `a.out hello.c hello.i hello.s hello hello-att.s hello-bits hello-xxd` files as described below

Recall **Figure 1.3** from CS:APP:



This is the process we will trace, taking the source code of `hello.c` and passing it through the pre-processor, compiler, assembler, and linker to yield an executable program. The exact steps are given for you below, please follow them. Your results will be turned in, along with your comments.

Step 1: The hello program

We are going to work with a modified version of the classic `hello.c`, which you can see in CS:APP **Figure 1.1** or in the original source by K&R. This file is provided in your handout files, in the `part2` directory.

The change in this version is that we use a different return value for the `main()` function. This will not have any particular meaning or change the behavior of the program [9], but it will allow us to see a common use of the preprocessor and to see that value in the assembly code produced by the compiler.

First, read the file `hello.c` using cat or nano.

Note that `RETURNVALUE` is *not* a variable in the program. This is what is called a *macro* defined by the preprocessor, and as we will see soon, it will be replaced by the value 67 before the C compiler

[9] It will actually matter in an insignificant way. The program will return 67 to the shell as its *exit status*, which is typically zero in the case of success, or a nonzero value can specify an error type. Our use of 67 breaks with that convention, but there is no harm in doing that here.

ever sees the source code. These are traditionally (but not necessarily) rendered in all capital letters to avoid confusion with “normal” variables.

Step 2: Examine the source code file

Figure 1.2 in CS:APP contains an ASCII representation of `hello.c`. ASCII[10] is a very simple mapping between numeric values and the alphabet used in English text, as well as some digits and symbols. Each character can be represented in 1 byte[11] and we can also display the contents of that byte using a number.

Let’s look at how this program is stored in bits by the computer. Use this command:

```
$ xxd -b hello.c
```

You should see the program text on the right; to the left of each line is a view of how the file looks with each bit written out. Each group of eight bits is separated by a space, denoting a byte. The first character of your program should be `#` which will be represented by `00100011`. You will notice that the first (most-significant) bit of each byte it zero, because ASCII only uses 7 bits to encode 128 possible characters. This limitation is why ASCII is insufficient for encoding characters other than the basic Latin alphabet. [12]

Save this output as `hello-bits`. This will be part of your handin. How should you do this? You can cut and paste into your text editor of choice, or use the shell redirect operator like this:

```
$ xxd -b hello.c > hello-bits
```

The `>` character takes the output of the `xxd` command and writes it to a file. You will use this later.

Regardless of what method you use, ensure you end up with a file called `hello-bits` which contains the output of `xxd`.

Clearly a file larger than this would quickly become unusable when viewed this way. Here’s another way to view the file in a more compact fashion:

```
xxd hello.c
```

Now we see each byte represented by two hexadecimal characters. The first character `#` is now represented by `23`, the next (letter `i`) is represented by `69`, the next (letter `n`) is represented by `6e`, and so on. You can see how this is a much more efficient way to display the data in the file.

In both of the above cases, the leftmost column is an *offset* which allows you to refer to a byte’s position within the file. Both start at 0000000, but the bits representation can only fit six characters per line so the offset grows more slowly.

Save this output as `hello-xxd`. This will also be part of your handin.

[10]This stands for American Standard Code for Information Interchange, dating back to the 1960s

[11]Actually, 7 bits; the most significant bit of each ASCII byte will be zero.

[12]Unicode is a far superior way to encode text, as it supports virtually all human languages and alphabets. See the aside on page 50 of the CS:APP book for more on Unicode. Fortunately, Unicode is backward-compatible with ASCII, so ASCII values are also correct Unicode values.

Extra credit question (2 points): The CS:APP textbook shows `hello.c` in **Figure 1.2**, but the values are not the same as what we got from `xxd`. For example, the `#` character is represented by `35` in the book, but `23` in our file. Why aren't these the same? (Write your answer in the file `bonus-answers2.txt` to hand in.)

Step 3: The C Preprocessor

Now we can begin the process of building an executable program from this source code. The exact steps are given for you below, please follow them. Your results will be turned in, along with your comments.

```
$ cpp hello.c hello.i
```

The file `hello.i` has been created with the output of the C preprocessor `cpp`. Only lines that begin with # are interpreted by `cpp`. Look at both files using `ls -l`.

The owners and permissions on both files should be the same, but `hello.i` should be much larger than `hello.c`.

Homework Question: What are the differences between `hello.c` and `hello.i`? Put your answers in the file `answers2.txt` to hand in.

Step 4: The Compiler

Now the compiler will convert C source code into assembly language instructions.

```
$ cc1 hello.i hello.s
```

Take a look at `hello.s` now. It should look somewhat familiar, but something is different from what you saw in your assembly language class. By default, the GNU compiler emits what we call *AT&T format* assembly code, which is common on UNIX systems. On Microsoft platforms it is more common to see *Intel format* assembly, which is probably what you saw in previous courses. The two formats are equivalent and neither is technically superior, but many people feel strongly about their preference, similar to big-endian and little-endian byte ordering.

In order to get the Intel-style assembly language syntax, we'll invoke the `cc1` command differently. Let's compare the differences.

First move[13] (a.k.a. rename) the *AT&T-style* assembly file `hello.s` to `hello-att.s`

```
$ mv hello.s hello-att.s
```

then compile again using the flag `-masm=intel` as shown below.

```
$ cc1 hello.i hello.s -masm=intel
```

[13]The `mv` command is similar to the `cp` command in that its arguments are *source* followed by *destination*.

Lab Task: Add comments to the assembly language code by using the # character followed by your comment. (On any single line of text, everything after the # symbol will be considered a comment) Add a comment on the correct line for each of the following instructions:

- indicate where the space for `int retval` is stored
- indicate where the value is placed in `retval`
- indicate where `printf` is called
- indicate where the function actually returns

Save your commented version as `hello.s`. You may choose which assembly language style you prefer, Intel or AT&T, and comment that one only. Remember that you cannot use C or Java style comments in this file (no `/* */` or `//` comments), and that the # character denotes a single-line comment.

Step 5: The Assembler

Now we turn the text file containing assembly instructions into a binary object file using the assembler:

```
$ as hello.s
```

That should produce a file called `a.out`. This is a binary *object file* and cannot be read in a text editor anymore, but you can examine it by running `xxd a.out`. You will see a few strings inside the program. The other stuff consists of machine instructions in the form of *opcodes*.

Step 6: The Linker

This is one of the more obscure and complicated parts of this process. Your CS:APP book devotes all of Chapter 7 to the topic of Linking. We do not plan on studying that chapter in detail for this semester, but we will need to know the basics of what the linker does.

The `a.out` file contains the binary opcodes for running the hello program, but it does not yet have the code for accessing external library functions. In this case, that means `printf` and `strcpy`. The linker makes it possible to use those library functions.

Invoking the linker is almost never done by hand, but it is the way we turn that `a.out` file into something that runs. (This would normally be the last of many steps `gcc` runs for you.) The command line for doing this is really long, most of it necessary to make `printf` (actually `puts`) work.

Run this command exactly as written. You may want to use the PDF version of this lab manual to copy the text, but be careful when copying and pasting, you may need to do it a little at a time because it must be run as one long line):

```
$ ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 -o hello a.out
/usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o
/usr/lib/x86_64-linux-gnu/crtn.o /usr/lib/gcc/x86_64-linux-gnu/7/crtbegin.o
/usr/lib/gcc/x86_64-linux-gnu/7/crtend.o -lc
```

Now you know why nobody wants to run the linker by hand!

At last, you should have an executable file called `hello` – did you have any idea that making a "hello, world" program could be so complicated? This is what really happens when your program gets compiled (we just don't normally see each step along the way).

You should now be able to run the program and see the output!

```
$ ./hello
Hello world.
```

That's what we should see! You'll need to use `./` when running the program. [14]

Step 7: A quick sanity check of your files

Here's a sanity check you can use to make sure your files are what we expect them to be. You can use the Unix command `file` to examine your various `hello` files and determine their type.

```
$ file *
a.out:        ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
hello:        ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.26, not stripped
hello-att.s: assembler source, ASCII text
hello.c:      C source, ASCII text
hello.i:      C source, ASCII text
hello.s:      assembler source, ASCII text
```

If your files have the correct types, that is a good sign.

Lab task (10 points): Make sure the following files you created while going through the steps above are properly named: `answers2.txt a.out bonus-answers2.txt hello hello.c hello.i hello.s`. You will receive points based on what you've written in `answers2.txt`, your comments on the assembly code in `hello.s`, and if your hello program still works! (Please make sure it does)

Part 3 (10 points): How long does it take to copy an array?

In this part we will find a surprising result in a very simple program which will illustrate what we've read in Chapter 1 of CS:APP, specifically the section *Caches Matter* and the figure labelled *The memory hierarchy.*

We will go through this process in class, but you will also need to run this program yourself and record your own results. Results may differ slightly but the same trends should be clear.

In the `part3` directory you extracted from the initial tar file you will find two versions of a C program which are only slightly different: `array_copy_ij.c` and `array_copy_ji.c`. Following the steps below, you will change into that directory, view the source code (using the `cat` command), and then compile the program. [15]

[14] This tells the shell you are running something in the current directory, represented by the dot (.)

[15] It turns out in the very simple case where we have only one file of source code, `make` works without a `Makefile` and saves us a little bit of typing compared to running `gcc`.

```
$ cd part3
$ cat array_copy_ij.c
$ make array_copy_ij
$ cat array_copy_ji.c
$ make array_copy_ji
```

These two programs do exactly the same thing, but they only differ in that one uses a function called `copyij` and the other uses `copyji` (the order of i and j is reversed). Here are the two functions, as you see the only difference is that the nested `for` loops are in a different order:

```
void copyij (int src[2048][2048], int dst[2048][2048])
{
  int i, j;
  for (i = 0; i < 2048; i++)
    for (j = 0; j < 2048; j++)
      dst[i][j] = src[i][j];
}

void copyji (int src[2048][2048], int dst[2048][2048])
{
  int i, j;
  for (j = 0; j < 2048; j++)
    for (i = 0; i < 2048; i++)
      dst[i][j] = src[i][j];
}
```

What do these functions do? They appear to do exactly the same thing, and in fact they produce identical results. So why should it matter which one we use?

We want to measure how fast this program runs, so we will use the `time` command which should look like this:

```
$ time ./array_copy_ij
Running copyij... done!

real    0m0.053s
user    0m0.034s
sys     0m0.015s
```

This should print out three different times. We will pay attention to the `real` time. In the example above, the program took 0.053 seconds. Run it a few more times. The results should change a little, but be similar.

Let's record this result to a file. The easiest way to do this is using the shell's redirection operator, described at the very end of *Unix Programming Tools*, to write a file `copyij_result.txt`. Unfortunately due to obscure reasons, we have to wrap the command in parenthesis and use some other weird syntax, run it like this:

```
$ (time ./array_copy_ij) > copyij_result.txt 2>&1
```

Now the time is saved in `copyij_result.txt`. Run: `cat copyij_result.txt` to make sure it really contains the results.

Now run `time ./array_copy_ji` as you did above. Has the result changed? It should. Record this result in a new file using a similar command as above, instead calling this file `copyji_result.txt` (note the order of i and j changed in the filename)

Based on your reading of Chapter 1 and our discussion in class, you should be able to describe briefly why these seemingly identical functions would run faster or slower. Put your answer in `answers3.txt`.

Lab task (10 points): Make sure the files `copyij_result.txt` `copyji_result.txt` and `answers3.txt` are named correctly so that you get credit for completing this part.

Extra credit (1 point): Use `gcc` to compile both programs with the `-O3` flag[16] and compare the results. Does it run any faster? What is the difference between the two functions now? Why? What's going on here? Add your answer to `bonus-answers3.txt`.

Wrap-up

Now it's time to create the `introlab-handin.tar` file that is to be submitted to Autolab. To create the `tar` file we must first be sure that our current working directory contains the directories `part1 part2 part3`. Follow the steps below:

```
$ cd
$ cd 2467
$ ls
part1 part2 part3
$ tar cvf introlab-handin.tar part1 part2 part3
```

The first line moves us back to our home directory. We then enter the `2467` directory with the second line. The third line is to ensure that we are in the right location and can see our `part1 part2 part3` directories. Finally, the last line creates `introlab-handin.tar` which we will submit to Autolab.

To submit `introlab-handin.tar`, go back to where the lab handout was downloaded from Autolab. On the right hand side, check the box that confirms that you have adhered to the academic integrity policy then click the submit button. This will open up a file upload window where you will select the `introlab-handin.tar` file you just created. Refresh the page after a few seconds and you will see that the autograder has graded your work. You can see detailed grading information by clicking on one of the highlighted scores for parts 1, 2, or 3. Keep in mind that if you are unhappy with your score, there are unlimited submission attempts as long as the assignment is turned in before the deadline.

You've completed Lab 0! Now on to the fun stuff...

[16]Note that this is the letter capital-o, not the digit 0. But it is followed by the digit 3!