Lab 2: Introduction to C Programming

CSCI 2122 - Winter 2022

1 Introduction

This lab is your first introduction to C programming and will introduce you to many of the basic concepts of C and low-level coding. By the end of this lab, you will be expecting to understand the follow key points:

1. Understanding C libraries.
2. Understanding functions in C.
3. Understanding arithmetic in C.
4. Printing text to stdout in C.
5. Reading text from stdin in C.
6. Creating C Source files (.c) and Header files (.h).
7. Compiling C code.

The proceeding labs will build and expand this knowledge into the realm of low level computing, including memory management, data structures, methods for approaching low-level computation, and a variety of other subjects. As such, knowing these key principles will be important for future programming and development.

In this lab you are expected to perform the basics of cloning your Lab 2 repository from the GitLab course group. A link to the course group can be found here and your repository can be found in the Lab2 subgroup. See the Lab Technical Document for more information on using git. You will notice that your repository has a file in the **Lab2** directory named **delete_this_file**. Due to the limitations of GitLab, we are not able to push completely empty directories. Before you push your work to your repository (which should be in the **Lab2** directory anyway), make sure to first use the **git rm** command to remove the extra file. If you do not, your pipeline will fail.

Be sure to read this entire document before starting!

2 Simple C Code Example

Since you should already have some experience with a previous programming language, the easiest way to get started with the C programming language is to try a working example. Read through the following code and then use the instructions at the bottom of this page to get it running.

2.1 example.c

```c
// The stdio.h library gives us access to printf and scanf.
#include <stdio.h>

// The stdlib.h library gives us access to atoi.
#include <stdlib.h>

int changeNumber(int number)
{
    // Add 15 to the number.
    int newNumber = number + 15;

    // Multiply the number by 12.
    newNumber = newNumber * 12;

    // Subtract 15 from the number.
    newNumber = newNumber - 15;

    // Divide the number by 5. This acts as integer division!
    newNumber = newNumber / 5;

    // Find the remainder of the number divided by 150.
    newNumber = newNumber % 150;

    // Return the new number after calculations.
    return newNumber;
}

int main(int argc, char** argv)
{
    // Create four integer variables.
    int one, two, three, four;

    // Alert the user that they have to enter some numbers.
    printf("Enter four integers: ");

    // Retrieve four integers separated by spaces from stdin.
    scanf("%d %d %d %d", &one, &two, &three, &four);

    // Print the four integers to the screen.
    printf("Numbers Received   : %d %d %d %d\n", one, two, three, four);

    // Update each number using the changeNumber function.
    one   = changeNumber(one);
    two   = changeNumber(two);
    three = changeNumber(three);
    four  = changeNumber(four);

    // Print the updated numbers to the screen.
    printf("Updated Numbers    : %d %d %d %d\n", one, two, three, four);

    // Create four string variables, each of length 10.
    char s1[10], s2[10], s3[10], s4[10];

    // Convert the four integers to strings, stored in our string variables.
    printf("Converting integers to strings...");
    sprintf(s1, "%d", one);
    sprintf(s2, "%d", two);
    sprintf(s3, "%d", three);
    sprintf(s4, "%d", four);
    printf(" Done.\n");

    // Print the integers from various sources, including a original integer,
    // a string, and two strings converted back to integers.
    printf("Testing AtoI Output: %d %s %d %d\n", atoi(s1), s2, atoi(s3), four);

    // Explicit successful exit code 0.
    return 0;
}
```

2.2 Output

```
Enter four integers: 1 2 3 4
Numbers Received   : 1 2 3 4
Updated Numbers    : 35 37 40 42
Converting integers to strings... Done.
Testing AtoI Output: 35 37 40 42
```

2.3 Running example.c on Timberlea

Make a test directory somewhere on Timberlea, create a file named **example.c** and place the above source code inside it using **vim**. Save and close **vim**, then enter the commands at the Unix bash prompt:

```
gcc --std=c18 example.c -o example
./example
```

Take a few minutes to try different inputs to the program to see what happens.

3 Data Types, Keywords, and Language Features

The C programming language has a variety of keywords which are reserved and may only be used for their intended purpose. These reserved keywords offer a collection of functionality and make up the key building blocks of your C programs, some of which you will use very regularly, and some you may never use at all.

3.1 Data Types

The reserved words you are likely to use the most are related to **data types**. Since C is statically typed, all of the data types of variables and values are checked at compile time to ensure the logic of your code will not fail. This means you will have to be careful and specific when assigning variable types. However, C has some methods of ”breaking” the static typing rules, such as being able to push data of one type into another type using pointers. We will discuss that in a later lab.

Integer Types

Type	**Values**	**Bits (Bytes)**
char	A character, but can also be represented as an integer in [-128, 127].	8 (1)
short	An integer in the range [32,768, 32,767].	16 (2)
int	An integer in the range [2,147,483,648, 2,147,483,647]	32 (4)
long	An integer in the range [9,223,372,036,854,775,808, 9,223,372,036,854,775,807]	64 (8)

Floating Point (Real) Types

Type	**Values**	**Bits (Bytes)**
float	A real value in the range [-3.4028234664e+38, 3.4028234664e+38]. The smallest fraction is +/-1.1754943508e-38.	32 (4)
double	A real value in the range [-1.7976931349e+308, 1.7976931349e+308]. The smallest fraction is +/-2.2250738585e-308.	64 (8)

One of the problems with C (and other languages such as C++) is that finding a compiler which adheres exclusively to the C standard is fairly uncommon. Very often, compiler designers will add extra features on top of the C standard rules to make coding easier. These are called compiler extensions. The compiler we’re using in this course is the GNU Compiler Collection (gcc), which compiles C code by default. The value ranges seen in the tables above are not completely adherent to the C standard, but are instead adherent to the GCC compiler and the underlying Unix system configuration on our servers. As such, these values are guaranteed on Timberlea, but not on any other Unix installation or compiler. If you are writing code with a compiler other than the **gcc** compiler on Timberlea, we will not help you debug that code (as a rule) and we don’t promise any of the above values will be correct.

If you’re curious, you can find a list of the GNU compiler extensions here.

3.2 Keywords

Outside of the data types, the remaining keywords are related to control structures that give you the core functionality of the language. These include conditional statements, loops, and some other functionality related to memory. A table of the various keywords, and their usage, can be found here. We will introduce many of these keywords in the next few labs, so don’t feel that you have to know how to use them all immedidately. Just be aware that you cannot use these words for names of variables or functions.

3.3 Comments

Like most languages, C has comments. Comments are a type of annotation designed to make the code more meaningful to the reader. In all solutions, comments will be used to explain most lines of code, or blocks of code wherein multiple lines have a similar or compound functionality.

In this class, we will generally expect your code to be commented line-by-line, or block-by-block, especially if you’re doing something out of the ordinary. See the example code above for a good example of proper commenting.

Note that the functions in the **example.c** code do not have comment headers, but it can be useful (especially if using a documentation generator such as Doxygen) to provide function information before the function in a series of informational comments or a comment block to help a reader get an immediate understanding of the function’s design.

3.3.1 Single Line

A single line comment is written using two forward slashes, // **Comment Here**. It can appear anywhere on a line of code and anything *after it on the same line* will be ignored by the compiler. There are many different places where single line comments are used, and often people will use single line comments for all of their commenting needs, saving multi line comments for debugging purposes.

```c
1 // This is a single line comment. I'm making a variable.
2 int var = 5;
3
4 int a = 16; // This is another comment. This comment has the "read twice" problem.
```

Often you will see comments appear after a line of code (as seen on line 4 above). This can have a negative effect on the reader, as the reader will often read the code, then read the comment, then read the code again to confirm what they’ve read. This means that every line could be read twice and that can be tiresome for large codebases. We recommend you always put a single line comment above the line you're explaining. That way the reader reads the comment, then the code. There's no repeating, which makes your code flow better.

3.3.2 Multi Line

Multi line comments have a start and end pattern, **/* Comment Here */**. The start and end pattern can appear on different lines in your code, as long as the start pattern appears before the end pattern. Multi line comments can also stay on a single line if you want them to.

```c
/* This is a multi line comment on a single line. I'm making a variable. */
int var = 5;

/* This is a multi line comment on multiple lines.
   I'm making a variable. */
int a = 16;
```

If you use single line comments exclusively throughout your submission, you may find it easier to debug your code with multi line comments as you will never run into a start/end pattern clash.

```c
// Someday I may need to debug this, so I
// never use multi line comments to avoid
// clashing the patterns.
int var = 5;
int a = 16 + var;
```

```c
/* Commenting out the code was easy here.

// Someday I may need to debug this, so I
// never use multi line comments to avoid
// clashing the patterns.
int var = 5;
int a = 16 + var;

*/
```

```c
/* This is broken. I'm clashing with the comment I want to keep.

/* Someday I may need to debug this, so I
 * never use multi line comments to avoid
 * clashing the patterns.
 */
int var = 5;
int a = 16 + var;

*/
```

3.4 C Statements

Each program in C is a collection of computer directions referred to as statements. A statement is a command defined by the C programming language that, once compiled, will issue direct commands to the CPU to perform some kind of work. There are many types of statements in C (expression, selection, etc.), but expression statements are the primary type we will look at in this lab. Expression statements, in general, contain an optional expression and must end with a semicolon (;) so that the compiler understands when the statement is completed. You will see many examples of expression statements throughout this document, so you will have many opportunities to see how they are used.

3.5 Strings and Characters

Depending on your past programming knowledge, strings and characters may be considered the same thing. In C, this is not the case. A string is exclusively defined as any text surrounded by double quotes. A character is defined exclusively by a single character (or escape sequence) surrounded by single quotes. While there are other ways to initialize characters and strings, their quotation mark types are not interchangeable.

```c
char a[] = "This is a string";
char c   = 'A';
```

For this lab, you will only need to know how to use strings in the scope of **printf** and **scanf**, thus you will not need to know how to create, store, or manipulate strings until later.

4 C Libraries

The C programming language allows the inclusion of built-in and external code bases known as libraries. Libraries typically contain a collection of related functions which can provide additional functionality beyond the basic C language. Some benefits include the ability to read and write to files, read from the standard input and output streams, access to a greater range of math functions, and the ability to manipulate strings. The libraries below are used very often and would be helpful libraries to understand throughout this course.

For all of the libraries found below, a simple Google search will yield many pages outlining their use and the functions they provide. You can also use the **man** command on Timberlea to get a manual page for any built-in C library or C function.

4.1 stdio.h

The stdio.h library (standard input/output library) contains functions which allow you to interact with files and streams (read/write). Notable functions in this library include printf and scanf for basic writing and reading to the standard input/output streams, but also fputs and fgets for allowing buffered writing and reading. This library also contains functions for converting strings into other data types, such as sscanf.

4.2 stdlib.h

The stdlib.h library (general purpose standard library) contains functions for general purpose code control, including memory allocation, direct conversions from strings to other data types (atox functions), a very sub-par random number generator, the exit and system functions, and integer math functions. This library will be required in many of the future labs when dealing with manual memory allocation and pointers.

4.3 math.h

The math.h library contains a variety of more complicated math functions, such as trigonometric functions, ceiling, floor, log, and power.

4.4 string.h

The string.h library contains functions which allow you to compare and manipulate strings, which include functions for string comparison, finding the length of a string, and concatenating two strings together.

4.5 time.h

The time.h library contains functions for managing clock time, dates, and timers. If you need to time your code (which may happen in later labs), you may want to consider using this library.

5 Functions

Functions are a special type of code block which, at their most basic level, allow you to break your code into smaller pieces, which improves code readability, allows for streamlined code reuse, helps with code debugging, and reduces the length of your code. The components of a function declaration can be seen here:

```
ReturnType FunctionName(ParameterList)
{
    FunctionBody
}
```

The individual components of the function are described here:

Component	**Explanation**
Return Type	The return type of a function can be any data type. This includes the list provided above, as well as a few others we have not yet introduced, such as structs. The return type may also be a type you have defined yourself. If the return type is void, the function returns nothing. If the return type is not void, there must be at least one return call made in the code body. All return statements in a single function must return the type defined by the return type.
Function Name	The function name can be anything you'd like, although you should try to make it descrptive enough that you remember what it's doing when you are reading your code. Like variables, function names in C have their own set of expected naming conventions.
Parameter List	The parameter list is a comma separated list of variable type and name declarations. They must be in the format **dataType variableName** and should follow the C variable naming conventions. These parameters define the type and order of input arguments which must be provided when calling the function elsewhere in your code. If you attempt to pass values of the wrong data type to a function, or you pass them out of order, your compiler will fail.
Function Body	The function body is a list of statements which define the behaviour of your function. They are written like any other code, except if your return type is not void, you must include at least one return statement which returns a value of the correct type.

Here we can see the **changeNumber** function definition from the **example.c** code given above. The comments have been removed to keep it concise. The function accepts a single integer argument named **number**. It then creates a new integer and performs a series of arithmetic operations, while using the incoming integer as a starting points. Once the arithmetic operations are complete, the function returns the new integer to the calling code.

```c
int changeNumber(int number)
{
    int newNumber = number + 15;
    newNumber = newNumber * 12;
    newNumber = newNumber - 15;
    newNumber = newNumber / 5;
    newNumber = newNumber % 150;
    return newNumber;
}
```

Unlike some other languages, when you are writing functions in C, the order in which they are defined in your code is important. C will compile the functions in the order in which they are written. If a function at the top of your source file calls a function at the bottom of your source file, you will receive an **implicit declaration of function** warning, which will stop your code from compiling. In order to prevent these, you should create **function prototypes** using forward declaration. For information on how to do that, check the Header Files section below.

The main function is a special function used in C to define the starting point of your programs. When you compile, your program, it must contain a main function and (to follow good practice) it must minimally be in the following form:

```c
int main(int argc, char** argv)
{
    return 0;
}
```

The anatomy of the main function is important, as it relates to the way the operating system handles program execution. First, the main function should always accept two parameters: the argument count (argc) and an array of arguments (argv), in that order. When your program is executed, it's possible for a user to provide a list of arguments to your program, similar to how Unix commands work. The operating system tokenizes the entire command line execution, stores each token in an array, and then passes that array into the program as **argv**. You'll notice that the data type of argv is **char****, which (for now) can be interpreted simply as a list of strings. The other argument the operating system passes to your program is the number of elements in the argv array, which is provided as an integer we've named **argc**. These will be important in other labs, but for this lab they're just worth having a brief understanding of.

Second, the main function should always return an integer. After execution, your operating system will wait for your program to finish. Once the program has completed execution, the main function will return an integer to the operating system, which is then referred to as an **exit code**. Exit codes are a useful diagnostic tool for the operating system, as it provides information about whether the program exited in a success state, or a failure state. Program designers will often designate their exit codes to be specific reasons for failure, listing the possible codes in a table which you can reference to understand why a program execution failed.

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

int main(int argc, char** argv)
{
    // Create an empty string of length 20.
    char a[20];

    // Tell the user to provide a string that does not start with an integer..
    printf("Enter a string which doesn't start with an integer: ");

    // Accept a string from the user, then remove the \n character.
    fgets(a, 20, stdin);
    sscanf(a, "%s\n", a);

    // If the string starts with a number, we should exit with code 1.
    if(a[0] > 47 && a[0] < 58)
        return 1;

    // If we get here, the string is valid, so print it.
    printf("%s\n", a);

    // Program exited successfully with code 0.
    return 0;
}
```

6 Input and Output Streams

In the C programming language, the **stdio.h** library provides a variety of functions for interacting with files and streams, but in this lab you will only be required to use simple input and output methods. In future labs you will be expected to understand the stdio.h library much more comprehensively.

6.1 Printing to stdout

For this lab, you can print to the stdout stream using the **printf** function, available by including the stdio.h library in your code. The printf function works differently from printing in other languages, as it uses format specifiers in a pattern string to determine output of variables in text format.

The default form of printf uses no variables and simply prints a string to the terminal screen.

```
printf("This is a string on your screen!\n");
```

Notice the \n escape sequence in the string. C is capable of recognizing a variety of escape characters. The most common escape characters you'll likely to use are:

Sequence	Result
\n	New line
\t	Horizontal tab space
\\	Backslash
\'	Single quote
\"	Double quote

In addition to escape characters, printf can also use pattern matching with codes called **format specifiers**. A format specifier tells C what kind of value it should expect to see when attempting to insert a value into the printf output stream. A list of common format specifiers can be found here:

Specifier	Data Type
%c	char
%d	int
%ld	long (long int)
%f	float (sometimes double)
%lf	double (long float)
%s	string

To use a format specifier in a printf statement, you construct a string as normal, but place the format specifier (a percent sign followed by a text pattern) where you expect your data value to appear. You can have multiple format specifiers in a single string. Once you have completed your string in the printf statement, you must follow it with a comma separated list of variable names. The variables' data will attempt to be inserted in the order of the format specifiers and the variable list. That is, the first specifier in your string will attempt to be filled by the data in the first variable listed, and so on. If a format specifier does not match the data type of a listed paired variable, the compiler will produce an error.

An example program for printing values of each of these types is available here:

```
#include <stdio.h>

int main(int argc, char** argv)
{
    char c    = 'a';
    int  i    = 15;
    long l    = 15L;
    float f   = 3.14159;
    double d  = 3.141591;
    char s[]  = "Whoa";

    printf("Char   : %c\n", c);
    printf("Int    : %d\n", d);
    printf("Long   : %ld\n", l);
    printf("Float  : %f\n", f);
    printf("Double: %f or %lf\n", d, d);
    printf("String: %s\n", s);

    return 0;
}
```

6.2 Reading from stdin

For this lab, you can accept input from the user via stdin using the **scanf** function, available by including the stdio.h library in your code. The scanf function operates in a fashion similar to printf in the way that it scans through the standard input stream, checking the text for pattern matches. If it finds a pattern match based on the string you provide it, it will store the values caught by that pattern in the variables you provide it. The pattern string scanf uses is constructed from text and format specifiers from the table above in order to filter and accept input.

A simple program to read in a single integer and a string from the user could be:

```c
#include <stdio.h>

int main(int argc, char** argv)
{
    int var;
    char a[50];

    scanf("%d %s", &var, &a);

    printf("%d\n", var);
    printf("%s\n", a);

    return 0;
}
```

Notice that the scanf and printf are very similar, with one key difference being the addition of **&** symbols before the variable names in the scanf variable list. The symbol is meant to get the memory address of a variable (referred to as the ”Address-of” operator) and has a variety of uses in C. Since this operator is related to pointers, which we are not covering in this lab, we will return to it at a later date.

One thing to be aware of with scanf is that it is capable of causing a **buffer overflow**. A buffer overflow can happen when an array or string is not allocated enough memory (isn't long enough) to accommodate something you're trying to store inside them. In the case of scanf, if you create a string variable and try to use %s to store a string from the user which is too big for your variable, you will end up with **undefined behaviour**.

If you spend any period of time working with C, you will run into the phrase ”undefined behaviour” often. The C standard has expectations of what you will do, but doesn't cover every single possibility. In situations where the specification does not tell us exactly what should happen, programmers writing compilers will instead do whatever they think is most useful: sometimes that means crashing the program, sometimes that means giving the action ”its best shot” at success, sometimes it means letting your software do any number of crazy things such as reading uninitialized memory. One thing that happens on one system may not happen on another, so be very careful when dealing with outcomes that are considered undefined!

In the case of scanf on Timberlea, it appears that it goes for the ”the best shot” form of undefined behaviour, where scanf will read the input from the user and fill your string with as much content as possible before stopping. This behaviour makes it similar to the **fgets** function, which is a far safer way of dealing with user and file inputs. We will talk more about that in the future. For now, try running the following program and see what output you get if you try to enter more than 10 characters to the input!

```c
#include <stdio.h>

int main(int argc, char** argv)
{
    char a[10];

    scanf("%s", &a);

    printf("%s\n", a);

    return 0;
}
```

7 Arithmetic

The basic arithmetic for C is similar to other languages, where you can add, subtract, multiply, divide, as well as calculate the division remainder with the modulus operator. Examples of those can be found in **example.c** in the **changeNumber** function.

C also allows you to increment and decrement a value by 1 by using the ++ and −− operators. These operators can be placed before or after a variable in order for you to choose when the increment or decrement happens. If the operator is before a variable (such as ++**i**), then the value of the variable will be increased or decreased by 1 *before* the value of the variable is used by your program.

```c
// Create an integer variable i and set it to 4.
int i = 4;

// A prefix increment or decrement operation such as this one...
int a = i++;

// ...is equivalent to the following code.
int a = i;
i = i + 1;

// This means a = 4, and i = 5
```

If the operator is after a variable (such as **i**++), then the value of the variable will be used by your program before it is incremented or decremented.

```c
// Create an integer variable i and set it to 4.
int i = 4;

// A postfix increment or decrement operation such as this one...
int a = --i;

// ...is equivalent to the following code.
i = i - 1;
int a = i;

// This means a = 3 and i = 3.
```

Note that the two above code blocks won't run, as they are an example for illustrating a comparison only.

8 Header Files

Header files are a means of organizing your code so that both you and the compiler are capable of better managing function definitions, macros, and preprocessor definitions. Header files come in two forms and are imported in different ways.

The first form is the built-in header type we call **libraries**. Built-in libraries are provided by the C compiler and/or the operating system and can be imported into your program using the following command:

```c
#include <library.h>
```

The second form of header is the user-created header, which you can write yourself. A header file written by you should always end with the **.h** file extension and can include any forward declarations and macros you desire. To import a user-generated header into your program, you can use the following command:

```c
#include "myHeader.h"
```

Notice that the user-generated header is included in double quotation marks, not within angle brackets. When you surround your header file name with double quotes, the C compiler will look for the header files in the current working directory. If you want to include other directories for your compiler to look for header files, you will need to compile using the **-I** option and include file paths to the headers.

8.1 Writing a Header File

Header files are written in a similar fashion to C source files, but with some changes. Headers should never contain any function bodies or code statements. Any function declarations in the header file should be forward declarations only. Preprocessor definitions can be included, and define guards should be used.
The primary purpose for creating header files is to provide forward declarations to C source files so that the compiler knows what to expect, and from where. When C compiles code, it will compile the files in the order that they are listed in the compiler command, and they will compile code in order in your files. When a C compiler finds **include** statements, it will read those headers and store any information inside before proceeding through the source while that imported them.
Under normal circumstances, functions in C are compiled in the order in which they are found. Sometimes you will create a situation in your code where functions can have **circular references**, which means that functionA can call functionB, but functionB can also call functionA. In this situation, it's impossible for your code to compile normally. To alleviate this problem, we can create **forward function declarations**. A forward declaration is provided to the C compiler before any functions are implemented (given a function body) which tell the compiler what to expect in the future.

Consider the following example:

```c
int functionA(int a)
{
    if(a <= 0)
        return 5;

    return functionC(a-1);
}

int functionB(int b)
{
    return functionA(b-1);
}

int functionC(int c)
{
    return functionB(c-1);
}

int main(int argc, char** argv)
{
    functionA(10);
    return 0;
}
```

In this example, we have a main function which calls functionA with a value of 10 as the only parameter. If the value coming into functionA is ever 0 or less, it will return 5 and the cycle of execution will end. Otherwise A will call C, C will call B, B will call A, and this execution path will continue forever with the integer argument decreasing by 1 on every function call.

The problem with this execution is that there is a circular dependency in the function calls: A needs C to compile, C needs B to compile, and B needs A to compile. Since there's no order where we can make each function appear before each of the other functions, we will need to create some forward declarations. While it is not necessary to create a forward declaration for every function, it's good practice for clarity, so we will always do so.

```c
int functionA(int);
int functionB(int);
int functionC(int);

int functionA(int a)
```

```c
{
    if(a <= 0)
        return 5;

    return functionC(a-1);
}

int functionB(int b)
{
    return functionA(b-1);
}

int functionC(int c)
{
    return functionB(c-1);
}

int main(int argc, char** argv)
{
    functionA(10);
    return 0;
}
```

As you can see above, creating a forward declaration is as simple as declaring the function without a body before any other functions are implemented. They are represented as a return value, the function name, and a parameter list in the form of a signature (which is just the list of types, in order, but which do not require any variable names). This is then closed with a semi-colon as any other C statement would be. When the C compiler sees a forward declaration, it will keep a note of it as it proceeds, so that if it sees your code trying to call that function, it will remember that you promised it would be implemented somewhere in the compilation process. If you write a forward declaration and then never implement (provide a function body for) the function later, the C linker will fail.

Forward declarations as above are great, but if you have a lot of functions it will quickly take up a lot of space in your C source files. This is where you should create a Header file and store your forward declarations there instead. Let's take a look at an example of how to do so here.

```c
// functions.h
#ifndef __FUNCTION_HEADER
#define __FUNCTION_HEADER

int functionA(int);
int functionB(int);
int functionC(int);

#endif
```

```c
// main.c
#include "functions.h"

int functionA(int a)
{
    if(a <= 0)
        return 5;

    return functionC(a-1);
}

int functionB(int b)
{
    return functionA(b-1);
}

int functionC(int c)
{
    return functionB(c-1);
}

int main(int argc, char** argv)
{
    functionA(10);
    return 0;
}
```

Starting with the main.c file, we can see that we've removed the forward declarations and added **#include "functions.h"**. This preprocessor directive will tell the C compiler to scan the function.h file and any lines it sees will be added directly to this C source file before continuing. This means it will always compile the header in the place you've added the include directive.

The functions.h file is more interesting. It contains the three forward declarations as normal, but we've also included what's called a **header, define, or include guard**. A header guard uses preprocessor directives to prevent importing the same code too many times, either explicitly or implicitly, on a per-source file basis. The **ifndef** directive says that if the following definition exists (in this case a definition named **__FUNCTION_HEADER**, but in practice should be a unique label of your choice for each header file), then we will not scan any of the code in the if block. If the definition does not yet exist, then we define it and scan the forward declarations. In practice, this means that each source file should only get one copy of the forward declarations added during the compilation process, and they should always be added as soon as they are needed by the first include directive. Since C is more than happy to repeatedly add the same code over and over if you don't pay attention to things such as nested include directives, the header guard should at least help you avoid some unneeded errors.

9 Compiling C Code

For the remainder of this course you will be working on writing C programs, with your primary compiler being **gcc** on the Timberlea server. This section will provide you with information on how the GNU Compiler Collection works so you can make the best use of the programming language throughout this semester.

9.1 Compiling Basics

The most basic form of compilation you can perform with gcc is to simply provide it a C source file. If you provide a file ending with a .c extension, it will compile it in C mode.

```
gcc file.c
```

In most beginner circumstances, your code will compile fine if you run it this way. However, by default, gcc will use the C89 standard from 1989 to compile your code. There have been many changes to the standard over the years and the most recent standard available on Timberlea is C18 from 2018. We will be using that in these labs, so you will have to tell gcc to use that standard specifically. To do so, you can use the −−**std** command:

```
gcc --std=c18 file.c
```

This will force your code to compile to the C18 standard and, by default, will produce an executable file named **a.out**. In order for you to execute your program, your **a.out** file will need user executable permissions. It should also have those by default, but if it does not you can use **chmod** to set the permissions to whatever you'd like. If you'd like your compiled code executable to have a nicer name, you can use the **-o** option with gcc to have it name the executable something different:

```
gcc --std=c18 file.c -o execute_me
```

Instead of a.out, this will produce an executable called **execute_me**. To run any of your compiled programs (assuming they have execution permissions), you can tell Unix to run them directly with **./execute_me**, **./a.out**, or any other file name you give them.

9.2 Steps of Compilation

In general, there are only a few major steps in compiling your code: the **preprocessor**, the **compiler**, the **assembler**, and the **linker**. These steps are taken in the order shown here, controlled completely by your compiler software. Note that we often refer to the entire process as compilation, and the program which makes it happen a compiler. Don't get this confused with the compilation step!
Be aware that the C compiler makes a single pass through your code, unlike other some other programming languages. This limitation is the reason why we need header files and forward function declarations in order to let the C compiler know about things it should expect to see.

9.2.1 Preprocessor

The preprocessing section of the C compiler looks for any preprocessor directives (which start with a) and enforce them. During this process it will keep track of any function declarations or data types that are necessary for the compiling step. If it detects any incorrect ordering of variables, data types, or functions, it will cause an error here.
If you want to stop the compilation process and have GCC provide you the preprocessor file (assuming it was successfully created), you can execute gcc with the **-E** option.

```
gcc --std=c18 -E file.c
```

Note that this command will likely just print the preprocessor output to your screen. If you want to be able to sort through it manually, I would suggest piping the output into the **less** program. Alternatively, you can use file redirection to save it in a file, or use the **-o** option to have it save to a specific filename.

9.2.2 Compiler

The compiler section of the compilation process views the preprocessor outputs and converts your program to assembly code. Sometimes you will see the compiler and the assembler represented as just the compiler (thus making this a three stage process), but it's worth differentiating them here. If you would like to see the assembly code generated by the compiler, you can use the **-S** option.

```
gcc --std=c18 -S file.c
```

This will automatically generate a **file.s** assembly code file that you can view.

9.2.3 Assembler

The assembler section of the compilation process converts the assembly code generated by the compiler into a binary object file. Note that if you provide multiple source files in your compiler command, it will generate an object file for each one separately before pushing them to the linker. The binary object file contains all of the machine code specifically designed to run on your compiled operating system. Since assembly can differ by system, your machine code may not necessarily run on another machine.
If you want to generate a binary object file without linking it, you can use the **-c** option on your compiler.

```
gcc --std=c18 -c file.c
```

This will automatically generate a **file.o** object file. Since it is in binary, it will be impossible to read with regular text editors. In the labs, all of our test files will be accompanied by a .o object file containing a main method and forward declarations based on the coding contracts provided.

9.3 Compiling with Libraries

Some libraries require you to tell the compiler to load them during compilation time. For example, the math.h library requires gcc to be executed with the **-lm** (or **-lmath**) option. If you are having trouble compiling with some built in libraries, check online or the library's **man** page to determine if you need to enable additional options on your compiler.

9.4 Compiling Lab Object Files

In the lab section of this course, you will be given the option to test your code with a variety of our test object files. **Do not confuse these with objects from object-oriented languages**, such as Java or Python. C object files contain pre-compiled binary code which you can compile with your own programs and are easily identified by their **.o** extension. Our test objects will be provided to you in the **CI/objects/** directory of your repository.

Each object file will be compiled on Timberlea and will contain a main method with a selection of test cases for the functions you have to write. It will have been compiled with a header file dictating forward function declarations which match the relevant coding contracts. In order for you to test your functions with our object files, you must select the correct object file (as per the contract instructions) and compile it together with your code using the following gcc command:

```
gcc --std=c18 source.c test.o
```

This should compile and generate the **./a.out** file as normal. As mentioned in this section, you can feel free to use the **-o** option to have the compiler rename your executable. This same test file will be used in the pipeline for each of your submissions, but the option is available for you to test offline using this method.

The object files provided by us contain everything your program will need to properly execute if compiled the same as the above example. Because they already know which functions to expect from your code, not providing every required function in a contract will produce errors letting you know that a function was promised, but never implemented. If you try to compile your code along with one of the provided object files and you get those errors, make sure you check that you have provided function implementations (actual coded logic) for each required function and ensure your **.c** file is being compiled properly with the **.o** file.

10 Comparing Your Outputs to Expected Outputs

In this lab (and beyond) we will provide you with expected output files, which are used in the pipeline to determine if your program's output matches the correct output. This is done in a single Unix if statement, where we use the diff command and wc command to decide whether or not your output is valid.

The **diff** command tells you the difference between two files. If two files are exactly the same, the diff command produces no output. Otherwise it lists which lines are different in each file. Check the **man** page for diff, or the internet, for more information. It's a very well-documented command.

The **wc** command outputs word, line, and character counts across a series of files or text input. We specifically use the **-c** option so that it will return a single value, which is the number of characters (bytes) in the input. Again, the **man** page and internet documentation for wc is extensive.

The marking section of the pipeline script performs a **diff** on your output file and the expected output file. It then takes the output from diff and pipes it into **wc -c**, which returns the number of characters in the diff's output. If that number is greater than 0, then that means it found at least one line in your output that doesn't match the expected output. You can run the same commands via the terminal, and you are provided the expected files in the **CI/** directory of your repository. If your pipeline fails on that check, we suggest viewing the diff output of your output files to see where you're going wrong.

If you are having pipeline failures, but it appears that your output values are correct, double check your use of whitespace and new lines by using diff. It could save you a lot of headaches and disappointment!

11 Lab 2 Function Contracts

In this lab you will be responsible for fulfilling three lab contracts: the **Coins** contract, the **Power Rule** contract, and the **Triangle** contract. Each contract is designed to test you on some of the things you've learned throughout the instruction portion of this lab.

All contracts must be completed exactly as the requirements are laid out. Be very careful to thoroughly read the contract instructions before proceeding.

All contracts are designed to be submitted without a main function, but that does not mean you cannot write a main function in order to test your code yourself while you're writing your programs. It may be more convenient for you to write a C source file with a main function by itself and take advantage of the compiler's ability to link files together by accepting multiple source files as inputs. When you push your code to Gitlab, you don't need to **git add** any of your extra main function source files.

For those of you who are concerned, when deciding which naming conventions you want to use in your code, favour consistency in style, not dedication to a style that doesn't work.

The contracts in this document are worth the following points values, for a total of 10.

Contract	**Points**
Coins	2
Power	3
Triangle	5
Total	10

11.1 Coins

11.1.1 Problem

You are given a number of pennies between 0 and 10000 (inclusive). How much money do we have if we convert the pennies to higher denominations, such that we have as many dollars as possible, followed by as many quarters as possible, then as many dimes as possible? How many pennies are left over?

11.1.2 Preconditions

You must provide a function which meets the requirements in the table below. You you may include other functions as long as the requested function executes correctly. Do not include a main function in your source or header files.

Requirement	**Conditions**
Function	void calculateCoins(int)
Input Parameters	Your function should accept a single integer, which represents some number of pennies. The integer provided is guaranteed to be in the range [0, 10000].
Return Value	Your function should not return a value.
Files Required	coins.c, coins.h

11.1.3 Postconditions

Your function must output to the standard output stream. The output must print the number of coins of each type in descending order by largest denomination. Your output must be terminated with a new line character.

11.1.4 Restrictions

You may not use any C statements or structures not introduced in this lab, which includes (but is not limited to) conditional statements and loops.

Important Note: The pipeline does not confirm these restrictions, but our marking scripts will, so the pipeline will not fail if any restricted code is included. However, if the marking script catches restricted code in your program, you will receive 0 for this contract.

11.1.5 File Requirements

This contract requires you to provide a C source file named **coins.c** and a C header file named **coins.h**. Your header file should contain your forward declarations. Your source file must not contain a main function, or it will fail during marking.

Your source and header files should be placed in the **Lab2/coins/** directory in your GitLab repository.

11.1.6 Testing

To test your code, you can compile your source file with the **coinsM.o** object file found in **CI/objects/coins/**. It can then be executed as normal. The coinsM.o file contains a main function, so you do not need to provide one.

11.1.7 Sample Inputs and Outputs

Input	**Output**
calculateCoins(1)	0 dollars, 0 quarters, 0 dimes, 0 nickels, 1 pennies
calculateCoins(1999)	19 dollars, 3 quarters, 2 dimes, 0 nickels, 4 pennies
calculateCoins(2000)	20 dollars, 0 quarters, 0 dimes, 0 nickels, 0 pennies
calculateCoins(1850)	18 dollars, 2 quarters, 0 dimes, 0 nickels, 0 pennies

11.2 Power Rule

11.2.1 Problem

Given a polynomial, use the power rule to find the first derivative.

11.2.2 Preconditions

You must provide a series of functions which meet the requirements in the table below. You you may include other functions as long as the requested functions execute correctly. Do not include a main function in your source or header files.

Each of your functions represents a polynomial of the degree included in its name. For example, **powerRule2** represents a power rule being applied to a polynomial of degree 2, which means its term with the largest exponent is 2. If you are not sure what the power rule is, a quick Google search will give you the formula.

Requirement	**Conditions**
Functions	void powerRule1(int, int) void powerRule2(int, int, int) void powerRule3(int, int, int, int) viud powerRule4(int, int, int, int, int)
Input Parameters	You are given a series of positive integers which represent the coefficients of the terms of the function's related polynomial, from highest degree term to the lowest. For example, powerRule2's related polynomial would be of the form: $ax^2 + bx^1 + cx^0$ which means we would call the function as **powerRule2(a, b, c)**.
Return Value	Your function should not return a value.
Files Required	power.c, power.h

11.2.3 Postconditions

Your function must output to the standard output stream. The output must print each derivative term, separated by plus signs (+) and the output must be terminated with a new line character. Each plus sign should have a single space on either side. You must use the hat character (ˆ) to signify an exponent, but you should not print an exponent unless it is positive. If you have a term with a zero exponent, you should only print the coefficient.

11.2.4 Restrictions

You may not use any C statements or structures not introduced in this lab, which includes (but is not limited to) conditional statements and loops.

Important Note: The pipeline does not confirm these restrictions, but our marking scripts will, so the pipeline will not fail if any restricted code is included. However, if the marking script catches restricted code in your program, you will receive 0 for this contract.

11.2.5 File Requirements

This contract requires you to provide a C source file named **power.c** and a C header file named **power.h**. Your header file should contain your forward declarations. Your source file must not contain a main function, or it will fail during marking.

Your source and header files should be placed in the **Lab2/power/** directory in your GitLab repository.

11.2.6 Testing

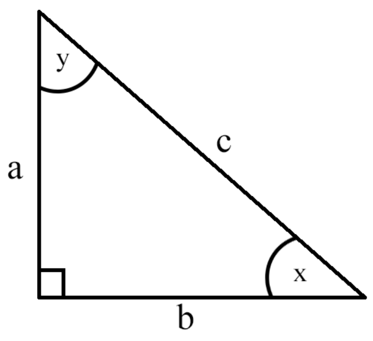
To test your code, you can compile your source file with the **powerM.o** object file found in **CI/objects/power/**. It can then be executed as normal.

11.2.7 Sample Inputs and Outputs

Input	**Output**
powerRule4(4, 3, 10, 4, 5)	16xˆ3 + 9xˆ2 + 20x + 4
powerRule4(1, 12, 1, 2, 2)	4xˆ3 + 36xˆ2 + 2x + 2
powerRule2(1, 2, 3)	2x + 2

11.3 Triangles

11.3.1 Problem



Given a right triangle, the length of two of its sides, and the interior angles in radians, find the length of the missing side. Input will always be given in the order of (opposite, adjacent, hypotenuse, x, y), with the missing side omitted. In this challenge, you will be writing functions to find the missing side length using both the Pythagorean theorem and by using trigonometry. Then, print the dimensions of all three sides, and both angles. The block of dimensions should start and end with a new line.

11.3.2 Preconditions

You must provide a series of functions which meet the requirements in the table below. You you may include other functions as long as the requested functions execute correctly. Do not include a main function in your source or header files.

In this contract, are you are only able to use a single printf statement (or equivalent) in any of your files. Having more than one printf statement will result in a grade of zero for this contract.

Each function asks you to find the opposite, adjacent, or hypotenuse using either the Pythagorean theorem (Pyth) or trigonometry (Trig). These functions will be checked to ensure proper math functions are used. Math functions can be found in the **math.h** library.

Requirement	**Conditions**
Functions	double findOppositePyth(double, double, double, double) double findAdjacentPyth(double, double, double, double) double findHypotenusePyth(double, double, double, double) double findOppositeTrig(double, double, double, double) double findAdjacentTrig(double, double, double, double) double findHypotenuseTrig(double, double, double, double)
Input Parameters	You are given a series of positive double floating point values which represent the sides and angles of a right triangle. The values correspond to the triangle diagram provided above, such that the values will always be in the order (a, b, c, x, y). The only difference between inputs on the various functions is that the side in the name will be missing, thus only providing you four values. Those four values still remain in the same order, so, for example, if you are calling *findOppositePyth*, the function arguments you receive will be (b, c, x, y), as the opposite side (a) needs to be calculated.
Return Value	Your function should return a double floating point value representing the length of the missing side of the triangle.
Files Required	triangle.c, triangle.h

11.3.3 Postconditions

Your function must output to the standard output stream, but it may only do so once per function call. The output consists of seven lines: a blank line, a line containing the value of the opposite side, a line containing the value of the adjacent side, a line containing the value of the hypotenuse, a line containing the value of the x angle, a line containing the value of the y angle, and another blank line. Each line with a value must include the name of the value and an equals sign (=). When printing your double values to the screen, you should print them to two decimal places. **This is not necessarily the same as rounding!** Make sure you research floating point format specifiers with **printf** and how to limit the number of decimal places printed.

11.3.4 Restrictions

You may not use any C statements or structures not introduced in this lab, which includes (but is not limited to) conditional statements and loops.

Important Note: The pipeline does not confirm these restrictions, but our marking scripts will, so the pipeline will not fail if any restricted code is included. However, if the marking script catches restricted code in your program, you will receive 0 for this contract.

11.3.5 File Requirements

This contract requires you to provide a C source file named **triangle.c** and a C header file named **triangle.h**. Your header file should contain your forward declarations. Your source file must not contain a main function, or it will fail during marking.

Your source and header files should be placed in the **Lab2/triangle/** directory in your GitLab repository.

11.3.6 Testing

To test your code, you can compile your source file with the **triangleM.o** object file found in **CI/objects/triangle/**. It can then be executed as normal. Note that because we're using floating point values, it's possible that your values may vary slightly after several decimal places are considered. By reducing the calculated values to two decimal places, we limit the error and have never had a correct submission fail the pipeline. However, it's possible that we could run into some kind of error with a valid calculation. If you have problems with your calculations failing because of a slight error at two decimal places, let us know right away and we can work out a solution.

11.3.7 Sample Inputs and Outputs

Input	**Output**
findOppositePyth(62, 86.9770084563, 0.7772682612, 0.7935280656)	opposite = 61.00 adjacent = 62.00 hypotenuse = 86.98 x = 0.78 y = 0.79
findAdjacentPyth(97, 129.6341004520, 0.8454352154, 0.7253611114)	opposite = 97.00 adjacent = 86.00 hypotenuse = 129.63 x = 0.85 y = 0.73
findHypotenusePyth(77, 78, 0.7789466400, 0.7918496868)	opposite = 77.00 adjacent = 78.00 hypotenuse = 109.60 x = 0.78 y = 0.79
findOppositeTrig(9, 53.7587202229, 1.4025895549, 0.1682067719)	opposite = 53.00 adjacent = 9.00 hypotenuse = 53.76 x = 1.40 y = 0.17
findAdjacentTrig(33, 60.7453701939, 0.5743048302, 0.9964914966)	opposite = 33.00 adjacent = 51.00 hypotenuse = 60.75 x = 0.57 y = 1.00
findHypotenuseTrig(78, 71, 0.8323435117, 0.7384528151)	opposite = 78.00 adjacent = 71.00 hypotenuse = 105.48 x = 0.83 y = 0.74

12 Submission

12.1 Required Files

When submitting this lab, you must submit your C source and header files only. Each file must be contained in the directory listed in the structure requirement diagram below. These files include:

1. coins.c
2. coins.h
3. power.c
4. power.h
5. triangle.c
6. triangle.h

As with all labs, submitting anything other than what is required in this section will yield a mark of 0.

12.2 Submission Procedure and Expectations

Your code will be submitted to your Lab 2 GitLab repository using the same method as outlined in the Lab Technical Document. Refer to that document if you do not remember how to submit files via the GitLab service. A link to your repository can be found in the **Lab2** subgroup of the CSCI 2122 GitLab group here.

As mentioned in the Lab Technical Document, we will provide you with a CI/CD script file which will help you test your submissions. The .yml file containing the CI/CD test script logic, and any other necessary script files, are available in your repository at all times. You are free to view any of the script files to help you understand how our marking scripts will function. We make extensive use of relative path structures for testing purposes, which is why strict adherence to directory structure and file contents is such a necessity. Also remember to check your pipeline job outputs on the GitLab web interface for your repository to see where your jobs might be failing.

Remember to follow the instruction guidelines as exactly as possible. Sometimes the pipeline scripts will not test every detail of your submission. **Do not rely on us to perfectly test your code before submission.** The CI/CD pipeline is a great tool for helping you debug major parts of your submissions, but you are still expected to follow all rules as they have been laid out.

12.3 Submission Structure

In order for a submission to be considered valid, and thus gradable, your git repository must contain directories and files in the following structure:

```
Lab2/
 ├─ coins/
 │   ├─ coins.c
 │   └─ coins.h
 ├─ power/
 │   ├─ power.c
 │   └─ power.h
 └─ triangle/
     ├─ triangle.c
     └─ triangle.h
```

As with all labs, accuracy is incredibly important. When submitting any code for labs in this class, you *must* adhere to the directory structure and naming requirements in the above diagram. Failure to do so will yield a mark of 0.

Remember to remove **Lab2/delete_this_file** from your repository using **git rm** to avoid any pipeline failures.

12.4 Marks

This lab is marked out of 10 points. All of the marks in this lab are based on the successful execution of each contract. Any marking pipeline failures of a given contract will result in a mark of 0 for that contract. Successful completion of the various contracts will award points based on the following table:

Contract	**Points**
Coins	2
Power	3
Triangle	5
Total	10

There are no partial points available for these contracts, but it's likely that if your pipelines pass, you should get full marks. Be careful to adhere strictly to the contract requirements and restrictions to avoid accidentally submitting code that may pass the pipeline, but could result in failure due to using disallowed statements or structures.