School Of Arts and Sciences
By: Ismael Francisco - BSCS
Intended for All

Programming 1
FUNDAMENTALS

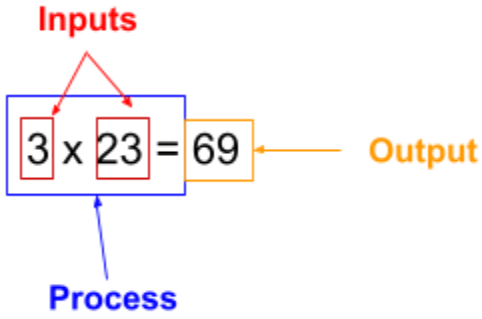
—

Notes

Introduction: What is Programming?

Programming is all about Algorithms - an Algorithm is a step-by-step process in the hopes of achieving a desired outcome. So in short, Programming is all about a step-by-step process. If it sounds oversimplified it's bec. it is. But don't worry.

Programming will always be composed of three parts , namely, the ***Input, Process, and the Output***. For instance, it takes at least two factors (A.K.A. multiplicand and multiplier) , and the process of Multiplication to end up with a result (A.K.A. the product of the multiplicand and multiplier). In this instance, the two factors are our inputs, our 'Process' is the process of multiplying said factors and our output is of course, the product that we have in the end.

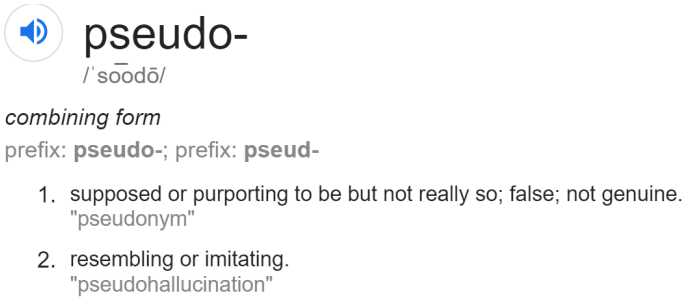


Before we can even begin to code using high-level programming languages, let us first tackle how to even begin with drafting the flow of our codes - this can be done using either a **Pseudocode or Flowchart**.

Things you must know and have beforehand:

- List of keywords in C
- Table of operators with precedence and associativity
- ASCII table
- How to represent arithmetic expressions
- How to use relational operators, logical operators and arithmetic operators
    - If this is a bit hard for you, please let me (Isma) know so I can help.

1. PSEUDOCODE


pseudo-
/ˈso͞odō/

combining form

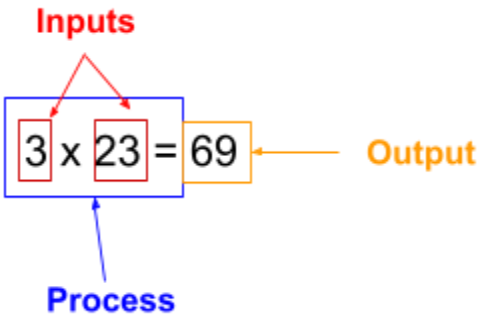
prefix: **pseudo-**; prefix: **pseud-**

1. supposed or purporting to be but not really so; false; not genuine.
   "pseudonym"
2. resembling or imitating.
   "pseudohallucination"

A Pseudocode resembles a code but in its simplified form. This is often used for program design. This will allow us to think more on the structure of our code before we even begin coding using our designated Programming Language. Pseudocodes may be initially written in simple English Language until we can slowly translate it into programming *jargons* or statements.

EXAMPLE (Multiplication Pseudocode)

Using the same example we had earlier. Let us try to create a pseudocode for multiplying two numbers.



Again, everything in programming is divided into 3 parts and should always follow the sequence of Input, then its Process, then the Output. So our Pseudocode may look something like this: (We encourage you to try and make your own PseudoCode before checking the one below, but it's up to you)

1. Input two numbers
2. Multiply the two numbers
3. Show the result of multiplying

or

1. Input your multiplicand
2. Input your multiplier
3. Multiply the two factors
4. Show the product

Your Pseudocode does not need to look exactly like the images provided above. You can use different wordings and phrases but the essence should be similar to the ones above. Ideally, you want your Pseudocode's statements to be closer to a programming statement but for now, this will do. The examples above will be altered later on when we get to discussing our Program.

(As an activity, you can try to do the same Pseudocode above but with different arithmetic operations like division and the likes.)

2. FLOWCHART

A flowchart is very similar to a Pseudocode in the sense that they both allow a programmer to first design their program. The difference is that in a Flowchart, things are represented through symbols and in a diagram form rather than your typical bulleted list in a Pseudocode.

Link for symbols and diagrams:
https://www.programiz.com/article/flowchart-programming

Symbols Used In Flowchart

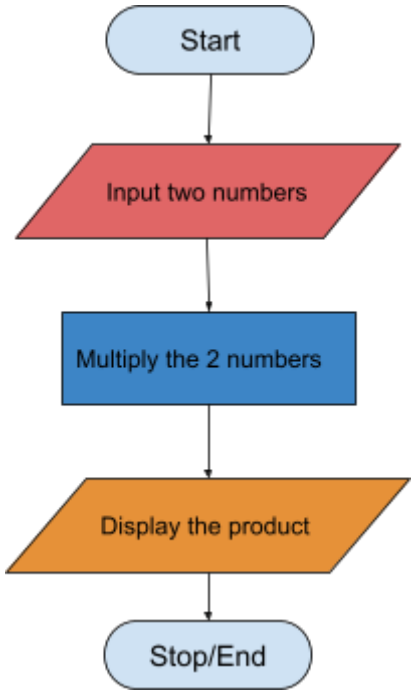
Symbol	Purpose	Description
	Flow line	Indicates the flow of logic by connecting symbols.
	Terminal(Stop/Start)	Represents the start and the end of a flowchart.
	Input/Output	Used for input and output operation.
	Processing	Used for arithmetic operations and data-manipulations.
	Decision	Used for decision making between two or more alternatives.
	On-page Connector	Used to join different flowline
	Off-page Connector	Used to connect the flowchart portion on a different page.
	Predefined Process/Function	Represents a group of statements performing one processing task.

(Please note that the symbol for input and output may vary - this depends on what your teacher will use also pls click the link above the diagram for a better view)

EXAMPLE (Multiplication Flowchart)

1. Input two numbers
2. Multiply the two numbers
3. Show the result of multiplying

The flowchart of our Multiplication Pseudocode may look something like this:



3. PROGRAM

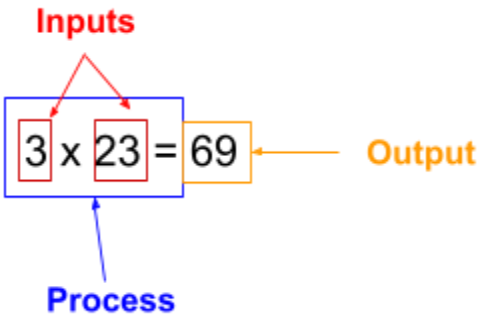
This is where we translate our Pseudocode / Flowcharts into our designated High-Level Programming Language - this is in order for the computer to interpret the set of *instructions or step-by-step process* that we want it to execute. For this course, we will be using C as our programming language.

3.1 QUICK HISTORY OF C PROGRAMMING

- Invented by Dennis M. Ritchie in 1972
- At the AT&T Laboratory

3.2 LITERAL VALUES

In our multiplication example, we've been using whole numbers as our factors and product. In C Programming, these numbers are what we call ***literal values.*** The term "***literal***" reflects the fact that these numbers literally display their values. Text such as "Sepnu Puas" is also considered a ***literal value*** because the text itself is displayed. Literal values are also known as constants.



You have been using literal values throughout your life and have commonly referred to them as numbers and words. In the sections below, you will see some examples of non-literal values - these are values that do not display themselves but are stored and accessed using identifiers. (If it sounds to *jargony* don't worry it'll make more sense as we go along)

3.3 VARIABLES

var·i·a·ble
/ˈverēəb(ə)l/

See definitions in:

All | Mathematics | Computing | Science | Nautical

noun

- COMPUTING
  a data item that may take on more than one value during the runtime of a program.

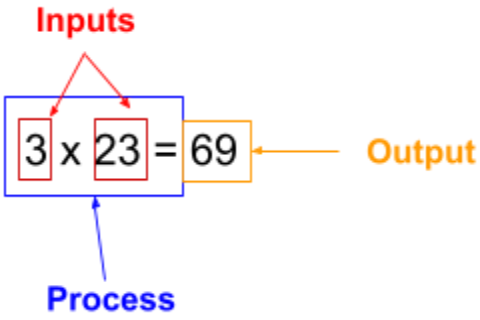
Almost if not everything that is *non-literal* in C Programming requires a variable , **especially** our inputs and outputs. **Variables** are a location in memory that servers as a *container* of values that we want to manipulate/process in our program. This means that whenever we declare a variable in our program, it takes up certain bytes of memory for us to use it.

ANALOGY: If I wanted to drink water, I'd get one glass from my cabinet of glasses then use it to contain the water that I am about to drink. Consider the *cabinet of glasses* as the memory in your computer / application, my *one glass* as your variable and the *water* I am pouring as my value. When I took a glass from the cabinet, I was taking "a space in memory" , and using it to store a value.

SYNTAX& DECLARATION

(Datatype) (VariableName) ;

Before you are able to use a variable and have it hold a value, you must first declare it. Imagine asking your classmate something about assignments but not even being sure that they're your classmate. So before you can do so, it must be established that you guys are classmates - that's the essence of declaring your variables first before using it.



In the context of our multiplication example. We are using two inputs, right? Therefore, we must declare two variables to serve as our multiplicand and multiplier. OP! Teka lang. We also have an output! So that also needs a variable. Therefore , we typically need 3 variables. They may be declared like this:

int multiplicand;

int multiplier; **or** **int multiplicand,multiplier,product;**

int product;

Wherein ***int*** is our data type and ***multiplicand, multiplier and product*** are the names of our variables. Later on in this document, we will be focusing on more data types but for now, let us settle with ***int*** which is short for integer since we are dealing with whole numbers.

NOTE:

- Most if not all statements in C must also end in a semicolon (;) it is basically the punctuation and it let's the computer know that that is the end of the specific line of code.

- C as a programming language is **CASE SENSITIVE**, meaning how you typed your variable name, and datatype matters. ***int*** must always be in lowercase, else you will end up with a syntax error.
    - The casing of your variable name also matters. int multiplicand and int Multiplicand , will result in having two different variable names, one having a capital letter at the start and the other having all lower case.

NAMING CONVENTION

- Variable names may only be composed of letters,numbers and underscore ('_')
- **BUT** a variable name cannot start with a number
- Instead, it must start with either a letter or an underscore (although it is recommended to start with a letter rather than the latter)
- There are no spaces allowed when naming a variable.
- If your variable name is composed of two words, you may use the *camel notation (e.g. int firstNum , int SecondNum , int theResult)* or separate the two words using an underscore *(e.g. int first_num, int second_num, int The_Result)*
- It is ideal to name your variable based on what value it holds. Hence why the variables for multiplication were named multiplicand and multiplier. But you may use whatever variable name you are comfortable with.
- You cannot name a variable if a certain word is a ***RESERVED WORD or KEYWORD*** such as but not limited to - auto,break,case,char,const,continue,do,double,default,entry,else,enum,float,for,goto,long,int,if,register,return,static,short,signed,sizeof,struct,switch,typedef,union,void,while,volatile,unsigned

REVISITING OUR MULTIPLICATION PSEUDOCODE

Earlier, we established how to create a Pseudocode and that we will be refining it to look more similar to a programming statement rather than your typical English Language.

1. Input two numbers
2. Multiply the two numbers
3. Show the result of multiplying

We also established that we must always declare our variables before using them. And so we will now revise our pseudocode to look more like this.

1. **int multiplicand, multiplier, product ;**
2. **Input multiplicand**
3. **Input multiplier**
4. **Get the value of product by multiplying multiplicand and multiplier**
5. **Show the value of our product**

You'll notice that I have separated the inputting of values for multiplicand and multiplier. You may be tempted to just put those two into one line since they're doing the same process - which is more than fine. But for the sake of easier translating into programming statements, I have separated these two. But if you chose to do "**input multiplicand and multiplier"** then that's fine for now as a pseudocode.

3.4 DATATYPES

From the name itself, a **Datatype** is used to identify the type of data/value that a variable is designated for. For instance, in an asian household, there is a cup designated for scooping rice (a rice cup , if you will) and there is a cup intended for drinking. Both are cups but they are different in their type/classification. As we proceed, we will encounter N number of datatypes but for now, let us focus on three : integers (int) , floating type (float) and character (char),

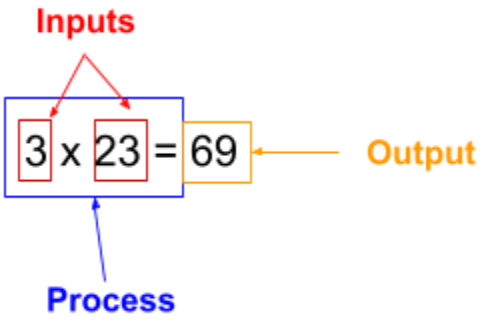
NOTE:

- The datatype of a variable must always be declared before its variable name, otherwise an error message will be shown.

****Insert Hierarchy of Datatype here

INTEGER (INT)

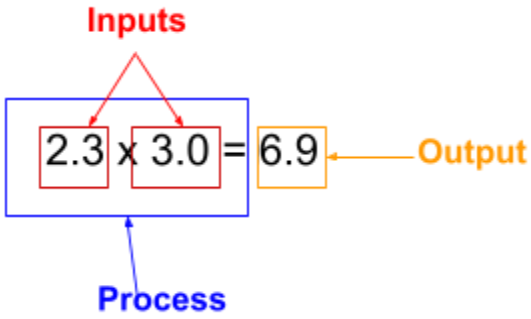
In mathematics, an integer refers to a negative or positive whole number and the number zero (0). The same applies in programming.



Earlier, we established that the variables that would hold our multiplicand, multiplier and product would be of an integer (int) datatype - this is because the numbers that we are using are whole numbers. If we were to assign a decimal value into an integer variable then we would encounter *an error of conflicting types* which means the type of value we want to assign a variable does not coincide with the data type of our said variable.

FLOATING TYPE (F L O A T)

If whole numbers are represented through an integer type, then decimal numbers may be represented through a floating type A.K.A. the ***float*** datatype. If we wanted to multiply decimals instead of fractions. For instance:

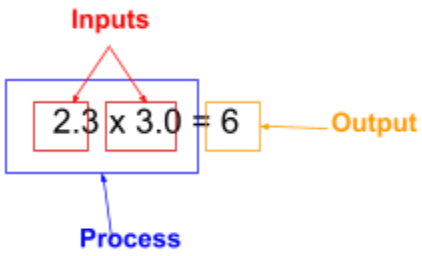


In this example, we are multiplying decimal factors with a product that is also a decimal. So our pseudocode would still be the same in certain lines - it would only differ in the declaration of our variables.

1. **float multiplicand,multiplier,product;**
2. **Input multiplicand**
3. **Input multiplier**
4. **Get the value of product by multiplying our two factors**
5. **Display the value of product**

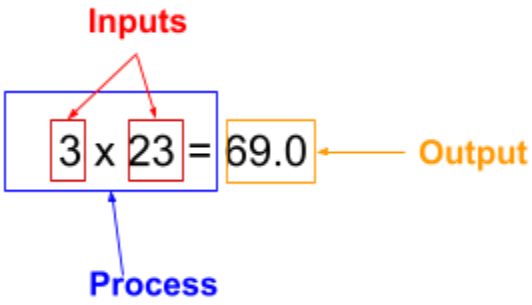
WHAT IF:

1. The variables of our multiplicand and multiplier are ***float*** but our product is an ***int*** datatype?



In that case, the decimal value of our product (which is .9) will be omitted. Meaning it will not be included when we want to display the value of our product

2. How about if our multiplicand and multiplier are both ***int*** but our product is a ***float***?



Then, our product would still hold the same value but with a decimal of .0 since it is holding a whole number value but represented as a decimal

CHARACTER (CHAR)

This is a datatype intended for holding a single character from a number,letter or special symbols. Values being held by a character variable are always enclosed in single quotes (‘’).

For example, let us make a pseudocode where we ask the user to input one character variable and have it be displayed. It would probably look something like:

1. **char var1;**
2. **Input var1**
3. **Display var1**

It would look something like this. Of course, you may use your own variable names and phrasing as long as the essence is similar.

3.5 DISPLAYING VALUES (PRINTF)

In C, there are a few ways for you to be able to display certain values/messages in your program - the most common is through the use of a function called ***printf().***

printf() is a function that can be found in the *C Library* <*stdio.h*> (more on this later) which is used to display a formatted output - this means that whatever you put inside the printf() and however you format is , will reflect in the output to be printed from it.

DECLARATION

printf(format, ...);

Wherein **format** refers to whatever statement you put within this parameter. For instance, if I wanted to display the literal value “Sapnu Puas” then my printf statement would look something like this.

printf(“sapnu puas”);

As you can see, the printf() function can accommodate 1 or N number of parameters, as denoted by the ellipsis ('…'). This will come in handy when we want to display the value of a variable and not just a typical sentence (A.K.A. string literal).

CONSIDER THIS...

Assume that the variable **val1** is of an integer type and holds the value 69. (As shown below - don't worry we'll discuss this statement later on)

```
int val1 = 69;
```

Using the ***printf() ,*** what code or statement will let us display the value being contained in the variable ***val1***? You may be tempted to do a statement like:

```
printf("val1");     or    printf("69");
```

In which the statement on the left is wrong bec it doesn't print the value of val1 but rather, it prints the literal word ***val1***. The statement on the right definitely prints the number 69 but fails to comply with our goal of printing it from the given variable ***val1.*** For us to print the value being contained by the variable ***val1*** , we will need to utilize a ***format specifier*** to display the value of our variable. It should look something like this:

```
printf("The value of val1 is : %d",val1);
```

Where in words "The value of val1 is: " is a sentence that I, as the creator, decided to use on a whim and **%d** being our format specifier , telling the program that we must display the value of a variable that is of integer type. The further we go into programming, there will be more datatypes to be encountered, meaning more format specifiers. But for now let's focus on the 3 datatypes above.

FORMAT SPECIFIERS

DATATYPE	FORMAT SPECIFIER
Integer (int)	%d-display an integer in base 10 %o - display an integer in base 8 %x - display an integer in base 16

Floating type (float)	%f -displays a floating-point number in base10 (decimal)
Character (char)	%c - to display a character

REVISITING OUR MULTIPLICATION PSEUDOCODE

1. **int multiplicand, multiplier, product ;**
2. **Input multiplicand**
3. **Input multiplier**
4. **Get the value of product by multiplying multiplicand and multiplier**
5. **Show the value of our product**

Now that we know how to use our printf() to display the values of a variable, we may not revise line #5 ! Instead of simply saying "Show the value of our product" we can now make it into a C statement :

1. **int multiplicand, multiplier, product ;**
2. **Input multiplicand**
3. **Input multiplier**
4. **Get the value of product by multiplying multiplicand and multiplier**
5. **printf("The product of %d x %d = %d",multiplicand,multiplier,product);**

You'll notice that there are 3 format identifiers in one printf() statement, and you may be wondering which values get displayed per %d . That will depend on the arrangement of our variables located after the **",** . Since the order is **multiplicand,multiplier,product** then the values will be displayed in that sequence as well. Making the statement **"The product of 3 x 23 = 69"**.

3.6 GIVING VALUES

Earlier when discussing printf() , you saw this image ... which was a cue to you, the reader, that values can be assigned to a variable through the assignment operator (=)

```
int val1 = 69;
```

An assignment operation is also the most basic C statement for performing a computation. The general syntax is **variable = value;** wherein the equal sign (=) is the assignment operator and **value** which is one the right can be a literal value or an expression (including arithmetic expressions).

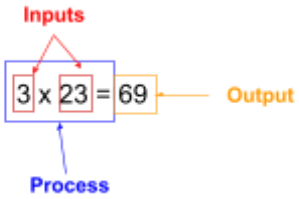
REVISITING OUR MULTIPLICATION PSEUDOCODE

1. **int multiplicand, multiplier, product ;**
2. **Input multiplicand**
3. **Input multiplier**
4. **Get the value of product by multiplying multiplicand and multiplier**
5. **printf("The product of %d x %d = %d",multiplicand,multiplier,product);**

Since we can assign a variable with an arithmetic expression, we can allow our variable named **product** to hold the arithmetic expression of **multiplicand x multiplier.** But in C, the multiplication operator is known as an asterisk (*). So, line #4 will now look something like this :

1. **int multiplicand, multiplier,product ;**
2. **Input multiplicand**
3. **Input multiplier**
4. **product = multiplicand * multiplier ;**
5. **printf("The product of %d * %d = %d",multiplicand,multiplier,product);**

But, in our multiplication example we know that our multiplicand has a value of 3 and our multiplier has a value of 23.



So we can also change lines 2-3 to be an assignment statement.

1. **int multiplicand, multiplier,product ;**
2. **multiplicand = 3;**
3. **multiplier = 23;**
4. **product = multiplicand * multiplier ;**
5. **printf("The product of %d * %d = %d",multiplicand,multiplier,product);**

But what if we are not sure of the value we want to multiply? What if we want the users to input any number they want? This is where we'll be using another function called ***scanf()***.

SCANF()

Just as the printf() function displays a copy of the value stored inside a variable, the scanf() function allows the user to enter a value at the screen. In other words it allows users to enter data into a program while it is executing.

And just like the printf(), a scanf() requires a control string A.K.A. a format identifier for it to know the type of data being inputted. The syntax being **scanf("format identifier",address of the variable);** wherein the format identifier is similar to the one being used in printf() statements and the address of the variable is denoted by an ampersand (&) followed by the name of the variable.

For example, let's say I want a pseudocode where I ask the user to input one float value and have it be displayed after. It might look something like this.

1. **float num;**
2. **scanf("%f",&num);**
3. **printf("The value of num is %f",num);**

When the statement **scanf("%f",&num);** is encountered, the computer stops program execution and continuously scans for an input from the keyboard.When a data is typed, the scanf function stores it using the address it was given. The program then continues the execution with the next statement after the call to scanf().

Previously, we defined variables as a **location in memory** , so for us to store the values into a variable, we must know the location that we wanna use. This means we need the address to that said location. Hence we have the 2nd parameter of our scanf() be **&num** which means *the address of variable num*.

ANALOGY: In order for you to know the location of your friend's place, you must first know the address. Only then will you be able to visit him/her.

REVISITING OUR MULTIPLICATION PSEUDOCODE

1. **int multiplicand, multiplier,product ;**
2. **Input multiplicand**
3. **Input multiplier**
4. **product = multiplicand * multiplier ;**
5. **printf("The product of %d * %d = %d",multiplicand,multiplier,product);**

Now that we know how to allow a user to input values, we may now revise our lines 2-3!

1. **int multiplicand, multiplier,product ;**
2. **scanf("%d",&multiplicand);**
3. **scanf("%d",&multiplier);**
4. **product = multiplicand * multiplier ;**
5. **printf("The product of %d * %d = %d",multiplicand,multiplier,product);**

Now that we've made our pseudocode resemble a C program more and more. I think it's time to make this into a legit program!

3.7 MAKING OUR MULTIPLICATION PROGRAM

Given our newly revised Pseudocode, we are a few steps away from making our very own program from it. But first we must put these line of codes inside a *function.*

1. **int multiplicand, multiplier,product ;**
2. **scanf("%d",&multiplicand);**
3. **scanf("%d",&multiplier);**
4. **product = multiplicand * multiplier ;**
5. **printf("The product of %d * %d = %d",multiplicand,multiplier,product);**

WHAT IS A FUNCTION

For those without any prior knowledge, you can think of functions as a "room where certain operations take place". In this case, our main room for our program to operate/work in called **main().**
In C programming, most lines of codes required can be found within a specific function. Functions may be predefined (just like our scanf() and printf()) they

may also be user defined, meaning a user can make their own function. And in general , a function (esp if user-defined) may call another function within it's line of codes. But we can talk more on this in later sections.

The ***main() function*** is also known as driver function because in every C program, there will always be a main function which serves as the "control room" for your program. It is denoted by either **int main()** or **void main()**. (more on this in later discussions). It is then followed up with an open and close curly braces ({}) wherein the line of codes will reside within the curly braces. It should look something like this :

```
void main(){
        /*insert code here*/
}
```

REVISITING OUR MULTIPLICATION PSEUDOCODE

With main() function in mind, and how it should contain our line of codes made from our Pseudocode, it should look something like this:

```
int main(){
int multiplicand, multiplier,product ;
scanf("%d",&multiplicand);
scanf("%d",&multiplier);
product = multiplicand * multiplier ;
printf("The product of %d * %d = %d",multiplicand,multiplier,product);
return 0;
}
```

Wherein **return** is a keyword that signifies the end of the function and program and that when it ends, the program will return the value of 0. If you chose to use **void main()** it should be something like this:

```
void main(){
        int multiplicand, multiplier,product;
        scanf("%d",&multiplicand);
        scanf("%d",&multiplier);
        product = multiplicand * multiplier;
        printf("The product of %d * %d = %d",multiplicand,multiplier,product);

}
```

C LIBRARY

By now, you're well aware that C has several predefined functions such as printf() and scanf(), these first two predefined functions that we will be using a lot from Prog 1 to Algorithms and Complexities. But what we must also take note of is that these predefined functions all belong to a certain header file.**Header files** is a file in C with an extension of .h which contains special predefined function and macros that can be accessed and used by the programmer. In the case of our printf() and scanf(), these two functions belong to the C Header called stdio.h , which means **Standard Input Output**.

Remember how before using a variable, we must first declare it? Well, the same can be said for our predefined functions - before we are able to freely use them in our program, we must first declare the C header file where they belong. This can be done by typing **#include<stdio.h>**. Easy.

REVISITING OUR MULTIPLICATION PSEUDOCODE

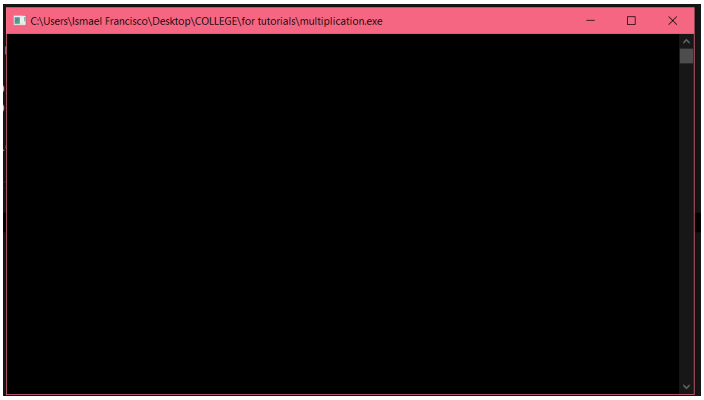
We're at the home stretch, ladies and gents and everyone in between! The declaration of our C header must always be the first thing we type in our program. Meaning before we even begin to declare our main() function and the codes within our main(), we must first declare our C header.It should now look like this:

```
#include<stdio.h>
int main(){
int multiplicand, multiplier,product ;
scanf("%d",&multiplicand);
scanf("%d",&multiplier);
product = multiplicand * multiplier ;
printf("The product of %d * %d = %d",multiplicand,multiplier,product);
return 0;
}
```

Now go onto your designation application for coding and try to run this program!

```c
#include<stdio.h>

int main(){
    int multiplicand, multiplier, product;

    scanf("%d",&multiplicand);
    scanf("%d",&multiplier);

    product = multiplicand * multiplier;

    printf("The product of %d * %d = %d",multiplicand,multiplier,product);
    return 0;
}
```

Now after running this program, you will be greeted by a an empty terminal with a blinking cursor.



You may think that this is an error, when in reality it is not. Just try to input one number, then press enter... Then input another number then press enter. It'll work. The reason why a blank cursor is what you see is because we , as programmers, have to make the program as user friendly as possible on our own. To any programmer, they would know that "ah this is where we input" but to someone who is new to this, it may be a tad bit confusing, so in order to make our program more user friendly, we can put some printf statements before every scanf to guide the users on what to input.

```c
#include<stdio.h>

int main(){
    int multiplicand, multiplier, product;

    printf("Enter a multiplicand: ");
    scanf("%d",&multiplicand);
    printf("\nEnter a multiplier: ");
    scanf("%d",&multiplier);

    product = multiplicand * multiplier;

    printf("\nThe product of %d * %d = %d",multiplicand,multiplier,product);
    return 0;
}
```

You'll notice that in the succeeding printf() statements, there is a backslash n (**n**) before the literal values that I've input. This \n is a special character that allows my literal text to be printed onto the next line.

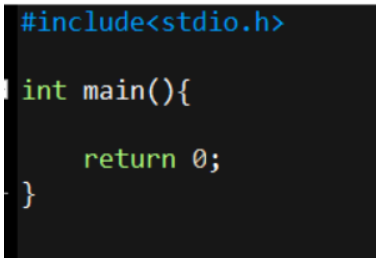
Y'know how you press enter on your google docs/text editor for it to go to the next line? That's the equivalent of what \n is for.

So now our command prompt when running the program will now look something like this:

```
Enter a multiplicand: 3

Enter a multiplier: 23

The product of 3 * 23 = 69
--------------------------------
Process exited after 3.043 seconds with return value 0
Press any key to continue . . .
```

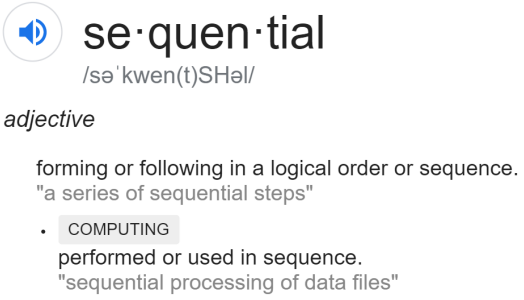
And with that, CONGRATS ON MAKING YOUR PSEUDOCODE INTO A PROGRAM! WE DID IT!

NOTE:

- In DevC, main() must be **int main()** not void main() for it to work properly. This is something not related to our programming fundamentals, this is more of a reminder when coding in DevC application
- Every c program when coded in Dev C must have this structure
  - 

3.8 TYPES OF PROGRAMS

SEQUENTIAL



Just as the name entails, a sequential program will always execute **the same instructions.** I personally call it a top-to-bottom program since whatever you code will be executed from top to bottom in a sequential program.

EXAMPLE:

You may not know this, but the multiplication program we just did is a good example of a Sequential Program.

```c
#include<stdio.h>

int main(){
    int multiplicand, multiplier, product;

    printf("Enter a multiplicand: ");
    scanf("%d",&multiplicand);
    printf("\nEnter a multiplier: ");
    scanf("%d",&multiplier);

    product = multiplicand * multiplier;

    printf("\nThe product of %d * %d = %d",multiplicand,multiplier,product);
    return 0;
}
```

In this example, the first line of code to be executed is the declaration of variables, followed by the printf() statement/s that let the user input a factor , then the scanf() that stores the user-input onto our variable. After that, our **product** variable will store the arithmetic expression's result and finally, display said result. This sequence will be constant within the given program unless a certain line of code is added, removed or rearranged.

CONDITIONAL

con·di·tion·al

/kənˈdiSH(ə)n(ə)l/

See definitions in:

All | Grammar | Philosophy

adjective

1. subject to one or more conditions or requirements being met; made or granted on certain terms.
   "the consortium has made a conditional offer"

   Similar: subject to | dependent on | depending on | contingent on

2. GRAMMAR
   (of a clause, phrase, conjunction, or verb form) expressing a condition.

If you've been doing other arithmetic programs, maybe division, you'll realize that the program won't always work. A concrete example of this would be the division by zero will result with an undefined value in mathematics, and it will also flag an error in programming. With this, we will come to the conclusion that certain programs will only work under certain conditions. That being said, a conditional program is best suited for such a scenario

The idea of a conditional program is that a segment of the program will only be executed if a set of conditions are met. Unlike sequential programs, the result in a conditional program will vary from whether or not the said conditions are met. For a conditional program to work, you will be needing a set of **conditional operators** , the most common being the **if()** operator. (More on this later) wherein the condition to be met will be inside of the parentheses. Our condition may either be an arithmetic expression, relational expression, or a hybrid of both and then some.

The syntax of an if() is as follows:

if(condition){
/*code to be executed only if the condition is met/is true*/
}

In the event that there is a code below this segment, then that code will be executed after the if condition is checked.

EXAMPLE:

For our division program (*if you're not that confident in coding things directly, start with a Pseudocode and work your way up*), it is natural to have whole numbers as our dividend and divisor but end up with a decimal value for a quotient. For instance, 4/2 = 2 but 2/4 = 0.5 , right? To accommodate this scenario, the datatypes for our variables will be of **floating type**. So our program might look like this:

```c
#include<stdio.h>
int main(){
    float dividend, divisor,quotient;

    printf("Enter a dividend: ");
    scanf("%f",&dividend);
    printf("\nEnter a divisor: ");
    scanf("%f",&divisor);

    quotient = dividend/divisor;

    printf("\nThe quotient of %f / %f = %f",dividend,divisor,quotient);
    return 0;
}
```

But of course, if we were to do a division of zero, the output would look like this:

```
Enter a dividend: 2

Enter a divisor: 0

The quotient of 2.000000 / 0.000000 = 1.#INF00
--------------------------------
Process exited after 2.765 seconds with return value 0
Press any key to continue . . .
```

And we want to give everyone a heads up when this happens! So , let's use our if() to denote that if our divisor is equal (==) to 0 then we will instead display a message that says "Error, a division by zero is happening" .

```c
#include<stdio.h>
int main(){
    float dividend, divisor,quotient;

    printf("Enter a dividend: ");
    scanf("%f",&dividend);
    printf("\nEnter a divisor: ");
    scanf("%f",&divisor);

    if(divisor==0){
        printf("\nWARNING: A division by 0 is happening\n");
    }

    quotient = dividend/divisor;

    printf("\nThe quotient of %f / %f = %f",dividend,divisor,quotient);
    return 0;
}
```

So that the next time we encounter a division by zero, the user will at least be warned of what is going on with the quotient's value.

```
Enter a dividend: 2

Enter a divisor: 0

WARNING: A division by 0 is happening

The quotient of 2.000000 / 0.000000 = 1.#INF00
--------------------------------
Process exited after 4.957 seconds with return value 0
Press any key to continue . . .
```

But if the division is not by 0. Then our output will be something like this:

```
Enter a dividend: 50

Enter a divisor: 5

The quotient of 50.000000 / 5.000000 = 10.000000
--------------------------------
Process exited after 3.877 seconds with return value 0
Press any key to continue . . .
```

But what if:

I don't want to compute the quotient when the divisor is 0? What if, I want the quotient to only be computed and displayed when the divisor is not equal to 0 , or else , I'll just flag a warning message about the division by 0?

Of course you could put another if() condition for when the divisor is not equal to 0 and your code will look something like this:

```c
#include<stdio.h>
int main(){
    float dividend, divisor,quotient;

    printf("Enter a dividend: ");
    scanf("%f",&dividend);
    printf("\nEnter a divisor: ");
    scanf("%f",&divisor);

    if(divisor==0){
        printf("\nWARNING: A division by 0 is happening\n");
    }

    if(divisor!=0){
            quotient = dividend/divisor;
            printf("\nThe quotient of %f / %f = %f",dividend,divisor,quotient);
    }

    return 0;
}
```

Which gets the job done, but it is not the most efficient means of doing so. Again, what we want our division program to do is - if the divisor is not equal to zero, then it will compute and display the quotient, or else, it will flag a message about the division by zero.

Now this is where the use of **other** conditional operators takes place.

OTHER CONDITIONAL OPERATORS

What we want our division program to do is - if the divisor is not equal to zero, then it will compute and display the quotient, or else, it will flag a

message about the division by zero. In our program only two outcomes are possible, right? And if the first condition isn't meant, (if the Divisor isn't 0) then it automatically means that the other outcome is to happen (wherein the quotient is to be computed and displayed).

For this scenario, we can use an ***if-else*** statement. It has a syntax like this:

```
if(condition){
        /*insert code here*/
}else{
        /*insert other code here*/
}
```

In which if the condition is met, only the codes within the first curly braces will be executed, the codes in the curly braces of **else** will be omitted. If the condition is not met (meaning it is false) the conditions within the first curly brace will be omitted, and the codes within the curly braces of **else** will be executed instead. Your code may look something like this:

```c
#include<stdio.h>
int main(){
    float dividend, divisor,quotient;

    printf("Enter a dividend: ");
    scanf("%f",&dividend);
    printf("\nEnter a divisor: ");
    scanf("%f",&divisor);

    if(divisor==0){
        printf("\nWARNING: A division by 0 is happening\n");
    }else{
            quotient = dividend/divisor;
            printf("\nThe quotient of %f / %f = %f",dividend,divisor,quotient);
    }



    return 0;
}
```

OR

```c
#include<stdio.h>
int main(){
    float dividend, divisor,quotient;

    printf("Enter a dividend: ");
    scanf("%f",&dividend);
    printf("\nEnter a divisor: ");
    scanf("%f",&divisor);

    if(divisor!=0){
            quotient = dividend/divisor;
            printf("\nThe quotient of %f / %f = %f",dividend,divisor,quotient);
    }else{
        printf("\nWARNING: A DIVISION BY ZERO IS HAPPENING");
    }



    return 0;
}
```

An ***if-else*** statement is our best bet when the program only requires two outcomes. But what if it requires 3 or more? Well, this is where we may use an **Else-if** statement. It is structured like this:

if(condition){
/*code to be executed if this condition is true*/
}else if(condition){
/* code that will be executed if this part is true*/
}else{
/*code that will be executed if the previous 2 conditions were not met*/
}

But as useful as this is, it becomes a liability if relied on for too long. Imagine having a good that looked this long.
if(){
}else if(){

}else if(){

}else if(){

}else if(){

}else{
}

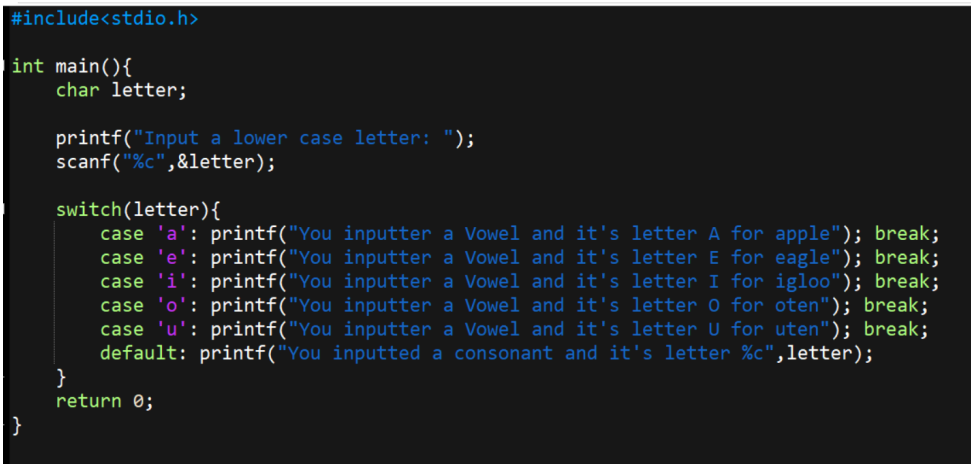
I don't know about you, but this looks unnecessarily long! So instead of this, we can use a **switch()** statement which looks something like this:

```
switch(variable){
        case constant: /*code to be executed*/ ;
                        break;
        default: /*code to be executed*/;
}
```

I know words aren't enough to explain the purpose of these things to let's give a concrete example.

EXAMPLE:

Let's say I wanna have a program that lets the user input a character in lower case. The program must determine which vowel letter it is, and if it isn't a vowel, it should display what consonant it is. With a switch statement, it'll prolly look like this.



```c
#include<stdio.h>

int main(){
    char letter;

    printf("Input a lower case letter: ");
    scanf("%c",&letter);

    switch(letter){
        case 'a': printf("You inputter a Vowel and it's letter A for apple"); break;
        case 'e': printf("You inputter a Vowel and it's letter E for eagle"); break;
        case 'i': printf("You inputter a Vowel and it's letter I for igloo"); break;
        case 'o': printf("You inputter a Vowel and it's letter O for oten"); break;
        case 'u': printf("You inputter a Vowel and it's letter U for uten"); break;
        default: printf("You inputted a consonant and it's letter %c",letter);
    }
    return 0;
}
```

If you'll notice , there is a keyword **break** in almost every case statement. This is to prevent the program from cascading and executing the rest of the cases.

For instance, without the break statement, and if I inputted the letter 'a' it would print all the printf statements from that case and downwards.

ITERATION

it·er·a·tion

/ˌidəˈrāSH(ə)n/

noun

the repetition of a process or utterance.

- repetition of a mathematical or computational procedure applied to the result of a previous application, typically as a means of obtaining successively closer approximations to the solution of a problem.
- a new version of a piece of computer hardware or software.
  plural noun: **iterations**

Imagine if I told you to print each number in their own printf() statement (*e.g. printf("1"); printf("2"); and so on).* Of course , if I asked you to print the numbers 1-5, it's a piece of cake. But what if I asked you to print from 1-100? It'd be a tad bit tedious right? When a certain task requires a **lot** of repetition, our best bet is to use an ***iteration statement*** or what we typically call ***loops***.

Iteration Statements or Loops will repeat a set of commands for N number of times for as long as the condition to execute said commands are satisfied. For instance:

```
#include<stdio.h>

int main(){
        int num = 1;

        while(num<=100){
                printf("\n%d",num);
                num++;
        }
        return 0;
}
```

Would be a good example of how to print the numbers 1-100 without having to manually code 100 printf() statements on our own. In order for our number to increase per iteration, we utilized a postfix increment operator. But a statement like **num=num+1;** will work just as fine. Now just like conditional statements, there are many ways to create loops. What you have just witnessed is what we call a ***while loop.*** It is typically structured like this:

```
initialization;
while(condition){
        /* line of code*/
}
```

OTHER FORMS OF LOOPS

With a **while() loop**, the checking of the condition comes first and the execution of the code (if the condition is met) comes after. Therefore, the condition is checked N+1 times while the statements are executed N times. In the case of our example earlier, the printf() executed 100 times while the condition was checked 1=1 times - with the 10st checking of the condition being the one that stops the loop, because by then, the variable **num** will have the value of 101 , which does not meet the condition of **num<=100.** There exists a looping method which is the exact opposite of our while() loop - the **do while()** loop. Which looks something like this:

```
initialization;

do{
        /*code to be executed*/
}while(condition);
```

In this scenario, the code is executed N times while the condition is checked N-1 - or to mirror the previous analogy , this time, the code is executed N+1 while the code is checked N number of times. With the **right/same conditions** , do while () and while() will execute the same outputs.

By now you've probably noticed a typical structure in a loop - there is a variable which we use to determine whether a loop is to be executed or not, and typically an arithmetic expression that alters said variable ... or in this case our increment operator. If the structure of our **while() loop** looks clunky, you can use a **for() loop** which works just like a **while loop** but just more organized to look at. It should look something like this:

```
for(initialization ; condition ; expression){
        /*code to be executed*/
}
```

Now let's take a quick look-back on that code we just did earlier.

```
#include<stdio.h>

int main(){
	int num = 1;

	while(num<=100){
		printf("\n%d",num);
		num++;
	}
	return 0;
}
```

Can now look like this…

```
#include<stdio.h>

int main(){
	int num;

	for(num=1;num<=100;num++){
		printf("%d",num);
	}
	return 0;
}
```

For anyone who is slightly confused about the sequence of how this'll go down, it'll be something like this :

```
for( 1st to work ; 2nd to be done --- then 5th ; 4th to be executed --- 7th){
	3rd to be executed --- 6th to be done
}
```

With this in mind, the initialization will only be encountered once.

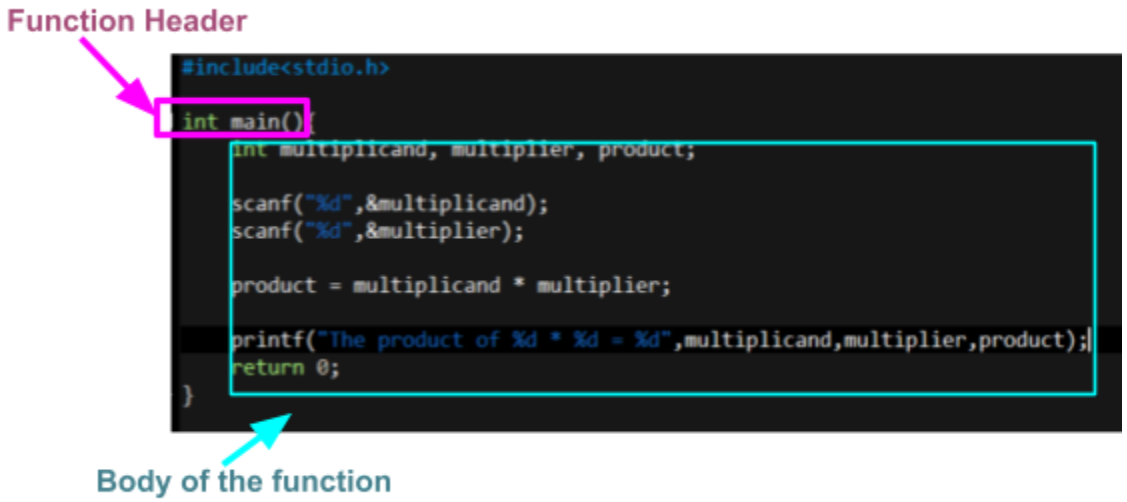
3.9 FUNCTIONS

In C Programming, most of everything you will be doing involves functions. Functions are a block of code intended to perform a specific task. C has several predefined functions that we, as programmers, can use. For instance, scanf() and printf() are staple functions that we will be using from Programming 1 , Programming 2, Data Structures

and onwards. But not only can you use these predefined functions, you can make your own - more on that later. For now, let's dissect the INs and OUTs of making a function.

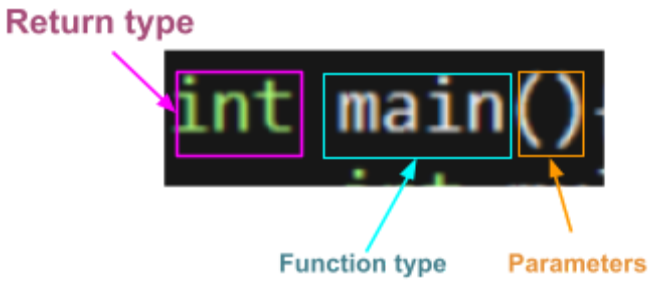
COMPONENTS

There are two major components involved: The Function Header and the Body of the Function .



FUNCTION HEADER

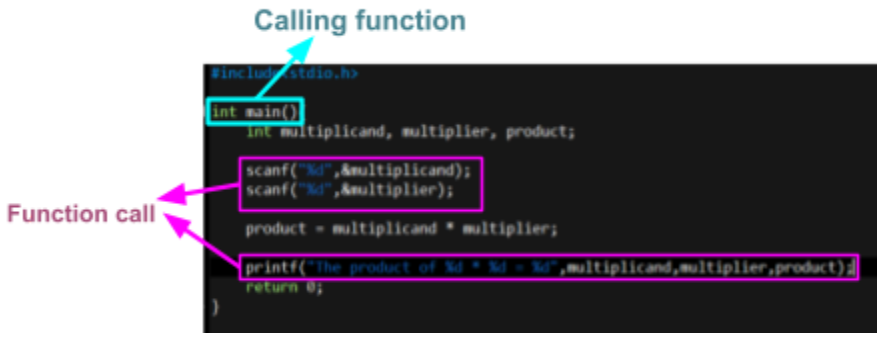
The function header includes the name of the **function** and tells us (and the compiler) what type of data it expects to receive (the parameters) and the type of data it will return (return value type) to the calling **function** or program.



For this example, there are no parameters within our main function, but if there was, a **function parameter** with one parameter should consist of **a datatype and the variable name.** We will be seeing an example of a function header with a parameter soon.

NOTE:

- When we use scanf() and printf() in our main() function, this is an example of a **function call** - wherein our **main()** is the **calling function()** since it is the function where the function call to scanf() and printf() happens.



- If we look at the syntax of scanf() it is `int scanf(const char *format, ...)` which means that scanf() is a function that returns a value to its **calling function**. Which is why when we call scanf() whatever is being inputted through the keyboard is returned and stored in a variable within the calling function which in this case is **main()**
- The syntax of most predefined library functions (like scanf()) are usually a form of **function header** so it explains to you what parameters are accepted and if it returns something.

BODY OF THE FUNCTION

```
{
    int multiplicand, multiplier, product;

    scanf("%d",&multiplicand);
    scanf("%d",&multiplier);

    product = multiplicand * multiplier;

    printf("The product of %d * %d = %d",multiplicand,multiplier,product);
    return 0;
}
```

The body of the function starts after the open curly brace (**{**) and ends before the close curly brace (**}**). In this example, you will notice that the

body of the function ends with a **return 0;** , this is because our function header has a return type of **int.** If our return type was **void**, then we wouldn't be needing this **return 0;** statement.

MAKING OUR OWN FUNCTION

As of the moment, our function call for **scanf()** is within main(), but what if I wanna make a function that uses just **one scanf()** rather than the two scanf()'s that we're using in our multiplication program?

Let us make a function that accepts allows a user to input 1 number that shall be returned to its calling function.

So where do we begin? How about with our function header! Let's start with a name. It is important that the function name conveys what the function is supposed to do - and since the function we'll make is all about inputting numbers , how about making the function name **inputNum**. Of course you have the freedom to name it something else.

Now for the return type. Do the instructions require something to be returned? YES. A number, and in C, numbers can either be of int or float datatype. But for now, let's stick with returning an integer. Therefore, the return type of our function header is **int.**

So right now, our function header looks something like this:

```
int inputNum()
```

Now all that's left is to determine if we need parameters or not. In this case, since we aren't passing anything when we do a function call, there is no need for any parameters.

Now that we have our function header, it's time to work on our function body. Now the instructions require us to be able to input 1 number and return it to the calling function. This means that a variable must store the value being inputted by the user. Therefore, your code should look something like this:

```
int num;
printf("Enter a number:");
scanf("%d",&num);
return num;
```

So our own function looks something like this:

```
int inputNum() {
        int num;

        printf(“Enter a number:”);
        scanf(“%d”,&num);
        return num;
}
```

REVISITING OUR MULTIPLICATION PROGRAM

Now that we have our **inputNum()** function, we can replace the printf() and scanf() in our main() with a function call to inputNum(). Our program should look something like this:

```
1  #include<stdio.h>
2
3  int inputNum();
4
5  int main(){
6      int multiplicand, multiplier, product;
7
8      multiplicand = inputNum();
9      multiplier = inputNum();
10     product = multiplicand * multiplier;
11
12     printf("\nThe product of %d * %d = %d",multiplicand,multiplier,product);
13     return 0;
14 }
15
16 int inputNum(){
17     int num;
18
19     printf("\nInput a number: ");
20     scanf("%d",&num);
21
22     return num;
23 }
```

Now you’ll notice that in line 3 , there is a declaration of our function header but it ends with a semicolon (;) . Before I explain further let’s understand how C programming works:

- C programming is sequential programming, meaning your code will be read from top to bottom. Hence why we always declare variables before using them.

That being said, if we didn't have the line of code in line 3, and when the program runs through line 8 and 9, it will not recognize inputNum() as a function since the function call appeared first in the program before the function itself. (In other words , main() appears first and inputNum() function follows after)

But what if we just rearrange our code to look this like?

```
1  #include<stdio.h>
2
3  int inputNum(){
4      int num;
5
6      printf("\nInput a number: ");
7      scanf("%d",&num);
8
9      return num;
10 }
11
12 int main(){
13     int multiplicand, multiplier, product;
14
15     multiplicand = inputNum();
16     multiplier = inputNum();
17     product = multiplicand * multiplier;
18
19     printf("\nThe product of %d * %d = %d",multiplicand,multiplier,product);
20     return 0;
21 }
22
23
```

And by all means, this will solve our problem of having the function be first to be *read* by the program so that the function call to it will be recognized later on.

But, the coding convention that we follow in USC requires us to always have the main() function as the first function we see in our program.

```c
#include<stdio.h>

int inputNum();

int main(){
    int multiplicand, multiplier, product;

    multiplicand = inputNum();
    multiplier = inputNum();
    product = multiplicand * multiplier;

    printf("\nThe product of %d * %d = %d",multiplicand,multiplier,product);
    return 0;
}

int inputNum(){
    int num;

    printf("\nInput a number: ");
    scanf("%d",&num);

    return num;
}
```

That being said, line 3 will *give the program a heads up on what user-defined functions* we have so that when we reach line 8 and 9, it will work just fine. Think of it like this : **as a student, you wouldn't know that these facilities and activities are offered to students unless someone/something informed you, right?** That's the basic gist of why we need line 3 code for this coding convention.

If you'll also look at line 22, our inputNum() function is returning something to the calling function , in this case main(). So in order to catch the value being returned, we used the assignment operator (=) and to be able to store the returned value into a variable.

SIMULATING OUR MULTIPLICATION PROGRAM

Please click this link for you to be redirected to the google slides that shows a simulation w/ an explanation on how the multiplication program works.
https://docs.google.com/presentation/d/1uEMDBy8_txAsNG6toJ421TvIsjEdNH9NxUj_o8nwCh4/edit?usp=sharing

3.A. POINTERS

Now before we even begin to explain the usage and relevance of a pointer. Let's take a step back and see how functions work.

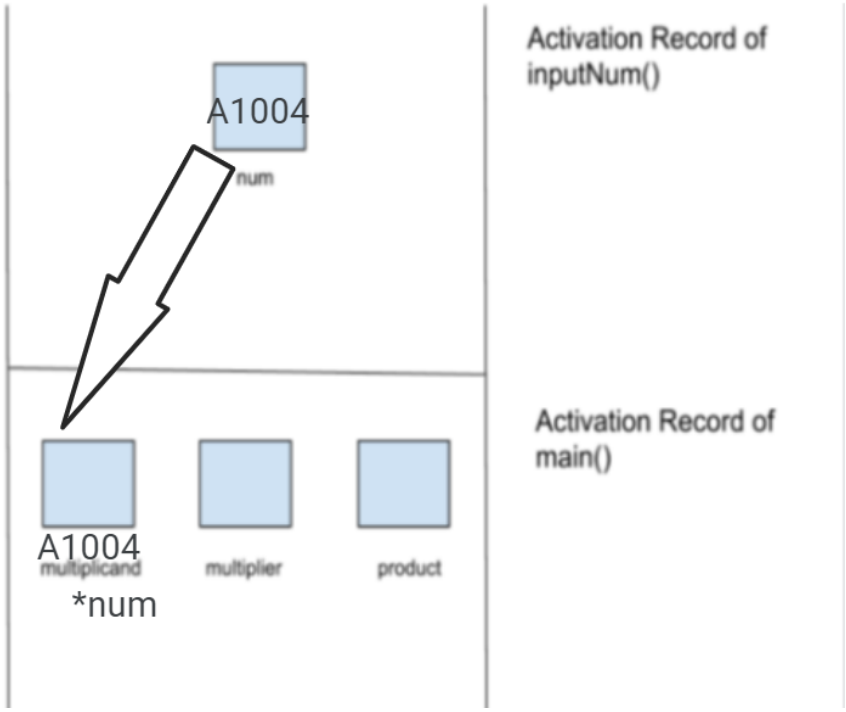
```c
#include<stdio.h>

int inputNum();

int main(){
    int multiplicand, multiplier, product;

    multiplicand = inputNum();
    multiplier = inputNum();
    product = multiplicand * multiplier;

    printf("\nThe product of %d * %d = %d",multiplicand,multiplier,product);
    return 0;
}

int inputNum(){
    int num;

    printf("\nInput a number: ");
    scanf("%d",&num);

    return num;
}
```

In the program above, we have a function that returns whatever number the user inputs. So, we can say it is a function that returns a value. But what if our function looked like this?

But what if our program looked more like this?

```c
#include<stdio.h>

void inputNum(int * num);

int main(){

    int multiplicand, multiplier, product;

    inputNum(&multiplicand);
    inputNum(&multiplier);
    product = multiplicand*multiplier;

    printf("\nThe product of %d * %d = %d",multiplicand,multiplier,produc
    return 0;
}

void inputNum(int * num){
    printf("\nInput a number: ");
    scanf("%d",&(*num));
}
```

It's still an inputNum() function, **but** we are passing the address of the variables **multiplicand and multiplier** so its activation record looks something like this



In this scenario **num is our pointer variable**

What pointers do is they hold the address of another certain variable.

Take this for example:

int number = 69;
int * ptr;

Number is our integer variable which holds the value 69 and ptr is our integer pointer. Therefore, the syntax to declaring a pointer is

[datatype] * [pointerName] ;