Engineer's Introduction to C Programming

Ginger Gengar

20th October 2022

Contents

0.1 Getting Started

Why bother learning C? C seems increidbly hard at first compared to other "higher-level" languages like Python but C is much much "closer" to the machine than Python would be. C is a low-level language, which means that it doesn't take too much translations to convert C code into machine instructions. This makes C incredibly fast and also gives the programmer low-level control over the hardware. This gives C a few important applications:

1. Mathematical Libraries: These applications are typically used in the context of massive simulations in many engineering disciplines like structures and fluid mechanics which are often very very large projects. The fundamental mathematical libraries like linear algebra matrix decompositions or finding eigenvalue-eigenvectors of large matrices need to be done as fast as possible because any incremental time savings would save hours of computation and alot of money in terms of electricity and computational costs.

2. Physics Solvers: We are talking about all kinds of problems like computational fluid dynamics and finite element analysis where we are no longer just interested in single-threaded programs but parallel programs. C gives alot of control on building parallel programs and makes it possible to write computer programs that can take advantage of large servers with thousands of threads. This is important because all of those problems are typically incredibly rescource intensive and an efficient language like C helps with improving our capabilities in solving these problems.

3. Embedded Devices: C gives us low-level control over computer rescources and is also a very computationally efficient language making it perfect for small devices that don't have alot of computational power and also need to access very low-level hardware like actuators and servos. This makes C the perfect language for robotics and its various applications.

4. Computer Graphics: Alot of things like rotation, ray-tracing in computer animations are extremely rescource intensive. Using a language like C allows the applications to run faster and smoother.

The list given above is not exhaustive: there are still many great applications out there for C. Those are just the ones that I happen to be aware about C. The point is that C is a fast language and gives us low-level control, so it will be relevant in applications where speed is important or interfacing to hardware is important.

0.1.1 Hello_World Program

This is arguably the simplest C program that anyone can write. This is typically the very first program most people would start writing. We have a declaration of the main function which is what C would execute. We have a statement `#include <stdio.h>` which is basically a way to use libraries. In our case, we are using the standard input output library which allows us to print things into the terminal console. Inside this main function, we have a simple command to print "Hello World!". The n character means newline which is skip a line.

```c
//The standard input output library is what gives you capabilities of printing
//Like printf. Printf accepts a string and some datatypes
#include <stdio.h>

//Welcome to a great adventure!
//Here you will learn your fisrst stepping stones in mastering the C Programming language
//C is widely used in many applications like scientific computing, Computational Sciences
//and engineering Dynamics. It is considered one of the best programming languages in performance
//You should be proud to learn C

//This is the a main function which is the start of your program
int main(void) {

```

```
//This is how you print to the terminal console
printf("Hello World!\n");

//This tells that the main function has executed succesfully
return 0;}
```

To run a C program, we need to first compile the program. We can use gcc which is a famous C compiler for that. Let us try compiling our file which is named Hello_World.c ,

```
gcc Hello_World.c
```

After we have succesfully compiled our program, we need to be able to run the program. To do that, we just have to call the produced library a.out,

```
./a.out
```

The result of running the C program above is shown below,

```
Hello World!
```

0.1.2 Printing Integers

After printing text, the next phase is to print numbers. In C an many other popular programming languages, there is a distinction between integers and floating points. When we print an integer, like 3 in this case, we need to use what is called a format specifier. In the example below, %d is a format specifier. It tells C where the next input argument should be inserted into the string. So we have the line printf("Printing an integer: %d\n", 3);. The first input argument to printf is the string "Printing an integer: %d\n". When C encounters the format specifier %d, C would look at the next input argument to printf which is 3. The next input argument 3 is indeed an integer and so C would print the number 3 exactly at the location of the format specifier %d. This means that the program would print something like "Printing an integer: 3" followed by a newline character.

```
//Give us printing capabilities
#include <stdio.h>

//This is a simple demonstration on how to print an integer

//This is the entry point to our program
int main(void){

//Print the integer 3
printf("Printing an integer: %d\n", 3);

return 0;}
```

The result of running the C program is shown below,

```
Printing an integer: 3
```

Now we know how to print integers using the format specifier %d. We had to be explicit and verbose in telling C we want to print an integer. Why don't you try changing the integer number 3 in the example above into a floating number 3.0 and see what C would print out.

Printing Floating Numbers

We are now going to demonstrate that we can print a floating number using the %lf format specifier. We can also print multiple integers and floating numbers all in a single call to printf. printf is what is known as a "variadic" function which means that it can take different numbers of input arguments.

Sometimes `printf` can take 1 input argument, other times 2 or 10, and it will work just as fine. From the example below, we should be able to tell how the format specifiers tell specifically where the next input arguments to `printf` would occur inside the string.

```c
//Give us printing capabilities
#include <stdio.h>

int main(void){

//Print the floating number 4.2
printf("Printing a Floating Number: %lf\n", 4.2);

//Printing multiple integers and floating numbers
printf("We are printing %d numbers, which is %lf and %lf\n", 2, 3.1, 8.2);

return 0;}
```

The result of running the `C` program above is shown below

```
Printing a Floating Number: 4.200000
We are printing 2 numbers, which is 3.100000 and 8.200000
```

0.1.3 Pre-Processed Directives

```c
//Give us printing capabilities
#include <stdio.h>

//This is a pre-processed directive
//Each time the pre-processor sees Num, it will literally change it with "3"
//This Num keyword though, it needs to be known before you compile
//You cannot change the value of "Num" when the code is running
//For something that you can change its value whilst its running, you need variables
#define Num 3

//This is the entry point to our program
int main(void){

//Print the integer 3
printf("Printing an integer: %d\n", Num);

return 0;}
```

The result of running the `C` program is shown below,

```
Printing an integer: 3
```

0.1.4 Variable Declarations & Initializations

```c
#include <stdio.h>

int main(void){

//This is just a declaration that an integer would exist.
//Right now, the integer "a" still contains garbage from previous random memory
//It is still not ready to be used. This line here is called a declaration
int a;

//Let us see what happens if we try to use the variable "a" before we initialize it
printf("The value of A after declaration but before initialization: %d\n", a);
//This line is called an initialization. It sets the value of a to some initial value for the proram
//Now a contains 1 and is ready to be used.
a = 1;

//This will print the value contained in a and should print 1
printf("The value of A after initialization: %d\n", a);

//The above is a bit tedious. We can declare and initialize in the same line.
//That is what we are trying to achieve by the following,
//We declare an integer b and initialize its value to 1
int b = 3;

//After the line above, b is already ready to be used so we can print its value and it
//should print the value 3 correclty
printf("The value of B after declaration and initialization in  a single line: %d\n", b);

return 0;}
```

The result of running the C program above is shown below,

```
The value of A after declaration but before initialization: 1369696944
The value of A after initialization: 1
The value of B after declaration and initialization in  a single line: 3
```

0.1.5 Declaring and Using Functions

```
//Gives us printing capabilities
#include <stdio.h>

//the "int" keyword in the front specifices what dataype this function will output
//The "add" keyword is what the function is going to be called
//the "int num1" means that the function's first input is an integer named num1
//the "int num2" means that the function's second input is an integer named num2
int add (int num1, int num2){
    //Declare a local variable named ans
    //Set thte value of ans to num1 added to num2
    int ans = num1 + num2;
    //This function will quit and "give back" its caller
    //the value inside local variable "ans"
    return ans;}

//This is the main function, entry point to the program
int main(void) {

//Declare an integer that has a value of 2
int a = 2;

//Declare an integer that has a value of 3
int b = 3;

//Decare an integer and initialize its value to 0
int c = 0;

//Show us the value of c which should be zero
printf("Value of C before add function: %d\n", c);

//Set the integer c to the output of a function named "add"
c = add(a,b);

//SHow us the value of c which should be 5
printf("Value of C after add function: %d\n", c);

return 0;}
```

The result of running the C program above is shown below,

```
Value of C before add function: 0
Value of C after add function: 5
```

0.2 Global Variables

0.2.1 Purely Reading Global Variables

```c
#include <stdio.h>

//This is a declaration-initialization of a global variable
//This global variable is an integer named kitty and contains the value of 3
//A global variable's value can be read by all functions in a program
int kitty = 3;

//Some arbitrary function that just prints out the value of kitty, a global variable
void function1 (){
    printf("The value of \"kitty\" inside function1: %d\n", kitty);
    return;}

//Some arbitrary function that just prints out the value of kitty, a global variable
void function2 (){
    printf("The value of \"kitty\" inside function2: %d\n", kitty);
    return;}

//Some arbitrary function that just prints out the value of kitty, a global variable
void function3 (){
    printf("The value of \"kitty\" inside function3: %d\n", kitty);
    return;}

int main(void){

//See whether Main can see the variable kitty and what main will see the value of kitty as
printf("The value of \"kitty\" inside main: %d\n", kitty);

//Call arbitrary function 1
function1();

//Call arbitrary function 2
function2();

//Call arbitrary function 3
function3();

return 0;}
```

The result of running the C program above is shown below,

```
The value of "kitty" inside main: 3
The value of "kitty" inside function1: 3
The value of "kitty" inside function2: 3
The value of "kitty" inside function3: 3
```

0.2.2 Reading and Writing to Global Variables

```c
#include <stdio.h>

//This is a declaration-initialization of a global variable
//This global variable is an integer named kitty and contains the value of 3
```

```c
//A global variable's value can be read by all functions in a program
int kitty = 3;

//Arbitrary function 1
void function1 (){
    //This is setting the global variable kitty to some value
    kitty = 2;
    printf("The value of \"kitty\" inside function1: %d\n", kitty);
    return;}

//Arbitrary function 2
void function2 (){
    //This is going to increment 2 to kitty and should be viewable to all functions
    kitty = kitty + 2;
    printf("The value of \"kitty\" inside function2: %d\n", kitty);
    return;}

//Arbitrary function 3
void function3 (){
    //Just dont modify kitty but read from it
    printf("The value of \"kitty\" inside function3: %d\n", kitty);
    return;}

int main(void){

//See whether Main can see the variable kitty and what main will see the value of kitty as
printf("The value of \"kitty\" inside main initially: %d\n", kitty);

//Call arbitrary function 1
function1();

//Print the value of kitty after  function 1 has finished executing
printf("The value of  \"kitty\" after arbitary function 1 has run inside main: %d\n", kitty);

//Call arbitrary function 2
function2();

//Print the value of kitty after function 2 has finished executing
printf("The value of  \"kitty\" after arbitary function 2 has run inside main: %d\n", kitty);

//Call arbitrary function 3
function3();

//Print the value of kitty after function 3 has finished executing
printf("The value of  \"kitty\" after arbitary function 3 has run inside main: %d\n", kitty);

return 0;}
```

The result of running the C program above is shown below,

```
The value of "kitty" inside main initially: 3
The value of "kitty" inside function1: 2
The value of  "kitty" after arbitary function 1 has run inside main: 2
The value of "kitty" inside function2: 4
The value of  "kitty" after arbitary function 2 has run inside main: 4
The value of "kitty" inside function3: 4
The value of  "kitty" after arbitary function 3 has run inside main: 4
```

0.3 Conditional Statements

When we write computer programs, sometimes we want the computer program to behave differently if we give different inputs. For example, if we are writing a C program to solve a quadratic equation, we might want our C program to behave differently if we have real roots compared to when we have complex roots. If we are writing a program for a small robot, perhaps we want it to make a decision and behave differently if it finds itself in different environments. That is why we need conditionals. Conditionals allow us to execute parts of our program selectively based on inputs that are determined while the program is running live.

0.3.1 Introduction

We need to introduce a new dataype called booleans. Booleans are datatypes that contain either true or false. Now C can interpret integers as booleans and vice versa.

If we have an integer that contains the number 0, then C can interpret this integer as a boolean that contains false. If we have an integer that contains a non-zero number, then C can interpret that integer as a boolean that contains true.

If we have a boolean that contains true, then C can interpret that boolean as an integer that contains 1. If we have a boolean that contains false, then C can interpret that boolean as an integer that contains 0.

Now we have to discuss the numerical boolean operators. These are operators like `==`, `<` , and `>`. These kind of work the way you would expect, so the `==` operator is used to compare between 2 numbers. If we have 2 numbers which are the same like `2==2`, then we would get true. If the numbers are different and we use the `==` operator, we would get false. For example, `1==2` would evaluate to false. The `<` operator us less than.

Now we have to discuss the logical boolean operators. We have shown 3 common logical operators which is the logical AND (`&&`), the logical OR (`||`) , and the logical NOT (`!`).

The C program below demonstrates some of the concepts discussed above.

```c
//This is the standard input output library.
//This allows us to print things to the terminal console
#include <stdio.h>

//We need to use a new library, standard boolean
//This library allows us to declare and use boolean datatypes
#include <stdbool.h>


int main(void){

//This is a declaration of a boolean datatype
bool koala;

//It can be set to true,
koala = true;

//Let us see the value of koala
printf("The boolean koala after setting it to true: %d\n", koala);

//The boolean can also be set to false
koala = false;

//Let us see the value of koala after setting it to false
printf("The boolean koala after setting it to false: %d\n", koala);

//From the above example we know that a true boolean is represented as 1
//We also know that a false boolean is represented as 0
//Just to separate things a bit better
printf("######################################################################\n");

//Introduction to equality comparison
printf("The number 1 is equals to 1: %d\n", 1==1);
printf("The number 1 is not equals to 2: %d\n", 1==2);

//Introduction to less than comparison
printf("The number 2 is less than 3: %d\n", 2<3);
printf("The number 3 is not less than 1: %d\n", 3<1);

//Introduction to greater than comparison
printf("The number 7 is greater than the number 4: %d\n", 7>4);
printf("The number 11 is not greater than the number 13: %d\n", 11>13);

//Just to separate things a bit better
printf("######################################################################\n");

//Introduction to AND comparison (arg1 && arg2)
printf("This is for AND Gates:\n");
printf("Only 2 trues can return a true: %d\n", true && true);
printf("1 true and 1 false is not enough to return a true: %d\n", true && false);
printf("1 false and 1 true is not enough to return a true: %d\n", false && true);
printf("Two falses in an AND gate will always give false: %d\n", false && false);

//Introduction to OR comparison (arg1 || arg2)
printf("This is for OR Gates:\n");
printf("2 trues will always return true: %d\n", true || true);
printf("1 true and 1 false will return a true: %d\n", true || false);
printf("1 false and 1 truw ill return a true: %d\n",  false || true);
printf("Only 2 falses will return a false: %d\n", false || false);

//Introduction to NOT operator (!arg1)
printf("This is for the NOT Operator:\n");
printf("This will invert the true into a false: %d\n", !true);
printf("This will invert the false into a true: %d\n", !false);

return 0;}
```

The result of running the `C` program above is shown below,

```
The boolean koala after setting it to true: 1
The boolean koala after setting it to false: 0
######################################################################
The number 1 is equals to 1: 1
The number 1 is not equals to 2: 0
The number 2 is less than 3: 1
The number 3 is not less than 1: 0
The number 7 is greater than the number 4: 1
The number 11 is not greater than the number 13: 0
######################################################################
This is for AND Gates:
Only 2 trues can return a true: 1
1 true and 1 false is not enough to return a true: 0
1 false and 1 true is not enough to return a true: 0
Two falses in an AND gate will always give false: 0
This is for OR Gates:
2 trues will always return true: 1
1 true and 1 false will return a true: 1
1 false and 1 truw ill return a true: 1
Only 2 falses will return a false: 0
This is for the NOT Operator:
This will invert the true into a false: 0
This will invert the false into a true: 1
```

0.3.2 Basic If Statements

```c
//This is the standard input output library which gives us functions like printf for printing
//This library allows us to print the values of variables
#include <stdio.h>

//This is the standard boolean library.
//This library allows us to declare boolean data types, which is a new datatype
//you will encounter here
#include <stdbool.h>

int main(void){

//This is a new datatype called a boolean. It is smaller than an integer in size.
//Its value can be true or false. True can be represented as 1
//False can be represented as 0
  bool bootcamp = true;
//bool bootcamp = 1;
//The two ways of declaring and initializing bootcamp above are perfectly valid

//So this is the head of the if statement
//If whatever is in the () brackets end up being true in a boolean sense
//Execute the contents of the if statement body
if (bootcamp)
    //This is the body of the if statement
    //Since bootcamp is true, this will execute
    {printf("Bootcamp is indeed True!\n");}
//This is the head of the else statement.
//If we have an if statement, we don't necessarily need to declare a corresponding else statement
//But if we have an else statement, we absolutely must have an if statement before that else
    statement
//So this part will execute only if the if statement condition is false
//Since bool is true, the if statement condition is true and so the body of this else will not
    execute
else
    //This is the body of the else statement
    {printf("Bootcamp is false!\n");}

return 0;}
```

The result of running the C program above is shown below,

```
Bootcamp is indeed True!
```

Alternatively, we can set the boolean to false,

```c
//This is the standard input output library which gives us functions like printf for printing
//This library allows us to print the values of variables
#include <stdio.h>

//This is the standard boolean library.
//This library allows us to declare boolean data types, which is a new datatype
//you will encounter here
#include <stdbool.h>

int main(void){

//This is a new datatype called a boolean. It is smaller than an integer in size.
//Its value can be true or false. True can be represented as 1
//False can be represented as 0
  bool bootcamp = false;
//bool bootcamp = 0;
//The two ways of declaring and initializing bootcamp above are perfectly valid

//So this is the head of the if statement
//If whatever is in the () brackets end up being true in a boolean sense
//Execute the contents of the if statement body
if (bootcamp)
    //This is the body of the if statement
    {printf("Bootcamp is indeed True!\n");}
//This is the head of the else statement.
//If we have an if statement, we don't necessarily need to declare a corresponding else statement
//But if we have an else statement, we absolutely must have an if statement before that else
    statement
//So this part will execute only if the if statement condition is false
else
    //This is the body of the else statement
    {printf("Bootcamp is false!\n");}

return 0;}
```

The result of running the `C` program above is shown below,

```
Bootcamp is false!
```

0.3.3 Nested Else If Statements

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

int main(void){

//This is a declaration of an integer named option
int option = 6;

//We are introducing a new operator here which is the == operator. This is an equality operator
//So it compares whether the 2 variables are the same with each other and returns a boolean
//Here is what I mean: 1==1 will return true because indeed the integer 1 is the same as the integer
     1
//1==2 will return false because the integer 1 is not the same as the integer 2
//3==(1+2) will return true because indeed the integer 3 is equals to the integer 3 which is 1+2

//This is an if statement
if (option == 0)
    {printf("The value of option is 0\n");}
//This is an else if statement and its body will execute only if its () condition is true
else if (option == 1)
    {printf("The value of option is 1\n");}
//The body of this else if statement will only execute if (option==2) is true.
//So if option is anything else othern than 2, the body of this else if will never execute
else if (option == 2)
    //This is the body of the else if statement
    {printf("The value of option is 2\n");}
//If a single else if statement has been triggered, for example if this one is triggered
//The body of this statement will get executed,
//but C would also ignore the remaining else if because it already found one that was true
//So for example if this one got triggered, then C would just skip to the end after the else
    statement
//and ignore all the else if 4-6.
else if (option == 3)
    {printf("The value of option is 3\n");}
//Likewise if this gets executed, C would go ahead do the body of this else if
//and then jump after the else statement
else if (option == 4)
    {printf("The value of option is 4\n");}
//The body of this else if statement will only execute if option's value is 5
else if (option == 5)
    {printf("The value of option is 5\n");}
//The body of this else if statement will only execute if the option's value is 6
else if (option == 6)
    {printf("The value of option is 6\n");}
//This is an else statement
else
    {printf("The value of option is not 1-6\n");}

return 0;}
```

The result of running the C program above is shown below,

```
The value of option is 6
```

Why don't you try changing the value of the variable `option`. You will find that as you change the variable `option`, different parts of the code will execute and we will get different print messages.

0.4 Looping

The way looping works typically is that the C program will check if a certain condition is true, and if it is true, then it will execute the body of a loop. After it reaches the end of the loop's body, it will recheck if that condition is true or not, again. If that condition is still true, then C would execute the body of the loop again. After it has reached the end of the loop's body it will recheck that condition. This process will keep executing until eventually the condition becomes false and that is where C would no longer execute the body of the loop.

0.4.1 While Loops

```
#include <stdio.h>

int main(void){

//This is to indicate the start of the main function
printf("This is the start of the main function...\n");

//Here we are declaring-initializing an integer named var1 to 0
int var1 = 0;

//This is the head of a while loop statement
//As long as (var1<10) is true, which means for var1 is 1-9, the body of this while loop
//will keep on executing
while (var1 < 10)
    //This is the body of the while loop statement
    //Note the same as the previous if and else and else if statement cases,
    //the body is encased in curly brackets
    //The body of the loop prints the value of var1 and then adds 1 to var1
    {printf("The value of var1: %d\n", var1);
    var1 = var1 + 1;}

//This is to indicate the end of the main function
printf("This is the end of the main function...\n");

return 0;}
```

The result of running the C program above is shown below,

```
This is the start of the main function...
The value of var1: 0
The value of var1: 1
The value of var1: 2
The value of var1: 3
The value of var1: 4
The value of var1: 5
The value of var1: 6
The value of var1: 7
The value of var1: 8
The value of var1: 9
This is the end of the main function...
```

0.4.2 For Loops

It is kind of tedious to write loops using the `while` keyword. So the `for` loop is implemented to make it more convenient to declare common `while` loop arrangements.

```c
#include <stdio.h>

int main(void){

//This does the same thing as the while loop example.
//A for loop allows you to be more concise when specifying a few types of while loops
//This is the structure of a for loop. A for loop has 3 things in its arguments ()
//arg1: the "int var1 = 0" is the declaration and initialization of a variable
//arg2: the "var1<0" is the condition that will be checked. If this condition false, stop loop
//arg3: "var1 = var+1" this is a loop update statement. It basically just incremenets var
//      by 1 each iteration of the for loop
//In short, this while loop performs the same thing as the previous example
for (int var1 = 0; var1<10; var1 = var1+1)
    { printf("Value of var1: %d\n", var1);}

return 0;}
```

The result of running the `C` program above is shown below,

```
Value of var1: 0
Value of var1: 1
Value of var1: 2
Value of var1: 3
Value of var1: 4
Value of var1: 5
Value of var1: 6
Value of var1: 7
Value of var1: 8
Value of var1: 9
```

0.5 Pointers and Addresses

When we declare and initialize a single variable in a computer program, there are 2 values that we can asociate to the variable. The first is the value inside the variable itself. The second is the location of where the variable is stored in the computer memory, this can be represented as some garbage integer. It is important to learn about pointers when we start dealing with arrays because of how similar arrays are to pointers. When we start using arrays and want to make functions that operate on arrays, we definitely must use pointers.

Now a pointer is a variable that holds an address. This pointer that holds an address can be given the address of a separate variable. A pointer can be used to change values in memory. This all sounds abstract, but hopefully would be a little clearer by the end of these examples.

0.5.1 Declaring, Setting, & Using Pointers

Now the small `C` program below demonstrates the basics of how to use a pointer. We need to introduce 2 new operators, which is the `*` operator and the `&` operator. Earlier, we mentioned that a pointer holds the value of an address. The way we "extract" the value of an address from a variable is to use the & operator. If we call the & operator on a variable named `var` like `&var`, then we get the location of where `var` is stored in computer memory. Now supposing we already have a pointer that contains the address of another variable, we can use the * operator to tell C to use the value found at the memory address contained inside the pointer. So if we have a pointer named `ptr` and it contains the address of variable `var`, then `*ptr` would mean the value of `var`. This is all demonstrated in the example below,

```c
#include <stdio.h>

int main(void){
    //This is a declaration of an integer
    int soap;
    //initialization of integer "soap" so that it has value 1
    soap = 1;
    //This is a declaration of an integer pointer. Right now it points nowhere
    int* pointer_to_soap;
    //We are now setting the integer pointer to hold the value of the address of "soap"
    //The & operator can be used to get the address of a certain variable
    pointer_to_soap = &soap;

    //Let us print the value of "soap"
    printf("Soap's value: %d\n", soap);
    //Let us print the memory address of soap
    printf("Soap's Memory address: %d\n", &soap);
    //Let us print the value of "pointer to soap"
    printf("The value inside pointer_to_soap: %d\n", pointer_to_soap);
    //Let us print the value of what "pointer_to_soap" is referencing
    printf("What value pointer_to_soap is pointing at: %d\n", *pointer_to_soap);

    //So pointer_to_soap contains the address of where the variable "soap" is stored in memory
    //Printing pointer_to_soap literally gives the address of the variable "soap"
    //The * operator can be used to fetch the data that the pointer is referencing
    //So when we print *pointer_to_soap, we retrieved the value of whatever pointer_to_soap is
    //pointing at, which in this case is Soap's value, so printing *pointer_to_soap prints 2

return 0;}
```

The result of running the C program above is shown below,

```
Soap's value: 1
Soap's Memory address: -81073732
The value inside pointer_to_soap: -81073732
What value pointer_to_soap is pointing at: 1
```

Let us analyze the output of running the program. We declared and initialized `soap` to be an integer that contains 1 so when we print the value of `soap`, the program will print 1. We then printed the memory address of the variable `soap`, which is a long list of integer. Now this memory address is just the location of where the variable `soap` is stored in computer memory. We then printed just the value contained inside the `pointer_to_soap` wherein we have set that pointer to contain the address of the variable `soap`. So we can verify that indeed `pointer_to_soap` contains the memory of the variable `soap`. After that, we are demonstrating that when we use * operator on `pointer_to_soap`, we succesfully got the value contained inside the computer memory that `pointer_to_soap` has. Since `pointer_to_soap` contains the memory address of the variable `soap` and `soap` contains value of 1, we succesfully printed 1 correctly. I hope it is clearer what the basic principles of pointers are.

0.5.2 Passing Pointers to Functions

Now here we are going to introduce the syntax of using pointers as input parameters to a function. This is useful when we are writing functions that modify arrays, more detail would be explained later. So we made the function `add1` accept an integer pointer. Within `Main`, we then passed the address of soap into function add1. The variable integer_pointer is a local variable within the function add1, and its value was initialized to the address of the variable soap. Since Main has not finished executing when the function add1 was called, the variable soap still exists in computer memory, so we can still use the memory location of the variable soap without worrying that it contains garbage values. Remember that integer_pointer now contains the address of soap, so when we use the * operator to do *integer_pointer = *integer_pointer+1; we are actually changing the value of soap. This means that when we pass pointers as parameters to some function named function1, we can change variables that are declared outside of function1, as we have done in this example. In our case, we are using the function add1 to modify a variable named soap that is declared not in add1, but instead in main.

```
#include <stdio.h>

//This function accepts an integer pointer and does not return anything
//Since an integer pointer contains a memory address of an integer variable,
//This function will accept memory addresses.
void add1(int* integer_pointer){
    //Whatver integer_pointer is pointing at is incremented by 1
    *integer_pointer = *integer_pointer+1;
    //Go back to this function's caller, which is main
    return;}

int main(void){
    //This is a declaration of an integer. initialization of integer "soap" so that it has value 1
    int soap = 1;

    //This should be printing 1
    printf("Value of Soap before \"add1\" is called: %d\n", soap);

    //We are calling the function add 1 here
    //We are passing the input to add1 as the address of the integer soap
    add1(&soap);

    //This should be printing 2
    printf("Value of Soap after \"add1\" is called: %d\n", soap);

return 0;}
```

The result of running the C program above is shown below,

```
Value of Soap before "add1" is called: 1
Value of Soap after "add1" is called: 2
```

Let us analyze the output of this program. We declared and then initialized the variable soap to initially contain 1 inside main, so when we print the value of soap we shouldn't be surprised that we got 1. We then called add1 passing the address of soap. The local variable within add1 named integer_pointer is now initialized with the address of variable soap and changed soap's value to 2. The function add1 finished executing and we go back to main. Within main, the variable soap now no longer contains 1. It contains 2 because add1 has changed the soap's value in main. Therefore, when we print the variable soap within main, we get the value of 2 and no longer 1. This demonstrates that a function can modify variables that are declared elsewhere as long as pointers were used.

0.5.3 Array-Pointers Similarities

```
#include <stdio.h>

int main(void){

//Declaration and initialization of an array
int array[3] = {1,2,3};

```

```c
//Arrays can be treated just like integer pointers. They are actually the same thing
//the variable "array" holds memory addresses, which is basically what a pointer does
printf("The value of the literal variable \"array\": %d\n", array);

//Just like integer pointers, the * operator can be used to show what the array is pointing at
printf("What the array is pointing at: %d\n", *array);

//We can accesss the elements of an array by using the * operator to access adjacent memory
printf("Second element of the array: %d\n", *(array+1));

return 0;}
```

The result of running the C program above is shown below,

```
The value of the literal variable "array": 1180113788
What the array is pointing at: 1
Second element of the array: 2
```

0.5.4 Acessing adjacent Memories for Pointers

```c
#include <stdio.h>

void PrintArr (int* arr){
    printf("What integer pointer is pointing at: %d\n", *arr);
    printf("What is right next to what integer pointer is pointing at: %d\n", *(arr+1));
    printf("What is right next 2 time to what integer pointer is pointing at: %d\n", *(arr+2));
    return;}

int main(void){

//Declaration and initialization of an array
int array[3] = {1,2,3};

//Since array is actually just an address, you can just pass array directly as if
//it was an integer pointer. It makes no difference
PrintArr(array);

return 0;}
```

The result of running the C program above is shown below,

```
What integer pointer is pointing at: 1
What is right next to what integer pointer is pointing at: 2
What is right next 2 time to what integer pointer is pointing at: 3
```