Structure of a C Program

Consider the following code:

```
#include <stdio.h>

int main() {
    int a, b, sum;
    scanf("%d %d", &a, &b);
    sum = a + b;
    printf("Sum of %d and %d is %d\n", a, b, sum);
    return 0;
}
```

For now, let's ignore what this program is trying to do. Right now, we are concerned about the structure.

Any `C` program follows a structure like this, and requires some parts of this code to run any sort of program. Let's dive deep and see what's going on in each line.

Preprocessor Directive

This tells the compiler to include contents from other files as mentioned. For example, over here, `#include <stdio.h>` tells the compiler to include all the contents from the header file `stdio`.

It can be really useful in modular codes, where we want to use our own custom functions that we have written on another file. Also, this is the standard way to get contents of an already existing header file, that comes installed with the language.

There are many header files that comes along with the language. The `stdio` (meaning standard input/output) is used to take input and give output.

The `main()` Function

This is the most important part of any `C` program as the program won't compile without this function.

Any compiler expects this `main()` function as a compiler only creates an executable for this main function, after reading the entire file or modules.

Hence, any code that must be part of the program must go inside this function. Also, it is a built-in function, meaning it cannot be defined by the user for any other purpose.

Block

All statements that come under the main function inside the `{ ... }` is part of the block. It is also called compound statements, and let's see what are there in this block.

Identifiers

We see three identifiers `a` , `b` and `sum` by the user and they are of type `int` , meaning, the data stored in these variables are integers.

Identifirs are names given to memory locations by the user for further usage and easy recognision. They last only until the program is completed, and have scopes in the program itself:

- **Local**: If a variable is declared inside a block, then it can only be accessed inside that block. Not anywhere else in the program, or other blocks.
- **Global**: If the variable is declared outside blocks .i.e. No block or the main program, then it can be accessed anywhere in the program, including blocks.

Data Types

We see that the program creates variables of type `int` meaning integer here. However, there are other data types which can be used as well in C programs. They are:

- Numerical
    - `int` : As mentioned, stores integer values only. If a decimal value is given, it extracts the integer part from it and stores it.
    - `float` : Stores numeric data with single precision. That is, 4 bytes or 6 - 7 decimal places.

    - `double` : Also called long float, stores numeric data with double precision. That is, 8 bytes or 15 - 16 decimal places.
    - `long double` : Stores an even longer float, whose length is compiler dependent. It varies from 10 - 16 bytes depending on the compiler.
- Text
    - `char` : Stores a single ASCII charecter. So it occupies 1 byte only.
    - `string` : Stores a sequence of charectars, or text. It requires the `string.h` file to be used.
- Special
    - `void` : A type which returns nothing. It holds no value. It is usually used for functions which don't return anything after process.
- User defined
    - `struct`
    - `union`
    - `enum`
    - `typedef`

Reading Input

To read input from the user, we need to keep three important things in mind. All of these are require to read the input from user:

- `scanf()` : A function that is used to read input. It is part of the `stdio.h` file.
- `%d` : Format specifier that tells the compiler what type of input to expect. There are various format specifiers.
- `&` : Reference or address-of tells the compiler to store the read value at the memory location of the variable specified after it.

Format Specifiers

Used while reading input or writing output. It acts like placeholders in strings while writing the output, and specifies the type of data to expect in that place. There are three major format specifiers we will be using:

- `%d` - `int`

- `%f` - `float`. It can be modified based on number of decimal places to represent. To round off to n decimal places, it becomes `%.nf`.
- `%lf` - `double`. It can be modified based on number of decimal places to represent. To round off to n decimal places, it becomes `%.nlf`.
- `%c` - `char`
- `%s` - `string`

Operations

The next line is just an operation. Just like in mathematics, we have an `=` to assign something and a `+` operator that does an operation based on two inputs. In this case, it is addition.

This concept will be covered in detail in the upcoming PDFs.

Writing Output

The next line talks about writing the output on the display. There are some things to keep in mind while writing the output:

- `printf()` is the function used to write something on the terminal or output window.
- `%d` is a format specifier that tells that the value that must get here must be of type `int`.
- The output must be in strings.
- The number of arguments passed after the string should be equal to the number of format specifiers in that string. Basically, the respective values passed gets substituted in the specifiers.

In this case, we want to write `sum = [some integer]` on the display.

Returning a Value

At the last line of the program, we can see a `return 0` statement. This does nothing other than satisfying the function's return type (mentioned before function name in it's header) which is `int` in this case.

Also, it tells the OS that the program has successfully executed and the compiler can finish compiling to create an executable. It does not matter what integer is being returned.

Common Mistakes

There are few mistakes that a programmer at any level can commit while using this language. Hence, it is important to be always watchful for these, although an error will interrupt the execution or something wierd happens. Here are these common errors:

- Not terminating a statement with `;` → Every statement ends with `;`
- Not closing blocks with `{}` → Make sure to open a block with `{` and close it correctly with `}`
- Not reading input using `&` → Make sure teh variables in which the input is stored is addressed using `&`
- Wrong specifiers used → Although the code works fine, it gives wierd results. Make sure to use a specifier of the type required.
- Passing in less or more arguments for repacing specifiers → Make sure to pass `n` arguments if there are `n` specifiers in the string.
- Forgetting `return 0` or returning values of wrong type → Make sure to either change the return type of function (preferably `void`) or check the value being returned.