1. Typical structure of a C program

The structure of C programming language is as follows:

1. Preprocessor directives
2. Global declarations
3. Main function
4. Function definition

1.1 Preprocessor directives

In C, preprocessor directives are instructions that are executed before the start of the code compilation process. The preprocessor carries out these instructions before the compilation process. They help with conditional compilation, code inclusion, macro definitions, and more. They begin with the # sign. In the C language, preprocessor directives come in many forms:

1. File inclusion
2. Macro definition
3. Conditional compilation

1.1.1. File inclusion

The **#include** directive is used to include the contents of a file in the program. This is used to include header files, which contain function declarations, macros, and constants. You can include standard C libraries using angle brackets **(< >)**. You can also include user-defined header files by using double quotes **(" ")**.

Eg –

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

This includes the standard I/O library which provides functions like **printf** and **scanf**

```
#include "myheader.h"
```

This includes the content of **myheader.h**, which contain custom declarations written by the user.

1.1.2. Macro definitions

The **#define** directive is used to create macros, which are constants or expressions that are substituted by the preprocessor wherever they appear in the code. Macros help to make code more readable and maintainable.

Eg –

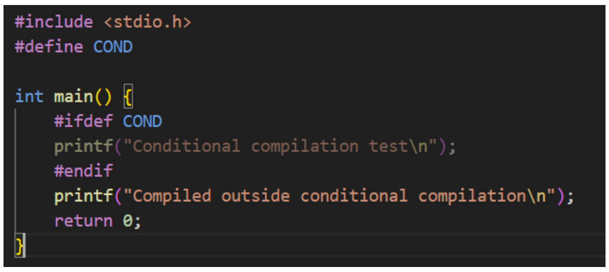
```
#define PI 3.14
```

In this case, **PI** is replaced with 3.14 throughout the program.

1.1.3. Conditional compilation

Conditional compilation allows you to include or exclude parts of the code based on conditions.

Eg –

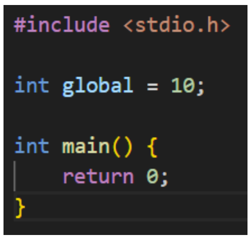


```c
#include <stdio.h>
#define COND

int main() {
    #ifdef COND
    printf("Conditional compilation test\n");
    #endif
    printf("Compiled outside conditional compilation\n");
    return 0;
}
```

Here #ifdef is used for conditional compilation. Since the macro COND is defined at the beginning of the program, the code enclosed within the #ifdef statement will run. But if the macro COND wasn't defined, the conditional print statement will not run. The print statement outside the conditional compilation will run no matter what.

1.2. Global declarations

Global variables and function are accessible throughout the entire program and exist for the duration of the program's execution.



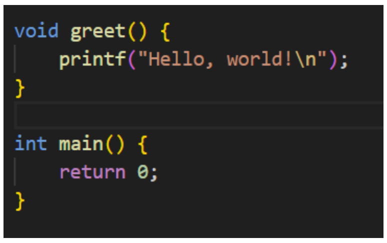
```c
#include <stdio.h>

int global = 10;

int main() {
    return 0;
}
```

1.3. Main function

Program Entry Point is the main function in C (int main()), which is the starting point for execution.

1.4. Function definitions

Functions facilitate improved organization and code reuse. A function is defined in C with a name, a list of parameters, and a return type. Expressions, return statements, and control flow statements (if, for, while) make up the body of the function. Specific grammar rules must be followed by every statement.



```c
void greet() {
    printf("Hello, world!\n");
}

int main() {
    return 0;
}
```

BNF of a typical simplified C program structure

<program> ::= <preprocessor_directive> <main_function>

<preprocessor_directive> ::= "#" "include" "<" <header_file> ">"

<header_file> ::= "stdio.h" | "stdlib.h"

<main_function> ::= "int" "main" "(" ")" "{" <statements> "return" <integer_literal> ";" "}"

<statements> ::= { <statement> }

<statement> ::= <expression_statement> | <declaration_statement>

<expression_statement> ::= <identifier> "=" <literal> ";"

<declaration_statement> ::= <type> <identifier> ";"

<type> ::= "int" | "float"

<literal> ::= <integer_literal>

<integer_literal> ::= digit { digit }

<identifier> ::= letter { letter | digit | "_" }

2. Comments in C programs

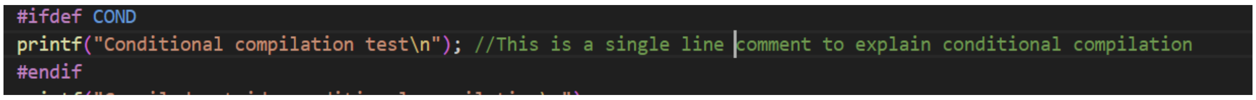
Comments are considered non-executable lexical elements. They are part of the source code but do not contribute to the program's behavior or logic. The compiler ignores them during execution. Comments make the code easier to understand and help during debugging or testing stages. C has two main types of comments:

1. Single lined comments
2. Multi lined comments

2.1. Single lined comments

Single lined comments in C are written with two forward slashes **(//)**. Anything that follows // on that line is treated as a comment and is not executed by the compiler.

Eg –

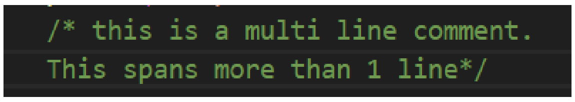


```c
#ifdef COND
printf("Conditional compilation test\n"); //This is a single line comment to explain conditional compilation
#endif
```

2.2. Multi lined comments

Multi-line comments are written between /* and */. Everything within this block is treated as a comment. This type of comment allows the user to include explanations that span multiple lines, making it suitable for detailed descriptions or temporarily disabling sections of code during debugging or testing.

Eg –



```c
  /* this is a multi line comment.
  This spans more than 1 line*/
```

3. Data types in C language

Data types are important in defining the kind of data that can be stored in variables. They provide semantic meaning to the data and are crucial for memory allocation, data interpretation, and operations performed on that data. C language provides several fundamental data types. Below are examples of five commonly used data types :

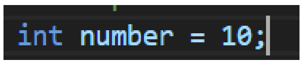
1. int
2. float

3. double
4. char
5. long

3.1. int data type

The int data type is used to store whole numbers (integers) without any decimal points.

Eg –



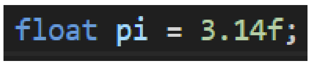
```
int number = 10;
```

Here the example literal is 10.

3.2. float data type

The float data type is used to store single-precision floating-point numbers, which are numbers with decimal points.

Eg –



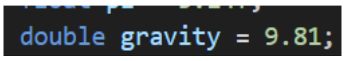
```
float pi = 3.14f;
```

Here the example literal is 3.14f.

3.3. double data type

The double data type is used to store double-precision floating-point numbers. It offers greater precision and a larger range than float, making it suitable for more complex mathematical computations where higher accuracy is required.

Eg –



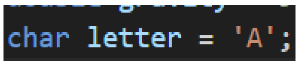
```
double gravity = 9.81;
```

Here the example literal is 9.81.

3.4. char data type

The char data type is used to store a single character or an ASCII value. It can represent letters, digits, punctuation, or other symbols.

Eg –



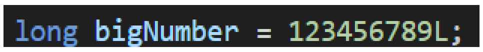
```
char letter = 'A';
```

Here the example literal is ‘A’.

3.5 long data type

The long data type is used to store larger integers than those stored by the int data type. It is useful when the range of values needed exceeds the limits of a standard int.

Eg –



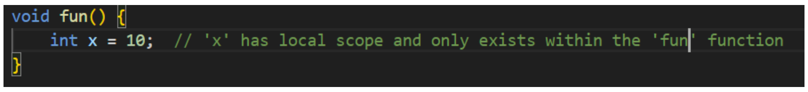
```
long bigNumber = 123456789L;
```

Here the example literal is 123456789L.

4. **Scope of variables is C language**

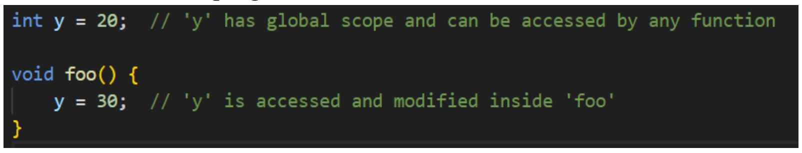
In C, there are three types of variable scopes:

1. Local scope - Variables declared within a function or block are local to that function or block and can't be accessed outside.



```c
void fun() {
    int x = 10;  // 'x' has local scope and only exists within the 'fun' function
}
```

2. Global scope - Variables declared outside any function are global and can be accessed from any function within the program.



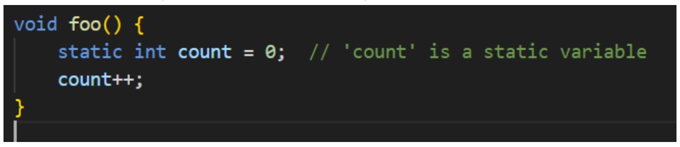
```c
int y = 20;  // 'y' has global scope and can be accessed by any function

void foo() {
    y = 30;  // 'y' is accessed and modified inside 'foo'
}
```

5. **Lifetime of variables**

In C, the lifetime of a variable refers to how long the variable exists in memory during the execution of a program. Variables in C can have different lifetimes depending on where and how they are declared.

1. Local variables – declared inside a function of a block and it exists from the point of declaration until the end of the function or block where it was defined.

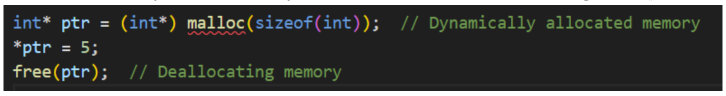
2. Static variables – it is declared inside a function or block with the static keyword or outside all functions (global scope). It exists for the entire duration of the program but retains its value across function calls (if inside a function).



```c
void foo() {
    static int count = 0;  // 'count' is a static variable
    count++;
}
```

3. Global variables – it is declared outside of all functions, at the top of a file. It exists for the entire duration of the program.

4. Dynamic variables – it is created using memory allocation functions (malloc, calloc, realloc) at runtime. It stays in memory until we deallocate it using free().



```c
int* ptr = (int*) malloc(sizeof(int));  // Dynamically allocated memory
*ptr = 5;
free(ptr);  // Deallocating memory
```