C

```
// helloworld.c

#include <stdio.h>

int main() {
     printf("Hello World!\nI'm learning C.\n");
return 0;
}
```

```
user@haxixe:~/$ gcc helloworld.c -o run
user@haxixe:~/$ ./run
Hello World!
I'm learning C.
user@haxixe:~/$ ▮
```

a. *return 0* ends the main() function and it signals to the OS that the program exited successfully;
b. Every C statement ends with a semicolon;
c. The compiler ignores white spaces. However, multiple lines makes the code more readable.

ESCAPE SEQUENCE	DESCRIPTION
\n	Creates a new line.
\t	Creates a horizontal tab.
\\	Inserts a backslash character (\).
\"	Inserts a double quote character (“).
\'	Inserts a single quote character(‘).
\0	Insert a null character.

ARITHMETIC OPERATORS	DESCRIPTION
+	Adds together two values.
-	Subtracts one value from another.
*	Multiplies two values.
/	Divides one value by another.
%	Returns the division remainder.
++	Increases the value of a variable by 1.
--	Decreases the value of a variable by 1.

COMPARISON OPERATORS	DESCRIPTION
==	X equal to Y.
!=	X not equal to Y.
>	X greater than Y.
<	X less than Y.
>=	X greater than or equal to Y.
<=	X less than or equal to Y.

DATA TYPE	FORMAT SPECIFIER	DESCRIPTION
int	%d %i	Stores whole numbers, without decimals.
float	%f	Stores fractional numbers, containing one or more decimals. Sufficient for storing 7 decimal digits.
double	%lf	Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits.
char	%c	Stores a single character/letter/ number, or ASCII values.
char varName[]	%s	Strings are used for storing text/characters.
varType varName = value;		

a. You can also declare a variable without assigning the value and assign it later;
b. If you assign a new value to an existing variable, it will overwrite the previous value;
c. When you don't want others (or yourself) to override existing variable values, use the *const* keyword (this will declare the variable as "constant", which means unchangeable and read-only). Another thing about constant variables, is that it is considered good practice to declare them with uppercase. It is not required, but useful for code readability and common for C programmers;

IF… ELSE

```
if (condition) {
   // block of code to be executed if
   the condition is true
} else {
   // block of code to be executed if
   the condition is false
  }
```

SWITCH

```
switch(expression) {
   case x:
      // code block
   break;
   case y:
      // code block
   break;
   default:
      // code block
}
```

FOR

```
for (statement 1; statement 2; statement
3){
   // code block to be executed
}
```

ARRAY
a. Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value;

```
int myNumbers[] = {25, 50, 75, 100};
```

b. To access an array element, refer to its index number;
c. Array indexes start with 0: [0] is the first element. [1] is the second element, etc;

```
int myNumbers[] = {25, 50, 75, 100};
printf("%d", myNumbers[0]);
```

d. o change the value of a specific element, refer to the index number.

```
int myNumbers[] = {25, 50, 75, 100};
myNumbers[0] = 33;
printf("%d", myNumbers[0]);
```

e. You can loop through the array elements with the *for* loop.

```
int myNumbers[] = {25, 50, 75, 100};
int i;

for (i = 0; i < 4; i++) {
   printf("%d\n", myNumbers[i]);
}
```

f. Another common way to create arrays, is to specify the size of the array, and add elements later.

```
// Declare an array of three integers:
int myNumbers[3];
// Add elements
myNumbers[0] = 25;
myNumbers[1] = 50;
myNumbers[2] = 75;
```

LOGICAL OPERATORS	DESCRIPTION
&&	Returns true if both statements are true.
||	Returns true if one of the statements is true.
!	Reverse the result, returns false if the result is true.

ASSIGNMENT OPERATORS	DESCRIPTION
x = y	x = y
x += y	x = x + y
x -= y	x = x - y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y
x &= y	x = x & y
x |= y	x = x | y
x ^= y	x = x ^ y
x >>= y	x = x >> y
x <<= y	x = x << y

ELSE IF

```
if (firstCondition) {
   // block of code to be executed if
   condition1 is true
} else if (secondCondition) {
   // block of code to be executed if the
   condition 1 is false and condition 2
   is true
} else {
   // block of code to be executed if the
   condition 1 is false and condition 2
   is false
}
```

WHILE

```
while (condition) {
   // code block to be executed
}
```

DO… WHILE

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

a. The *break* statement can also be used to jump out of a loop.
b. The *continue* statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.
c. Do not forget to increase the variable used in the condition, otherwise the loop will never end.

STRINGS / CHAR ARRAY

```
char greetings[] = "Hello World!";
printf("%s", greetings);
```

```
char greetings[] = {'H', 'e', 'l', 'l',
'o', ' ', 'W', 'o', 'r', 'l', 'd', '!',
'\0'};
printf("%s", greetings);
```

MEMORY ADDRESS and POINTERS
a. When a variable is created in C, a memory address is assigned to the variable.
b. The memory address is the location of where the variable is stored on the computer.
c. To access it, use the reference operator (&), and the result will represent where the variable is stored.

```
int myAge = 67; // An int variable
int* ptr = &myAge; // A pointer variable, with the name ptr, that stores the address of myAge

// Output the value of myAge (67)
printf("%d\n", myAge);

// Output the memory address of myAge(e.g. 0x7ffe5367e044)
printf("%p\n", &myAge);

// Output the memory address of myAge with the pointer (e.g. 0x7ffe5367e044)
printf("%p\n", ptr);
```

d. The memory address is in hexadecimal form (0x..). You definitely won't get the same result in your program.
e. You should also note that &myAge gives the memory address of the variable. A pointer is a variable that stores a memory address as its value. To print a pointer (an address), use the %p format specifier.
f. Pointers are a core feature of C because they allow direct memory access and manipulation. This enables more efficient code by reducing unnecessary data copying, and can improve performance when used properly. This low-level control is one of the key characteristics that sets C apart from many other languages.

FUNCTIONS
a. Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

```
returnType functionName(parameter1,
parameter2, parameter3) {
    // code to be executed
}
```

b. For code optimization, it is recommended to separate the declaration and the definition of the function.
c. You will often see C programs that have function declaration above main(), and function definition below main(). This will make the code better organized and easier to read.

```
// Function declaration
void myFunction();

// The main method
int main() {
    myFunction(); // call the function
return 0;
}

// Function definition
void myFunction() {
    printf("I just got executed!");
}
```

STRUCTURES
a. Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a member of the structure.

```
// Create a structure called myStructure
struct myStructure {
    int myNum;
    char myLetter;
};

int main() {

    // Create a structure variable of
    myStructure called s1
    struct myStructure s1;

    // Assign values to members of s1
    s1.myNum = 13;
    s1.myLetter = 'B';

    // Print values
    printf("My number: %d\n", s1.myNum);
    printf("My letter: %c\n",
    s1.myLetter);

return 0;
}
```

PREPROCESSORS
a. In C, the preprocessor runs before the actual compilation begins. It handles things like including files and defining macros;
b. Preprocessor commands begin with a # symbol and are called directives.

#INCLUDE
a. You have already seen the #include directive many times. It tells the compiler to include a file.
b. It is used to add libraries or custom header files.

```
/* These two directives will add standard library stdio.h and locally available myfile.h, both headers */

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

#IFDEF and #IFNDEF
a. These directives let you include or skip parts of the code depending on whether a macro is defined.
b. This is called conditional compilation, and it's useful for debugging or creating different versions of a program.

```
#define DEBUG

int main() {
    #ifdef DEBUG
       printf("Debug mode is ON\n");
    #endif
return 0;
}
```

c. If DEBUG is defined, the message will be printed. If it's not defined, that part of the code is skipped.

#DEFINE AND MACROS
a. A macro is a name that represents a value, or a piece of code, defined using the #define directive.

```
/* In the example below, PI is replaced with 3.14 before the program is compiled. This means that every time PI appears in the code, it will be replaced with 3.14 */

#define PI 3.14

int main() {
   printf("Value of PI: %.2f\n", PI);
return 0;
}
```

COMMAND-LINE ARGUMENTS
a. Command-line arguments allow you to pass values to a program when it is executed from the terminal;
b. They are received by the main function through its parameters;
c. This is useful for configuring program behavior without changing the source code.

PARAMETER	DESCRIPTION
argc	Number of command-line arguments passed to the program.
argv	An array of strings containing the arguments.
int main(int argc, char *argv[])	

a. argc is always at least 1.
b. argv[0] contains the program name.
c. Additional arguments start at argv[1].
d. Command-line arguments are strings. To use them as numbers, they must be converted.

```c
//cliarguments.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Argument count: %d\n", argc);

    for (int i = 0; i < argc; i++) {
        printf("argv[%d]: %s\n", i,
argv[i]);
    }

    return 0;
}
```

```
user@haxixe:~/$ ./run haxixe 99
Argument count: 3
argv[0]: ./run
argv[1]: haxixe
argv[2]: 99
user@haxixe:~/$ ▮
```

<stdio.h>

FILE HANDLING
a. In C, you can create, open, read, and write to files by declaring a pointer of type FILE, and use the fopen() function.

```
FILE *fptr;
fptr = fopen(filename, mode);
```

PARAMETER	DESCRIPTION	
filename	The file name or the path to it.	
mode	w	Writes to a file.
	a	Appends new data to a file.
	r	Reads from a file.

WRITE

```
FILE *fptr;

// Open a file in writing mode
fptr = fopen("filename.txt", "w");

// Write some text to the file
fprintf(fptr, "Some text");

// Close the file
fclose(fptr);
```

APPEND

```
FILE *fptr;

// Open a file in append mode
fptr = fopen("filename.txt", "a");

// Append some text to the file
fprintf(fptr, "\nHi everybody!");

// Close the file
fclose(fptr);
```

READ

```
FILE *fptr;

// Open a file in read mode
fptr = fopen("filename.txt", "r");

// Store the content of the file
char myString[100];

// Read the content and store it
fgets(myString, 100, fptr);

// Print the file content
printf("%s", myString);

// Close the file
fclose(fptr);
```