Difference between low level and high level languages

High level language	Low level language
It is considered as a programmer friendly language.	Considered as machine friendly language
Very easy to execute	Very difficult to execute
High-level languages require the use of a compiler or an interpreter for their translation into the machine code.	Low-level language requires an assembler for directly translating the instructions of the machine language.
Very low memory efficiency-consume more memory than any low-level language.	very high memory efficiency-consume less energy as compared to any high-level language.
These are portable from any one device to another.	A user cannot port these from one device to another.
High-level languages do not depend on machines.	Low-level languages are machine-dependent and thus very difficult to understand by a normal user.
very easy to debug these languages.	A programmer cannot easily debug these languages.
High-level languages have a simple and comprehensive maintenance technique.	It is quite complex to maintain any low-level language.
High-level languages take more time for execution as compared to low-level languages because these require a translation program.	The translation speed of low-level languages is very high.
One does not require a knowledge of hardware for writing programs.	Having knowledge of hardware is a prerequisite to writing programs.
The process of modifying programs is very difficult with high-level programs. It is because every single statement in it may execute a bunch of instructions.	The process of modifying programs is very easy in low-level programs. Here, it can directly map the statements to the processor instructions.

Difference between syntax and semantics

Syntax	**Semantics**
It defines the rules and regulations that helps write any statement in any programming language	It refers to the meaning of the associated line of code in a programming language.
It doesn't have any relationship with the meaning of the statement.	It tells about the meaning, and helps interpret what function the line of code/program is performing
It is associated with the grammar and structure of the programming language.	Even when a statement has correct syntax, it wouldn't do the function that was intended for it to do
A line of code is syntactically valid and correct if it follows all the rules of syntax.	Semantic errors are encountered at runtime.
Easy to catch the errors	Such errors are difficult to catch

Types of all preprocessor directives

As the name suggests, Preprocessors are programs that process our source code before compilation.

Preprocessor programs provide preprocessor directives that tell the compiler to preprocess the source code before compiling. All of these preprocessor directives begin with a '#' (hash) symbol. The '#' symbol indicates that whatever statement starts with a '#' will go to the preprocessor program to get executed.

There are 4 Main Types of Preprocessor Directives:

1. Macros
2. File Inclusion
3. Conditional Compilation
4. Other directives

Macro

Syntax: #define

This macro defines constant value and can be any of the basic data types. If the macro with the name '*macro_name*' is defined, then the block of statements will execute normally, but if it is not defined, the compiler will simply skip this block of statements.

Header file inclusion

Syntax: #include<file_name>

The source code of the file "file_name" is included in the main program at the specified place. This type of preprocessor directive tells the compiler to include a file in the source code program. There are two types of files that can be included by the user in the program: **Header files or Standard files**: These files contain definitions of pre-defined functions like **printf(), scanf(),** etc. These files must be included to work with these functions. Different functions are declared in different header files. For example, standard I/O functions are in the 'iostream' file whereas functions that perform string operations are in the 'string' file.

Conditional compilation

Syntax: #ifdef,
#endif,
#if,
#else,
#ifndef

Set of commands are included or excluded in source program before compilation with respect to the condition. Conditional Compilation directives are a type of directive that helps to compile a specific portion of the program or to skip the compilation of some specific part of the program

based on some conditions. This can be done with the help of the two preprocessing commands '**ifdef**' and '**endif**'.

Other directives
Syntax: #undef,
#pragma

#undef is used to undefine a defined macro variable.

#Pragma is used to call a function before and after main function in a C program. This directive is a special purpose directive and is used to turn on or off some features. This type of directives are compiler-specific, i.e., they vary from compiler to compiler.

#pragma startup and **#pragma exit**: These directives help us to specify the functions that are needed to run before program startup (before the control passes to main()) and just before program exit (just before the control returns from main()).

#pragma warn Directive: This directive is used to hide the warning message which is displayed during compilation. We can hide the warnings as shown below:

- **#pragma warn -rvl**: This directive hides those warnings which are raised when a function that is supposed to return a value does not return a value.
- **#pragma warn -par**: This directive hides those warnings which are raised when a function does not use the parameters passed to it.
- **#pragma warn -rch**: This directive hides those warnings which are raised when a code is unreachable. For example, any code written after the *return* statement in a function is unreachable.