NAME: **OMISORE DAMILOLA OLUWASENI**

MATRIC NO: **17/SCI01/068**

COURSE CODE: **CSC 312**

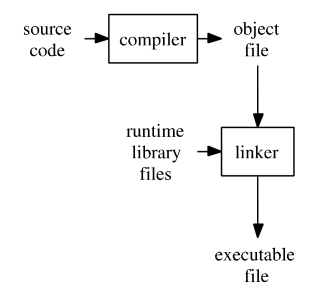
**ASSIGNMENT**

With the aid of a diagram, describe how a C++ code can be converted to Machine Language code.

**ANSWER**

A compiler takes the program code (source code) and converts the source code to a machine language module (called an object file). Another specialized program, called a linker, combines this object file with other previously compiled object files (in particular run-time modules) to create an executable file.

The following illustrates the programming process for a compiled programming language.



**STEP 1: PREPOCESSING**

When you write a source code file in C++, you include header files with extensions .h, .hxx, or .hpp, and sometimes with no extensions. You use the directive #include to mark a header file. The source file usually has the extension .cc, .cxx or .cpp.

In the first step of compilation, the compiler sends the code to a preprocessor. Now, a preprocessor is simply a [directive](https://msdn.microsoft.com/en-us/library/3sxhs2ty.aspx) that starts with #. So, #define, #include, #if, #then, #else and #line are some of the preprocessors with which the compiler interacts.

In this way, the compiler runs the preprocessor on each C++ source file. When it comes across #include, the preprocessor searches for the specified header file to include in the compilation. At this stage, the preprocessor also takes a look at conditional compilation blocks such as #ifdef, #ifndef, #endif, and removes code that won't be needed. These conditional directives let you include or discard a part of a program if a specific condition is met.

Overall, in the preprocessor stage, the source code file is temporarily expanded to prepare for compilation. This file has a greater number of lines that your simple source code. You can print this preprocessed file on stdout. Header files add bulk to the code. The more header files you include, the longer the preprocessed file will be.The preprocessor also adds some markers on the code to tell the compiler where each line came from. This helps to produce error messages that make sense to you.

**STEP 2: COMPILATION AND ASSEMBLY**

The next stage is that the compiler takes each output from the preprocessor and creates an object file from it in two steps.

First, it converts the pure C++ code (without any # directives) into assembly code. Assembly code is binary code that we can read.

Sometimes, it can be useful to read assembly code. It is the stage in which the compiler optimizes the source code - and does a better job of it than humans do. Let us look at how compilation works through an example. Compile this code to get the cpp-main.o object file, and look at the imported and exported symbols.

Next, the assembler converts the [assembly code](https://godbolt.org/) into bit code, line by line. The output of this stage is a binary file in format COFF, ELF, a.out and similar. You can always stop compiling at this point, which is a useful feature since you can compile each code separately.

You can put every object file that you get out of this process into archives called static libraries. Later, when you want to use these object files, you can simply pull them out of your libraries without having to recompile all the source files if you only change one file.

**STEP 3: LINKING**

You would not get the result without linking the object files that the assembler produced in the previous stage. It is the job of the linker to produce either a dynamic (or shared) library or an executable file. Let's take a look at each of these outputs.

Shared or dynamic libraries have nothing to do with static libraries, which we spoke about before. Static libraries are archives of object code linked with an end-user application, that can become a part of an executable.

[Dynamic libraries](http://www.geeksforgeeks.org/static-vs-dynamic-libraries/) are modules that contain data and functions that can be used by another application. The linker links all the object files by replacing all the references to undefined symbols with their correct addresses. Each symbol can be defined in other libraries or in object files. If the symbols are defined in a library that is not a standard library, then you need to let the linker know about it.

The stage of linking may also produce some errors. These errors are typically related to duplicate or missing definitions. Missing definitions are not only definitions that you didn't write; a definition could also be missing if you haven't given the linker any reference to the library or the object file where it could find the definition. Duplicate definition errors occur when two libraries or object files contain the definition of the same symbol.

These are the stages that compilation takes your code through. There are more complexities in the process that we don't have space for here, but knowing how compilation is done can help you prevent some weird bugs in your code. For instance, understanding preprocessing will help you make good use of header guards. Header guards are snippets of code you can use to protect the header file contents from multiple inclusions.

[Header guards](http://www.learncpp.com/cpp-tutorial/1-10a-header-guards/) can be placed using three pre-processor directives in a header file.