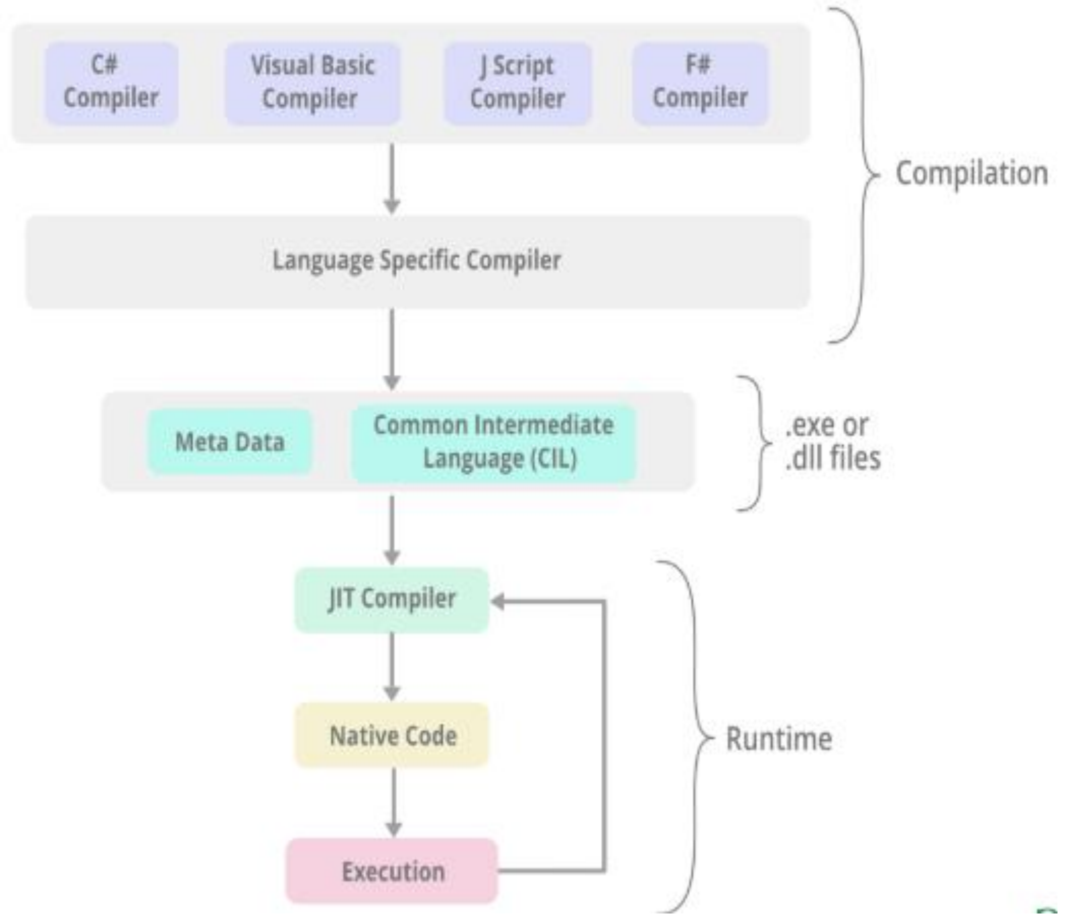
.NET Framework Introduction

Assignment

Q1) Demonstrate the process of conversion of Source code into the native machine code in .Net framework with the help of a flowchart.



.Net Framework provides runtime environment called Common Language Runtime (CLR).It provides an environment to run all the .Net Programs. The code which runs under the CLR is called as Managed Code. Programmers need not to worry on managing the memory if the programs are running under the CLR as it provides memory management and thread management.

The conversion of source code to machine code can be broadly be divided into 2 stages:

1. Compilation
2. Runtime

Compilation:

- The .Net framework has one or more language compilers, such as Visual Basic, C#, Visual C++, JScript, or one of many third-party compilers such as an Eiffel, Perl, or COBOL compiler.
- Any one of the compilers translate your managed source code into Microsoft Intermediate Language (MSIL) code also known as Common Intermediate Language (CIL) code.

Runtime:

- The Common Language Runtime (CLR) environment includes a JIT compiler for converting MSIL to native code.
- The JIT Compiler in CLR converts the MSIL code into native machine code that is then executed by the OS.
- The source code that is not managed by the Common Language Infrastructure (CLI) is executed directly by the OS i.e. code written in C/C++.
- During the runtime of a program the "Just in Time" (JIT) compiler of the Common Language Runtime (CLR) uses the Metadata and converts Microsoft Intermediate Language (MSIL) into native code.

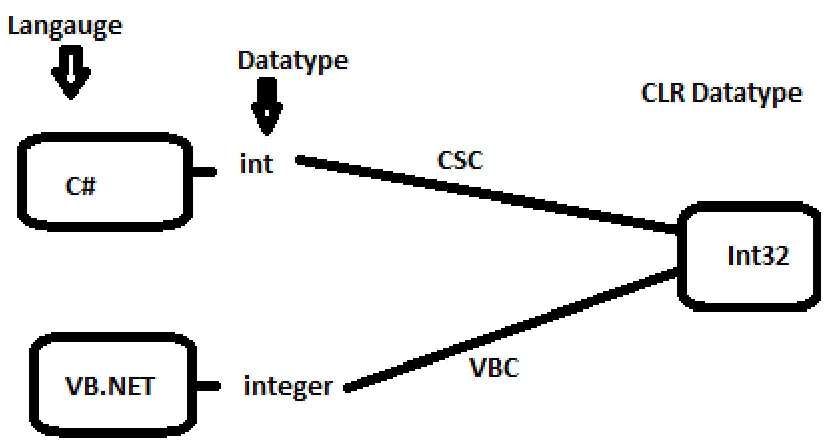
Q2) Explain CTS and how the .net framework implements CTS.

Common Type System (CTS) is the underlying subset of the CLI framework that) is a standard for defining and using data types in the .NET framework. .NET framework supports multiple languages that contain a type system that is common for all the languages. The languages which are CTS compliant do not require any type of conversion when calling the source code which is written in one language from within the code written in another programming language.

The main functions of CTS are

- Enables cross-language integration, type safety, and high-performance code execution.
- Provides an object-oriented model for the implementation of many programming languages.
- Defines rules that every language must follow which runs under .NET framework. It ensures that objects written in different .NET languages like C#, VB.NET, F# etc. can interact with each other.

CTS provide set of data types for all the languages supported by .NET framework. CTS use the data type System.Int32 to represent a 4 bytes integer. VB.NET uses the alias Integer for the same; while C#.NET uses the alias int. This conversion is shown below.



Similarly a Boolean variable is denoted by a "bool" in C# and "Boolean" in VBNet. Since, CTS is language agnostic, the code written in different languages are able to interact using the Common Intermediate Language (CIL).

The types in CTS can be categorised as value and reference types with these being analogous to those used in C/C++.

The CTS supports the following types of data:

- Class
- Structure
- Enum
- Interface
- Delegate

Q3 Name at least 3 runtime services provided by CLR and explain their role in .net framework.

3 of the many runtime services provided by CLR are:

- **Garbage Collection**

  Garbage collection is an automated process of Common Language Runtime (CLR) to manage memory by the allocation of memory for live objects and releasing memory for dead objects. Garbage collection can certainly improve performance but not in all cases. Real-time embedded systems can be problematic to automated memory management, the challenge is the successful deployment of safe scheduling of tasks.

The GC comes into picture when we need to free the memory allocated to an object which has been initialised and utilised completely. The GC assumes that the new objects are likely to live shorter than the older objects and thus, divides them into generations.

Generations: The heap is categorized into generations for reclamations of short-lived objects. The long-lived objects might not be reclaimed because they can cost performance drawback. There are three categories: Generation 0 which contains mainly temporary and short-lived objects targeted by the GC, Generation 1 and Generation 2 which contains medium and long lived objects like static objects likely to sustain the GC.

Phases of GC:

- Marking of all objects with their generation and creating a list.
- Relocating the live objects and the reference to them is updated in the next phase.
- Compacting the space reclaimed from dead objects and reallocation of live objects.

- **Exception Handling**

An exception is a problem that arises during the execution of a program. An exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero. In the .Net Framework, exceptions are represented by classes. The exception classes in .Net Framework are mainly directly or indirectly derived from the System.Exception class. Some of the exception classes derived from the System.Exception class are the System.ApplicationException and System.SystemException classes.

The System.ApplicationException class supports exceptions generated by application programs. So, the exceptions defined by the programmers should derive from this class.

The System.SystemException class is the base class for all predefined system exception. .Net provides a structured solution to the exception handling problems in the form of try and catch blocks. Using these blocks the core program statements are separated from the error-handling statements. These error handling blocks are implemented using the Try, Catch and Finally keywords.

- **Base Class Library Support**

A .NET Framework library, BCL is the standard for the C# runtime library and one of the Common Language Infrastructure (CLI) standard libraries. BCL stands for Base class library also known as Class library (CL). It is the collection of reusable types that are closely integrated with CLR. BCL provides types representing the built-in CLI data types, basic file access, collections, custom attributes, formatting, security attributes, I/O streams, string manipulation, and more. It also helps in performing day to day operation e.g. dealing with string and primitive types, database connection, IO operations, etc.

Q4 What are the differences between Library vs DLL vs Exe? Explain.

Firstly, we try to understand what is meant by these three terms.

Libraries are used when we may have a code that we want to use in many programs. For example, if we write a function that counts the number of zeros in an array, that function will be useful in lots of programs. Once we get that function working correctly, we don't want to have to recompile the code every time we use it, so we put the executable code for that function in a library, and the linker can extract and insert the compiled code into your program. Static libraries are sometimes called 'archives' for this reason.

Dynamic Linking Libraries (DLLs) enhance the efficiency of Libraries as they do not need to be compiled every time a program runs and thus, reduces the redundancy of having multiple copies of the same library function by mapping it during runtime when loaded into the memory. Thus, they can be used as and when required as they are preinstalled in every machine the program runs on.

Executable File (.EXE) contains a main function as an entry point and is in itself an independent application which contains several DLL files. Usually, in an application package, there is atleast one EXE file which may contain several or none DLLs.

The main differences highlighted above are:

- Libraries are static whereas DLLs are dynamically mapped.
- DLLs save up space and are more reusable but are more prone to versioning problem as compared to Libraries.
- EXE can be executed independently whereas DLLs need to be reused in other applications. Thus, an EXE has an entry point which is missing in a DLL.
- DLLs shares the process and memory space of the calling application whereas EXE creates its own.

5. How does CLR in .net ensure security and type safety? Explain.

The common language runtime and the .NET provide many useful classes and services that enable developers to easily write secure code and enable system administrators to customize the permissions granted to code so that it can access protected resources. In addition, the runtime and the .NET provide useful classes and services that facilitate the use of cryptography and role-based security.

Type safety

Type-safe code accesses only the memory locations it is authorized to access. (For this discussion, type safety specifically refers to memory type safety and should not be confused with type safety in a broader respect.) For example, type-safe code cannot read values from another object's private fields. It accesses types only in well- defined, allowable ways.

During just-in-time (JIT) compilation, an optional verification process examines the metadata and Microsoft intermediate language (MSIL) of a method to be JIT-compiled into native machine code to verify that they are type safe. This process is skipped if the code has permission to bypass verification.

Although verification of type safety is not mandatory to run managed code, type safety plays a crucial role in assembly isolation and security enforcement. When code is type safe, the common language runtime can completely isolate assemblies from each other. This isolation helps ensure that assemblies cannot adversely affect each other and it increases application reliability. Type-safe components can execute safely in the same process even if they are trusted at different levels. When code is not type safe, unwanted side effects can occur. For example, the runtime cannot prevent managed code from calling into native (unmanaged) code and performing malicious operations.

When code is type safe, the runtime's security enforcement mechanism ensures that it does not access native code unless it has permission to do so.

Security

The common language runtime of the .NET Framework has its own secure execution model that isn't bound by the limitations of the operating system it's running on. In addition, the CLR enforces security policy based on where code is coming from rather than who the user is. This model is called code access security.

CLR implements its own secure execution model that is independent of the host platform. Beyond the benefits of bringing security to platforms that have never had it, this also is an opportunity to impose a more component-centric security model that takes into account the nature of dynamically composed systems.

The CLR implements a code access security model in which privileges are granted to code, not users. Upon loading a new assembly, the CLR gathers evidence about the code's origins. This evidence is associated with the in-memory representation of the assembly by the CLR and is used by the security system to determine what privileges to grant to the newly loaded code. This determination is made by running the evidence through a security policy. The security policy accepts evidence as input and produces a permission set as output. To avoid performance hits, security policy is typically not run until an explicit security demand is made.