**Managed C++ (C++/CLI)**

1. **Explain the difference between managed and unmanaged code?**

MANAGED CODE is a code created by the .NET compiler. It does not depend on the architecture of the target machine because it is executed by the CLR (Common Language Runtime), and not by the operating system itself.

***CLR and managed code offers developers few benefits, like garbage collection, type checking and exceptions handling.***

UMANAGED CODE is directly compiled to native machine code and depends on the architecture of the target machine. It is executed directly by the operating system. In the unmanaged code, the developer has to make sure he is dealing with memory usage and allocation (especially because of memory leaks), type safety and exceptions manually.

***In .NET, Visual Basic and C# compiler creates managed code. To get unmanaged code, the application has to be written in C or C++.***

***COM components, ActiveX interfaces, and Windows API functions are examples of unmanaged code.***

**System.Runtime.InteropServices** namespace provides a collection of classes useful for accessing COM objects, and native APIs from .NET.

1. **How to access an exported function of a native Dll from .NET**

The **DllImportAttribute** attribute provides the information needed to call a function exported from an unmanaged DLL.

[DllImport("vcmowrap.dll")]

public static extern Int32 wcmo\_init(…);

**What is marshalling?**

Marshalling is the process of creating a bridge between managed code and unmanaged code. In managed code, you have a String, while in the unmanaged world strings can be Unicode, non-Unicode, AnsiBstr etc.

**MarshalAs** attribute to specify what is the expected type on unmanaged side.

***for example***

[DllImport("somenativelibrary.dll")]

static extern int MethodA([MarshalAs(UnmanagedType.LPStr)] string parameter);

public static class Marshal Provides a collection of methods for allocating unmanaged memory, copying unmanaged memory blocks, and converting managed to unmanaged types, as well as other miscellaneous methods used when interacting with unmanaged code.

1. **How to convert managed string to unmanaged char\***

String^ name = L”Hello World”;

IntPtr^ intptr = **Marshal::StringToHGlobalAnsi**(name);

char \*str = (char \*)(intptr->ToPointer());

public static IntPtr StringToHGlobalAnsi (string? s);

**Marshal::StringToHGlobalAnsi(…)** method to copy the Unicode string to unmanaged memory as an ANSI (one-byte) character. The method returns an **IntPtr** object that points to the beginning of the unmanaged string.

1. **How to convert unmanaged char\* to managed string**

**Marshal::PtrToStringAnsi(…)** method Copies all characters up to the first null character from an unmanaged ANSI or UTF-8 string to a managed String, and widens each character to UTF-16.

char\* name = “Hello World”;

IntPtr^ intptr = gcnew IntPtr((void \*) name);

String^ str = Marshal::PtrToStringAnsi(\*intptr);

1. **How to copy unmanaged array to managed array**

public static void Copy (IntPtr source, int[] destination, int startIndex, int length);

|  |
| --- |
| int src\_arr[] = {10,20,30}; // unmanaged array  IntPtr^ intptr = gcnew IntPtr((void \*) src\_arr);  array<Int32>^ targ\_arr = gcnew array< Int32 >(3);  Marshal::Copy (\*intptr, targ\_arr,0,3); |

1. **How to pass managed array to unmanaged array**

Declares a pinning pointer, which is used only with the common language runtime.

A pinning pointer is an interior pointer that prevents the object pointed to from moving on the garbage-collected heap. That is, the value of a pinning pointer is not changed by the common language runtime.

This is required when you pass the address of a managed class to an unmanaged function so that the address will not change unexpectedly during resolution of the unmanaged function call.

array<double>^ arr = gcnew array<double>(5);

pin\_ptr<double> pinptr = &arr[0];

double \*ptr = NULL; // Unmanaged pointer // void fun(double ptr[])

ptr = pinptr;

1. **Explain about IntPtr**

A platform-specific type that is used to represent a pointer or a handle. IntPtr size is 32 bits in 32-bit hardware, and 64 bits in 64-bit hardware.

***for example***

// to do //

<https://docs.microsoft.com/en-us/dotnet/api/system.intptr?view=net-5.0>

**.NET Framework**

1. **Explain about garbage collection in .NET**

Garbage collector is a feature of CLR which cleans unused managed (it does not clean unmanaged objects) objects and reclaims memory. It’s a background thread which runs continuously and at specific intervals it checks if there are any unused objects.

**What are generations in garbage collector (Gen 0, Gen 1 and Gen 2)**

Gen 0: When application creates fresh objects, they are marked as Gen 0

Gen 1: When GC is not able to clear the objects from Gen 0 in first round it moves them to Gen 1 bucket

Gen 2: When GC visits Gen 1 objects and it is not able to clear them, it moves them Gen 2

Garbage collector will spend more time on Gen 0 objects rather than Gen 1 and Gen 2.

To enforce garbage collection in your code manually, you can run the following command (C#)

**System.GC.Collect();**

1. **What do the following acronyms in .NET stand for: IL, CIL, MSIL, CLI and JIT?**

**IL**, or Intermediate Language, is a CPU independent partially compiled code. IL code will be compiled to native machine code using current environmental properties by Just-In-Time compiler (JIT).

JIT compiler translates the IL code to an assembly code and uses the CPU architecture of the target machine to execute a .NET application. In .NET, IL is called Common Intermediate Language (CIL), and in the early .NET days it was called Microsoft Intermediate Language (MSIL).

**CLI**, or Common Language Infrastructure, is an open specification developed by Microsoft. It is a compiled code library used for deployment, versioning, and security.

In .NET there are two CLI types: process assemblies (EXE) and library assemblies (DLL). CLI assemblies contain code in CIL, and as mentioned, during compilation of CLI programming languages, the source code is translated into CIL code rather than into platform or processor specific object code.

* When compiled, source code is first translated to IL (in .NET, that is CIL, and previously called MSIL).
* CIL is then assembled into a bytecode and a CLI assembly is created.
* Before code execution, CLI code is passed through the runtime’s JIT compiler to generate native machine code.
* The computer’s processor executes the native machine code.

1. **Explain about static constructor?**

**a class can have static constructor.** **Static constructors are called automatically, immediately before any static fields are accessed, and are generally used to initialize static class members.** It is called automatically before the first instance is created or any static members are referenced. Static constructors are called before instance constructors.

* we cannot use access modifiers on static constructor.
* static constructors cannot have parameters.

**Give 2 scenarios where static constructors can be used?**

* A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.
* Static constructors are also useful when creating wrapper classes for unmanaged code, when the constructor can call the **LoadLibrary ()** method

1. **Explain singleton vs static class**

**// to do //**

1. **Explain about destructors, and finalize () in C#.Net**

Destructors are invoked automatically, and cannot be called explicitly.

Destructors cannot be used with structs.

In class hierarchy, destructors will be called in reverse order from most derived to least derived.

***for example, If I’m writing my code as…***

***class Abc***

***{***

***~Abc () { }***

***}***

Here compiler will convert the class Abc code as…

class Abc

{

protected override void Finalize ()

{

try

{

// …

}

finally

{

**base.Finalize();**

}

}

}

Here the compiler-generated Finalize method contains the destructor body inside try block, followed by a finally block, that calls base class finalize. This ensures that destructors always call its base class destructor.

So, our conclusion from this finalize is another name for destructors in C#.

1. **Explain about IDisposable interface (or) How to clean unmanaged objects of your class**

If the application is using expensive external resource (unmanaged types), it is recommended to explicitly release the resource before the garbage collector runs and frees the object.

We can do this by implementing the Dispose method from the IDisposable interface that performs the necessary clean-up for the object. This can considerably improve the performance of the application.

**class Abc:** **IDisposable**

**{**

// Called from outside

public void **Dispose ()**

**{**

// user wants to release the resources.

Dispose(true);

// We have done the cleanup already, there is nothing left for

// the finalizer(destructor) to do. so, lets **tell the GC not to call it later.**

GC. SuppressFinalize(this);

**}**

protected virtual void **Dispose (bool dispossing)**

**{**

if (dispossing == true)

**{**

//someone wants the deterministic release of all resources.

// let us release all the managed resource.

**}**

// release unmanaged resources, which will not be released by GC.

**}**

**~Abc ()**

**{**

// the object went out of scope, and finalized is called.

// let’s call dispose in to release unmanaged resources.

Dispose(false);

**}**

**};**

1. **Explain the difference between the Stack and the Heap.?**

Stack are stored value types (types inherited from System.ValueType), and in the Heap are stored reference types (types inherited from System.Object).

We can say the Stack is responsible for keeping track of what is actually executing and where each executing thread is (each thread has its own Stack).

The Heap, on the other hand, is responsible for keeping track of the data, or more precise objects.

1. **Explain the difference between boxing and unboxing. Provide an example.**

Boxing is the process of converting a value type to the type object, and unboxing is extracting the value type from the object. While the boxing is implicit, unboxing is explicit.

|  |
| --- |
| int i = 13;  object myObject = i; // boxing  i = (int)myObject; // unboxing |

1. **Discuss the difference between constants and read-only variables.**

constants and read-only variable share many similarities, but there are some important differences:

* **Constants are evaluated at the compile-time**, while the **readonly variables are evaluated at the runtime.**
* Constants support only value-type variables, while read-only variables support value and reference type variables.
* **const fields are to be assigned at the time of declaration**, **readonly variables are declared as instance variable and assigned values in constructor.**
* Constants should be used when the value is not changing during the runtime, and read-only variables are used mostly when their actual value is unknown before the runtime.

1. **Explain the differences between an Interface and an Abstract Class in .NET?**

**An interface** merely declares a contract or a behaviour that implementing classes should have. It may declare only properties, methods, and events with no access modifiers. All the declared members must be implemented.

**An abstract class** provides a partial implementation for a functionality and some abstract/virtual members that must be implemented by the inheriting entities. It can declare fields too.

Neither interfaces nor abstract classes can be instantiated.

1. **Explain about extension methods in c#**

**It is the ability to add new methods, to the existing class without inheritance.**

***for example: -***

|  |  |  |
| --- | --- | --- |
| class Abc  {  *public void fun1 ()* { }  *public void fun2 ()* { }  } | static class Xyz  {  ***public static void fun3(this Abc a)***  **{ }**  ***public static void fun4*(this Abc a, string str)**  **{ }**  }  **Here “this” keyword is the binding parameter.** | Abc obj = new Abc ();  obj1.fun1();  obj1.fun2();  obj1.fun3();  obj1.fun4(“Hello”); |

* Extension methods are always defined as a static method, but when they are bound with any class or structure they will convert into non-static methods.
* When an extension method is defined with the same name and the signature of the existing method, then the compiler will print the existing method, not the extension method. Or in other words, the extension method does not support method overriding.
* You can also add new methods in the sealed class also using an extension method concept.
* It cannot apply to fields, properties, or events.

1. **Difference between int.parse(…), Convert.ToInt32(…) and int.TryParse(…)**

|  |  |  |  |
| --- | --- | --- | --- |
|  | int.Parse (string s) | Convert.ToInt32(string s) | bool bReturn = int.TryParse (string s, out result) |
| Integer in RANGE | returns integer value | returns integer value | result = int value  bReturn = true |
| Null value | ArguementNullException | returns "0" | result = 0  bReturn = false |
| Not in format | FormatException | FormatException | result = 0  bReturn = false |
| Value not in RANGE | OverflowException | OverflowException | result = 0  bReturn = false |

1. **Explain about delegate, and generic delegate?**

**A delegate is a type safe function pointer.** A delegate type references to methods with a particular parameter list and return type.

**A delegate instance associates with any method with a compatible signature and return type.**

* The purpose of writing a delegate is to create a plugin, which can be pluggable into any object.
* Delegates allows us to create reusable objects in C#
* delegate is a reference type, which can be passed as an argument to any method**.**

A multicast delegate is a delegate that has references to more than one function. When you invoke a multicast delegate, all the functions the delegate is pointing to, are invoked.

Observer design pattern can be implemented by the multicast delegate. Observer design pattern also known as publisher/subscriber design pattern.

1. **Explain about generics?**

Generics are introduced in C#2.0. Generics allows us to design classes and methods decoupled from the data types. Generics are similar to the template functions, and template classes in C++

* Use generic types to maximize **code reuse, type safety, and performance.**
* The most common use of generics is to create collection classes.
* The .NET class library contains several generic collection classes in the **System.Collections.Generic** namespace.
* We can create our own generic interfaces, classes, methods, events, and delegates.

1. **How to define a generic delegate?**

|  |
| --- |
| public delegate T AddDelegate<T>(T arg1, T arg2);  public delegate T3 AddDelegate<in T1, in T2, out T3>(T1 arg1, T2 arg2);    class Program  {  static void Main(string[] args)  {  AddDelegate<int> del1 = (x, y) => { return x + y; };  AddDelegate<int, double, double> del2 = (x, y) => { return x + y; };  Console.WriteLine(del1(10, 20));  Console.WriteLine(del2(10, 1.234));  }  } |

1. **Explain about Action, Func, Predicate delegates**

**Action delegate: -** It takes 1 to 16 input parameters, and doesn't have return type.

**Func delegate: -** It takes 0 to 16 input parameters, and returns a value.

**Predicate delegate: -** It takes 1 input parameter, and returns true/false.

|  |
| --- |
| Action<int, int> del1 = (x, y) => Console.WriteLine("Addition is: {0}", x + y);  del1(10, 20);  Func<string, string, string> del2 = (str1, str2) => { return str1 + " " + str2; };  Console.WriteLine(del2("Shaik", "Nazeeruddin"));  Func<string, string> del3 = (str1) => "Mr." + str1;  Console.WriteLine(del3("Nazeeruddin"));  Predicate<string> del4 = (str1) => str1.Equals(str1.ToUpper());  string str = "Benefice";  Console.WriteLine("{0} is Uppercase: {1}", str, del4(str)); |

1. **Explain Async, Await?**

async and await are the new keywords, which have been introduced in .NET framework 4.5

async await are markers, which mark code positions from where the control should resume after the task(thread) completes.

Wherever you write “await”, that method must be marked with “async” keyword.

|  |  |
| --- | --- |
| class Program  {  static void Main()  {  ProcessFile();  }  static async void ProcessFile( )  {  Task<int> task = new Task<int>(CountCharacters);  task.Start();  Console.WriteLine("Processing file. Please wait...");  int count = await task; // wait until the long task finishes.  Console.WriteLine(count.ToString() + " characters in file.");  }  } | static int CountCharacters ()  {  // reading a big file and returns the count of characters  Thread.Sleep(5000);  return count;  } |

1. **Explain Lazy loading with code?**

Lazy loading is a concept where we delay the loading of the object until the point where we need it.

***for example***

A customer has a list of orders (which is loaded from database/file). While creating the object of customer, we are delaying the loading of orders. Whenever we need the orders, then only we will load the orders.

**Advantages**

* Minimizes start up time of the application.
* Application consumes less memory because of on-demand loading.
* Unnecessary database SQL execution is avoided.

1. **Explain Dependency Injection?**

[**https://www.tutorialsteacher.com/ioc/lifetime-manager-in-unity-container**](https://www.tutorialsteacher.com/ioc/lifetime-manager-in-unity-container)

**Dependency injection** allows program designs to be loosely coupled, and follow the ***dependency inversion principle.*** Dependency Injection, is a design pattern in which one or more dependencies (services) are injected into a dependent object (client).

Dependency Injection Involves 3 types of classes.

* **Client Class:** The client class (dependent class) is a class which depends on the service class
* **Service Class:** The service class (dependency) is a class that provides service to the client class.
* **Injector Class:** The injector class injects the service class object into the client class.

**Injector class injects the service (dependency) to the client (dependent).** The injector class injects dependencies in three ways: through a constructor, through a property, or through a method.

***for example***

***CustomerService(dependency)*** object injected into the ***CustomerBusinessLogic(dependent)*** object by the ***Injector (UI Application).***

1. **Explain about Unity Container, how it is used to resolve dependency objects, in dependency injection?**

The Unity Container (Unity) is a lightweight, extensible dependency injection container. It's an open source IoC container for .NET applications supported by Microsoft.

**Advantages: -**

* Simplified type-mapping registration for interface type or base type.
* Supports registration of an existing instance.
* Supports code-based registration as well as design time registration.
* Automatically injects registered type at runtime through a constructor, a property or a method.

**RegisterType**<…>, **RegisterInstance**<…>, **Resolve**<…> methods will be called, on **UnityContainer** object.

1. **Explain covariance and contravariance?**

Covariance and contravariance introduced in C# 4.0

**Covariance and contravariance** enable implicit reference conversion for array types, delegate types, and generic type arguments. Covariance preserves assignment compatibility and contravariance reverses it.

We can say Covariance and Contravariance are the polymorphism extensions to array types, delegate types, and generic type

**Covariance deals with return type. For example, I have a delegate**

public delegate Base MyDelegate (Derived d); // Derived class derived from Base

Here, I have a method

|  |
| --- |
| static Derived Method (Derived d)  {  Console.WriteLine("Method");  return new Derived ();  } |

**MyDelegate d = Method; // it allows, it is covariance**

**Another example**

**IEnumerable<Base> objs = new List<Derived> (); // it will throw exception in prior to C#4.0, and will work fine in C#4.0**

**Contravariance deals with parameters. For example, I have a delegate**

public delegate Base MyDelegate (Derived d);

Here, I have a method

|  |
| --- |
| static Derived Method2(Base d)  {  Console.WriteLine("Method");  return new Derived ();  } |

**MyDelegate d = Method; // contravariance, (params) Less Derived type assigned to More derived type**

1. **Singleton Vs Static Class**

* Singleton has an instance/object while static class is a bunch of static methods
* Singleton can be extended e.g., through an interface while static class can't be.
* Singleton can be inherited which supports open/close principles in SOLID principles on the other hand static class can't be inherited and we need to make changes in itself.
* Singleton object can be passed to methods while static class as it does not have instance can't be passed as parameters

1. **Difference between var and dynamic keyword?**

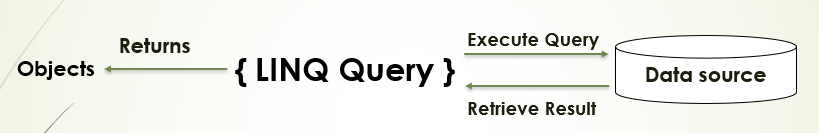
|  |  |
| --- | --- |
| **Var** | **Dynamic** |
| It is introduced in C# 3.0 | It is introduced in C# 4.0 |
| It is statically typed, and type of the variable decided at compile-time by the compiler | It is dynamically typed, and type of the variable decided at run-time by the compiler. |
| If it is not initialized, It will throw an error | If it is not initialized, it will throw an error. |
| It supports intelliSense in visual studio | It does not support intellisense in visual studio |
| It cannot be used for properties or returning values from the function. It can only be used as local variable | It can be used for properties or return values from the function. |

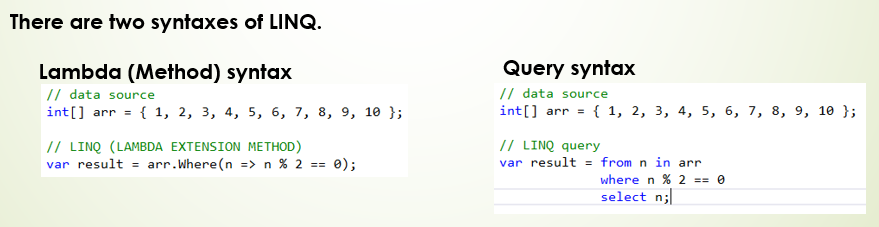
**Explain about LINQ**

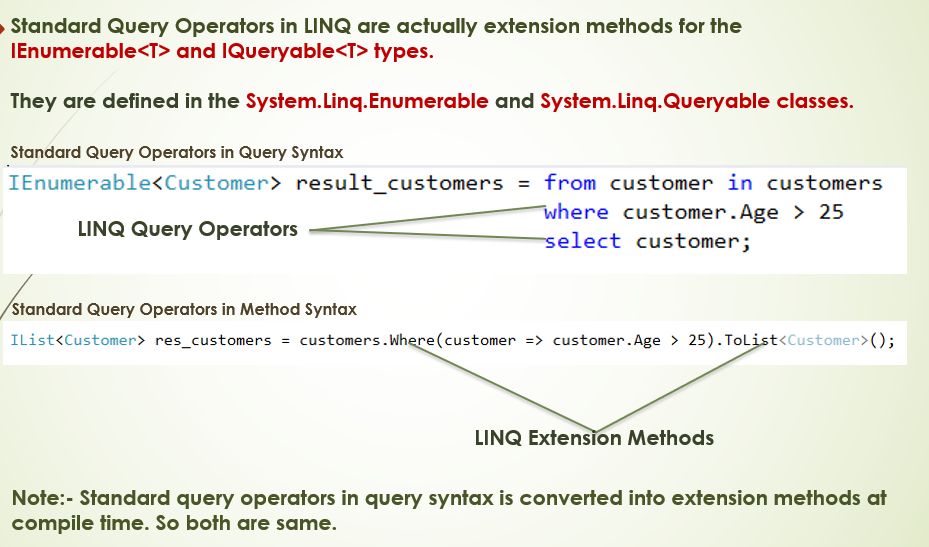
LINQ is an acronym for Language Integrated Query, and was introduced with Visual Studio 2008.

LINQ (Language Integrated Query) is uniform query syntax in C# and VB.NET to retrieve data from different sources and formats.

Supported data sources are: .NET Framework collections, SQL Server databases, ADO.NET Datasets, XML documents, and any collection of objects that support IEnumerable or the generic IEnumerable<T> interface.

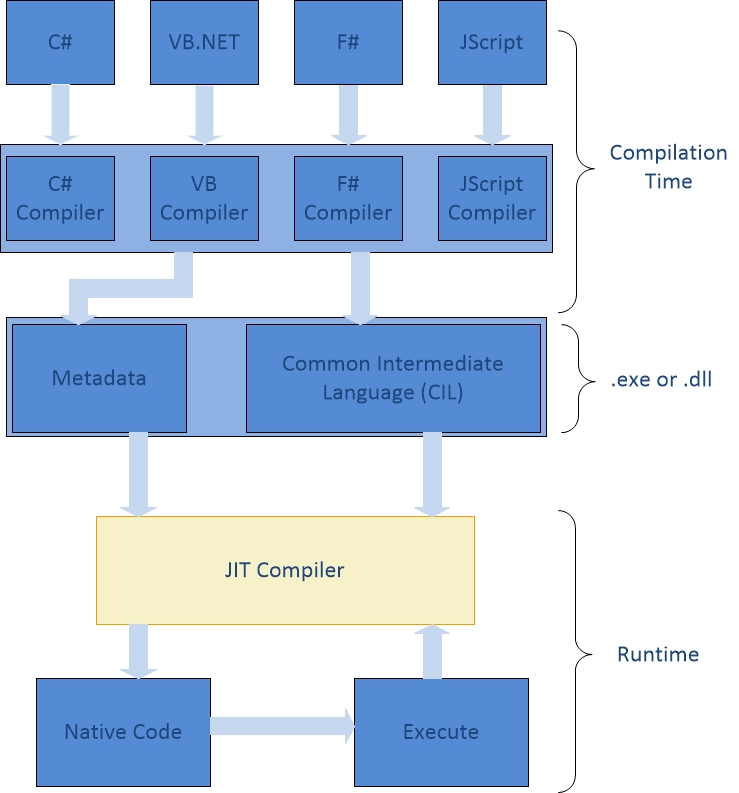






<https://ankitsharmablogs.com/csharp-coding-questions-for-technical-interviews/>

Explain about Unit Testing, Integration Testing, Acceptance Testing, and Mock Testing



**COMPILATION AND EXECUTION**

C# Source VB.NET

Source Code Source Code

COMPILATION

ASSEMBLY containing IL CODE

ASSEMBLY

Containing IL CODE

.NET base classes

|  |
| --- |
| **CLR ORGANIZES:**  JIT compilation  Security permissions granted  Memory type safety checked  Creates App Domain |

Assemblies loaded

EXECUTION

|  |
| --- |
| PROCESS |
| Application domain CODE EXECUTES HERE |
|  |

COM interop services

Garbage collector cleans up sources

Legacy COM component