1. Когда говорят C#, нередко имеют в виду технологии платформы .NET (Windows Forms, WPF, ASP.NET, .NET MAUI). И, наоборот, когда говорят **.NET,** нередко имеют в виду C#. Однако, хотя эти понятия связаны, отождествлять их неверно. Язык C# был создан специально для работы с фреймворком .NET, однако само понятие .NET несколько шире.

Можно выделить следующие ее основные черты:

Поддержка нескольких языков. Основой платформы является общеязыковая среда исполнения Common Language Runtime (CLR), благодаря чему .NET поддерживает несколько языков: наряду с C# это также VB.NET, C++, F#, а также различные диалекты других языков, привязанные к .NET, например, Delphi.NET. При компиляции код на любом из этих языков компилируется в сборку на общем языке CIL (Common Intermediate Language) - своего рода ассемблер платформы .NET. Поэтому при определенных условиях мы можем сделать отдельные модули одного приложения на отдельных языках.

Кроссплатформенность. .NET является переносимой платформой (с некоторыми ограничениями). Например, последняя версия платформы на данный момент - .NET 8 поддерживается на большинстве современных ОС Windows, MacOS, Linux. Используя различные технологии на платформе .NET, можно разрабатывать приложения на языке C# для самых разных платформ - Windows, MacOS, Linux, Android, iOS, Tizen.

Мощная библиотека классов. .NET представляет единую для всех поддерживаемых языков библиотеку классов. И какое бы приложение мы не собирались писать на C# - текстовый редактор, чат или сложный веб-сайт - так или иначе мы задействуем библиотеку классов .NET.

Разнообразие технологий. Общеязыковая среда исполнения CLR и базовая библиотека классов являются основой для целого стека технологий, которые разработчики могут задействовать при построении тех или иных приложений. Например, для работы с базами данных в этом стеке технологий предназначена технология ADO.NET и Entity Framework Core. Для построения графических приложений с богатым насыщенным интерфейсом - технология WPF и WinUI, для создания более простых графических приложений - Windows Forms. Для разработки кроссплатформенных мобильных и десктопных приложений - Xamarin/MAUI. Для создания веб-сайтов и веб-приложений - ASP.NET и т.д.

К этому стоит добавить активной развивающийся и набирающий популяность Blazor - фреймворк, который работает поверх .NET и который позволяет создавать веб-приложения как на стороне сервера, так и на стороне клиента. А в будущем будет поддерживать создание мобильных приложений и, возможно, десктоп-приложений.

Производительность. Согласно ряду тестов веб-приложения на .NET в ряде категорий сильно опережают веб-приложения, построенные с помощью других технологий. Приложения на .NET в принципе отличаются высокой производительностью.

Также еще следует отметить такую особенность языка C# и фреймворка .NET, как автоматическая сборка мусора. А это значит, что нам в большинстве случаев не придется, в отличие от С++, заботиться об освобождении памяти. Вышеупомянутая общеязыковая среда CLR сама вызовет сборщик мусора и очистит память.

**.NET Framework и .NET 8**

Стоит отметить, что .NET долгое время развивался премущественно как платформа для Windows под названием .NET Framework. В 2019 вышла последняя версия этой платформы - .NET Framework 4.8. Она больше не развивается

С 2014 Microsoft стал развивать альтернативную платформу - .**NET Core**, которая уже предназначалась для разных платформ и должна была вобрать в себя все возможности устаревшего .NET Framework и добавить новую функциональность. Затем Microsoft последовательно выпустил ряд версий этой платформы. И текущей версией является расматриваемая в этом руководстве платформа .NET 8. Поэтому следует различать .NET Framework, который предназначен преимущественно для Windows, и кросплатформенный .NET 8. В данном руководстве речь будет идти о C# 12 в связке с .NET 8, поскольку это актуальная платформа.

**ASP.NET Core** является кроссплатформенной, высокопроизводительной средой с открытым исходным кодом для создания современных облачных приложений, подключенных к Интернету.

ASP.NET Core позволяет выполнять следующие задачи:

* Создавать веб-приложения и службы, приложения Интернета вещей (IoT) и серверные части для мобильных приложений.
* Использовать избранные средства разработки в Windows, macOS и Linux.
* Выполнять развертывания в облаке или локальной среде.
* Запустите в .NET.

Преимущества, обеспечиваемые ASP.NET Core

ASP.NET Core предоставляет следующие преимущества:

* Единое решение для создания пользовательского веб-интерфейса и веб-API.
* Разработано для тестируемости.
* Razor Pages упрощает написание кода для сценариев страниц и повышает его эффективность.
* Blazor позволяет использовать в браузере язык C# вместе с JavaScript. совместное использование серверной и клиентской логик приложений, написанных с помощью .NET;
* Возможность разработки и запуска в ОС Windows, macOS и Linux.
* Открытый исходный код и ориентация на сообщество.
* Интеграция современных клиентских платформ и рабочих процессов разработки.
* Поддержка размещения служб удаленного вызова процедур (RPC) с помощью gRPC.
* Облачная система конфигурации на основе среды.
* Встроенное введение зависимостей.
* Упрощенный высокопроизводительный модульный конвейер HTTP-запросов.

Возможность размещения на следующих узлах:

Kestrel

IIS

HTTP.sys

Nginx

Docker

Управление параллельными версиями.

Инструментарий, упрощающий процесс современной веб-разработки.

**Common Intermediate Language (CIL)** — это байт-код, в какой

компилируется код на любом языке .NET. CIL не зависит от архитектуры процессора или операционной системы.

**Common Language Runtime (CLR)** — это виртуальная машина, исполняющая CIL-код. CLR обеспечивает такие функции, как сборка мусора, обработка исключений, безопасность и отладка.

**Common Type System (CTS) —** это набор правил для определения типов данных и их взаимодействия в CIL-коде. CTS гарантирует совместимость типов между разными языками.

**Common Language Specification (CLS)** — это подмножество CTS, которое определяет минимальный набор функций, которые должен поддерживать любой язык .NET. CLS обеспечивает взаимодействие кода на разных языках.

**Base Class Library (BCL)** — это набор стандартных классов и интерфейсов, которые предоставляют базовые функции для работы с файлами, строками, коллекциями, сетью.

Рассмотрим поэтапно:

* Когда программист пишет код на любом языке .NET, он использует BCL и другие библиотеки .NET.
* Затем он компилирует код в CIL-код с помощью специального компилятора для своего языка. Например, для C# используется компилятор csc.exe.
* Полученный CIL-код сохраняется в файле с расширением .exe или .dll.
* Потом пользователь запускает приложение на .NET, CLR загружает CIL-код в виртуальную машину CLR.
* CLR транслирует CIL-код в нативный код для конкретной платформы с помощью технологии JIT (Just-In-Time) компиляции.
* Результат: CIL-код становится исполняемым на любой платформе, где есть CLR.

3. **Стек** — это область оперативной памяти, которая создаётся для каждого потока. Он работает в порядке LIFO (Last In, First Out), то есть последний добавленный в стек кусок памяти будет первым в очереди на вывод из стека. Каждый раз, когда функция объявляет новую переменную, она добавляется в стек, а когда эта переменная пропадает из области видимости (например, когда функция заканчивается), она автоматически удаляется из стека. Когда стековая переменная освобождается, эта область памяти становится доступной для других стековых переменных.

Из-за такой природы стека управление памятью оказывается весьма логичным и простым для выполнения на ЦП; это приводит к высокой скорости, в особенности потому, что время цикла обновления байта стека очень мало, т.е. этот байт скорее всего привязан к кэшу процессора. Тем не менее, у такой строгой формы управления есть и недостатки. Размер стека — это фиксированная величина, и превышение лимита выделенной на стеке памяти приведёт к переполнению стека. Размер задаётся при создании потока, и у каждой переменной есть максимальный размер, зависящий от типа данных. Это позволяет ограничивать размер некоторых переменных (например, целочисленных), и вынуждает заранее объявлять размер более сложных типов данных (например, массивов), поскольку стек не позволит им изменить его. Кроме того, переменные, расположенные на стеке, всегда являются локальными.

В итоге стек позволяет управлять памятью наиболее эффективным образом — но если вам нужно использовать динамические структуры данных или глобальные переменные, то стоит обратить внимание на кучу.

**Куча**

Куча — это хранилище памяти, также расположенное в ОЗУ, которое допускает динамическое выделение памяти и не работает по принципу стека: это просто склад для ваших переменных. Когда вы выделяете в куче участок памяти для хранения переменной, к ней можно обратиться не только в потоке, но и во всем приложении. Именно так определяются глобальные переменные. По завершении приложения все выделенные участки памяти освобождаются. Размер кучи задаётся при запуске приложения, но, в отличие от стека, он ограничен лишь физически, и это позволяет создавать динамические переменные.

Вы взаимодействуете с кучей посредством ссылок, обычно называемых указателями — это переменные, чьи значения являются адресами других переменных. Создавая указатель, вы указываете на местоположение памяти в куче, что задаёт начальное значение переменной и говорит программе, где получить доступ к этому значению. Из-за динамической природы кучи ЦП не принимает участия в контроле над ней; в языках без сборщика мусора (C, C++) разработчику нужно вручную освобождать участки памяти, которые больше не нужны. Если этого не делать, могут возникнуть утечки и фрагментация памяти, что существенно замедлит работу кучи.

В сравнении со стеком, куча работает медленнее, поскольку переменные разбросаны по памяти, а не сидят на верхушке стека. Некорректное управление памятью в куче приводит к замедлению её работы; тем не менее, это не уменьшает её важности — если вам нужно работать с динамическими или глобальными переменными, пользуйтесь кучей.

**4. Сборщик мусора**

Так, при использовании переменных типов значений в методе, все значения этих переменных попадают в стек. После завершения работы метода стек очищается.

При использовании же ссылочных типов, например, объектов классов, для них также будет отводиться место в стеке, только там будет храниться не значение, а адрес на участок памяти в хипе или куче, в котором и будут находиться сами значения данного объекта. И если объект класса перестает использоваться, то при очистке стека ссылка на участок памяти также очищается, однако это не приводит к немедленной очистке самого участка памяти в куче. Впоследствии сборщик мусора (garbage collector) увидит, что на данный участок памяти больше нет ссылок, и очистит его.

Сборщик мусора не запускается сразу после удаления из стека ссылки на объект, размещенный в куче. Он запускается в то время, когда среда CLR обнаружит в этом потребность, например, когда программе требуется дополнительная память.

Как правило, объекты в куче располагаются неупорядочено, между ними могут иметься пустоты. Куча довольно сильно фрагментирована. Поэтому после очистки памяти в результате очередной сборки мусора оставшиеся объекты перемещаются в один непрерывный блок памяти. Вместе с этим происходит обновление ссылок, чтобы они правильно указывали на новые адреса объектов.

Так же надо отметить, что для крупных объектов существует своя куча - Large Object Heap. В эту кучу помещаются объекты, размер которых больше 85 000 байт. Особенность этой кучи состоит в том, что при сборке мусора сжатие памяти не проводится по причине больших издержек, связанных с размером объектов.

Несмотря на то что, на сжатие занятого пространства требуется время, да и приложение не сможет продолжать работу, пока не отработает сборщик мусора, однако благодаря подобному подходу также происходит оптимизация приложения. Теперь чтобы найти свободное место в куче среде CLR не надо искать островки пустого пространства среди занятых блоков. Ей достаточно обратиться к указателю кучи, который указывает на свободный участок памяти, что уменьшает количество обращений к памяти.

Кроме того, чтобы снизить издержки от работы сборщика мусора, все объекты в куче разделяются по поколениям. Всего существует три поколения объектов: 0, 1 и 2-е.

К поколению 0 относятся новые объекты, которые еще ни разу не подвергались сборке мусора. К поколению 1 относятся объекты, которые пережили одну сборку, а к поколению 2 - объекты, прошедшие более одной сборки мусора.

Когда сборщик мусора приступает к работе, он сначала анализирует объекты из поколения 0. Те объекты, которые остаются актуальными после очистки, повышаются до поколения 1.

Если после обработки объектов поколения 0 все еще необходима дополнительная память, то сборщик мусора приступает к объектам из поколения 1. Те объекты, на которые уже нет ссылок, уничтожаются, а те, которые по-прежнему актуальны, повышаются до поколения 2.

Поскольку объекты из поколения 0 являются более молодыми и нередко находятся в адресном пространстве памяти рядом друг с другом, то их удаление проходит с наименьшими издержками. Функционал сборщика мусора в библиотеке классов .NET представляет класс System.GC.

**5. Обьявление переменных. Типы var, def, dynamic.**

Перед использованием любую переменную надо определить. Синтаксис определения переменной выглядит следующим образом:

Вначале идет тип переменной, потом ее имя. В качестве имени переменной может выступать любое произвольное название, которое удовлетворяет следующим требованиям:

* имя может содержать любые цифры, буквы и символ подчеркивания, при этом первый символ в имени должен быть буквой или символом подчеркивания
* в имени не должно быть знаков пунктуации и пробелов
* имя не может быть ключевым словом языка C#. Таких слов не так много, и при работе в Visual Studio среда разработки подсвечивает ключевые слова синим цветом.

Хотя имя переменой может быть любым, но следует давать переменным описательные имена, которые будут говорить об их предназначении.

Ключевое слово var в языке программирования C# предназначено для объявления переменных без явного указания их типа данных. Вместо этого компилятор C# автоматически определяет тип переменной на основе значения, которое ей присваивается при инициализации.

Dynamic - динамический тип данных. Это такой тип данных, который не имеет привязанного типа и в любой момент может быть переопределен другим типом.

При передаче параметров по ссылке перед параметрами используется модификатор ref

При передаче значений параметрам по ссылке метод получает адрес переменной в памяти. И, таким образом, если в методе изменяется значение параметра, передаваемого по ссылке, то также изменяется и значение переменной, которая передается на его место..

Так, в метод Increment передается ссылка на саму переменную number в памяти. И если значение параметра n в Increment изменяется, то это приводит и к изменению переменной number, так как и параметр n и переменная number указывают на один и тот же адрес в памяти.

Обратите внимание, что модификатор ref указывается как перед параметром при объявлении метода, так и при вызове метода перед аргументом, который передается параметру.

**6. Иммутабельные переменные(Константы)**

Константа должна быть обязательно инициализирована при определении, и после определения значение константы не может быть изменено

Константы предназначены для описания таких значений, которые не должны изменяться в программе. Для определения констант используется ключевое слово const, которое указывается перед типом константы.

При использовании констант надо помнить, что объявить мы их можем только один раз и что к моменту компиляции они должны быть определены.  
**7. Типы данных в C#**

Типы значений:

* Целочисленные типы (byte, sbyte, short, ushort, int, uint, long, ulong)
* Типы с плавающей запятой (float, double)
* Тип decimal
* Тип bool
* Тип char
* Перечисления enum
* Структуры (struct)

Ссылочные типы:

* Тип object
* Тип string
* Классы (class)
* Интерфейсы (interface)
* Делегаты (delegate)

Параметры и переменные метода, которые представляют типы значений, размещают свое значение в стеке. При создании объекта ссылочного типа в стеке помещается ссылка на адрес в куче (хипе). Когда объект ссылочного типа перестает использоваться, в дело вступает автоматический сборщик мусора: он видит, что на объект в хипе нету больше ссылок, условно удаляет этот объект и очищает память - фактически помечает, что данный сегмент памяти может быть использован для хранения других данных.

**8. Примитивные типы данных**

В языке C# есть следующие базовые типы данных:

* bool
* byte
* sbyte
* short
* ushort
* int
* uint
* long
* ulong
* float
* double
* decimal
* char
* string
* object

**9 .Ссылочные типы данных**

Ссылочными типами данных называют типы, которые не хранятся непосредственно в стеке(Там хранятся их адреса в хипе). Фрагментированные ссылочные переменных хранятся в куче, до момента работы сборщика мусора. Преимущество ссылочных типов заключает в месте их хранения: они не ограничены размерами примитивных переменных, а так же могут быть доступны в том или ином виде из любой точки программы.

В С# существуют следующие ссылочные типы данных:

* Обьект
* Класс
* Интерфейс
* Делегат

**10. Размеры и значения типов данных**

В языке C# есть следующие базовые типы данных:

* bool: хранит значение true или false (логические литералы). Представлен системным типом System.Boolean
* byte: хранит целое число от 0 до 255 и занимает 1 байт. Представлен системным типом System.Byte
* sbyte: хранит целое число от -128 до 127 и занимает 1 байт. Представлен системным типом System.SByte
* short: хранит целое число от -32768 до 32767 и занимает 2 байта. Представлен системным типом System.Int16
* ushort: хранит целое число от 0 до 65535 и занимает 2 байта. Представлен системным типом System.UInt16
* int: хранит целое число от -2147483648 до 2147483647 и занимает 4 байта. Представлен системным типом System.Int32. Все целочисленные литералы по умолчанию представляют значения типа int:
* uint: хранит целое число от 0 до 4294967295 и занимает 4 байта. Представлен системным типом System.UInt32
* long: хранит целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт. Представлен системным типом System.Int64
* ulong: хранит целое число от 0 до 18 446 744 073 709 551 615 и занимает 8 байт. Представлен системным типом System.UInt64
* float: хранит число с плавающей точкой от -3.4\*1038 до 3.4\*1038 и занимает 4 байта. Представлен системным типом System.Single
* double: хранит число с плавающей точкой от ±5.0\*10-324 до ±1.7\*10308 и занимает 8 байта. Представлен системным типом System.Double
* decimal: хранит десятичное дробное число. Если употребляется без десятичной запятой, имеет значение от ±1.0\*10-28 до ±7.9228\*1028, может хранить 28 знаков после запятой и занимает 16 байт. Представлен системным типом System.Decimal
* char: хранит одиночный символ в кодировке Unicode и занимает 2 байта. Представлен системным типом System.Char. Этому типу соответствуют символьные литералы:
* string: хранит набор символов Unicode. Представлен системным типом System.String. Этому типу соответствуют строковые литералы.
* object: может хранить значение любого типа данных и занимает 4 байта на 32-разрядной платформе и 8 байт на 64-разрядной платформе. Представлен системным типом System.Object, который является базовым для всех других типов и классов .NET.

Если превысить предел типа переменной, то переменная будет равна 0.

**11. Checked и Unchecked**

Контексты checked и unchecked влияют на реакцию компилятора на арифметическое переполнение. Если задан контекст checked арифметическое переполнение, например, в цикле, сгенерирует исключение, а если контекст unchecked то переполнение будет проигнорировано. Это помогает отследить и не допустить переполнение

какой либо переменной.

**12. Приведение типов. Boxing и unboxing.**

Операция преобразования типов предполагает указание в скобках того типа, к которому надо преобразовать значение.Преобразования могут быть сужающие (narrowing) и расширяющие (widening). Расширяющие преобразования расширяют размер объекта в памяти.Сужающие преобразования, наоборот, сужают значение до типа меньшей разядности.

**Неявные преобразования**

В случае с расширяющими преобразованиями компилятор за нас выполнял все преобразования данных, то есть преобразования были неявными (implicit conversion). Такие преобразования не вызывают каких-то затруднений. Тем не менее стоит сказать пару слов об общей механике подобных преобразований.

Если производится преобразование от безнакового типа меньшей разрядности к безнаковому типу большой разрядности, то добавляются дополнительные биты, которые имеют значени 0. Это называется дополнение нулями или zero extension.

Если производится преобразование к знаковому типу, то битовое представление дополняется нулями, если число положительное, и единицами, если число отрицательное. Последний разряд числа содержит знаковый бит - 0 для положительных и 1 для отрицательных чисел. При расширении в добавленные разряды компируется знаковый бит.

При явных преобразованиях (explicit conversion) мы сами должны применить операцию преобразования (операция ()). Суть операции преобразования типов состоит в том, что перед значением указывается в скобках тип, к которому надо привести данное значение:

**Явные преобразования**

Расширяющие преобразования от типа с меньшей разрядностью к типу с большей разрядностью компилятор проводит неявно. Это могут быть следующие цепочки преобразований:

* byte -> short -> int -> long -> decimal
* int -> double
* short -> float -> double
* char -> int

Упаковка (boxing) и распаковка (unboxing) — операции преобразования значимых типов данных в ссылочные и обратно. Эти операции могут негативно сказываться на производительности из-за дополнительных вычислений: выделения памяти под новый объект и копирования данных.

Упаковка из за сущности ссылочных типов очень полезна для получения доступа к нужному локальному значению.