**1) Ознакомьтесь с концепцией «небезопасного кода и указателей» в .NET. Познакомьтесь с ключевыми словами unsafe и fixed.**

Большая часть кода C#, который вы пишете, является «проверяемо безопасным кодом». Проверяемо безопасный код означает, что средства .NET могут проверить безопасность кода. Как правило, безопасный код не обращается непосредственно к памяти с помощью указателей. Он также не выделяет необработанную память. Вместо этого он создает управляемые объекты.

C# поддерживает [небезопасный](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/unsafe)контекст, в котором можно написать непроверяемый код. В контексте код может использовать указатели, выделять и освобождать блоки памяти, а также вызывать методы с помощью указателей функций. Небезопасный код в C# не обязательно опасен; это просто код, безопасность которого не может быть проверена.unsafe

В небезопасном контексте тип может быть типом указателя, в дополнение к типу значения, или ссылочному типу. Кроме того, бокс и распаковка не поддерживают указатели. Однако можно выполнять преобразование между различными типами указателей, а также между типами указателей и интегральными типами.object

При объявлении нескольких указателей в одном объявлении звездочка () пишется только вместе с базовым типом. Он не используется в качестве префикса к имени каждого указателя.

int\* p1, p2, p3; // Ok

int \*p1, \*p2, \*p3; // Invalid in C#

C# предоставляет типы [делегатов](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/reference-types#the-delegate-type)для определения объектов указателя безопасных функций. Вызов делегата включает создание экземпляра типа, производного от [System.Delegate,](https://docs.microsoft.com/en-us/dotnet/api/system.delegate) и вызов виртуального метода к его методу.

Указатель функции можно определить с помощью синтаксиса. Компилятор вызовет функцию, используя инструкцию, а не создав экземпляр объекта и вызов . Следующий код объявляет два метода, которые используют a или a для объединения двух объектов одного типа.

static int localMultiply(int x, int y) => x \* y;

int product = UnsafeCombine(&localMultiply, 3, 4);

**Оператор fixed и закрепление указателей**

Однако кроме структур в C# есть еще и классы, которые в отличие от типов значений, помещают все связанные значения в куче. И в работу данных классов может в любой момент вмешаться сборщик мусора, периодически очищающий кучу. Чтобы фиксировать на все время работы указатели на объекты классов используется оператор fixed.

Оператор fixed создает блок, в котором фиксируется указатель на поле объекта person. После завершения блока fixed закрепление с переменных снимается, и они могут быть подвержены сборке мусора.

**Ключевое слово unsafe**

Блок кода или метод, в котором используются указатели, помечается ключевым словом unsafe:

**2) Ознакомьтесь с работой сборщика мусора (garbage collector, GC) в .NET.**

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.

Функционал сборщика мусора в библиотеке классов .NET представляет класс System.GC. Через статические методы данный класс позволяет обращаться к сборщику мусора. Как правило, надобность в применении этого класса отсутствует. Наиболее распространенным случаем его использования является сборка мусора при работе с неуправляемыми ресурсами, при интенсивном выделении больших объемов памяти, при которых необходимо такое же быстрое их освобождение.

3) using

Конструкция using оформляет блок кода и создает объект некоторого класса, который реализует интерфейс IDisposable, в частности, его метод Dispose. При завершении блока кода у объекта вызывается метод Dispose.

У них вообще мало общего. Dispose нужен для освобождения ресурсов "здесь и сейчас" (не совсем так на самом деле. Вызов Dispose сигнализирует, что вы хотите освободить ресурс, но не факт, что это обязательно случится вот прямо тут же). Необходимость и преимущество интерфейса IDisposable именно в том, что его реализация позволяет освобождать ресурсы не тогда, когда до них доберется сборщик мусора, а тогда, когда это нужно программисту. Ресурсы могут быть дорогими, и держать их в памяти неопределенно долгое время может быть слишком расточительным. Деструкторов в C# нет вовсе. Есть *финализаторы*. Разница в том, что время вызова финализатора не определено. Стоит также отметить, что если Dispose предназначен для вызова вручную, то финализатор вручную вызвать нельзя. Это делается автоматически.