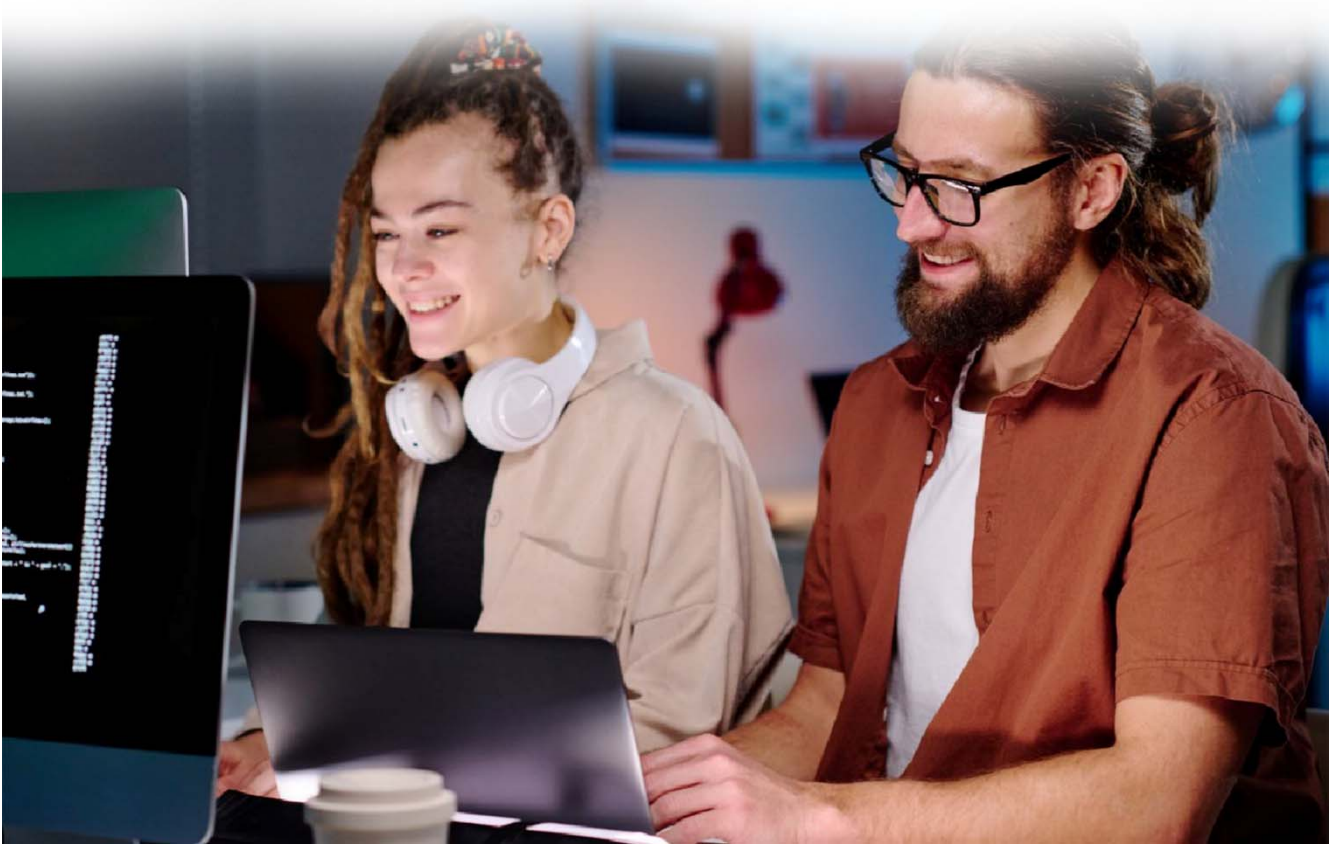
ROZDZIAŁ 6

GARBAGE COLLECTOR, WSKAŹNIKI, STRUKTURY



6 Garbage Collector, wskaźniki, struktury

6.1 Garbage Collector

GarbageCollector to jedna z najważniejszych funkcjonalności w języku C#. Mechanizm ten, pracuje w tle podczas działania programu. Jego zadaniem jest zarządzanie pamięcią programu poprzez usuwanie obiektów, których już nie używa się w kodzie.

Dzięki temu programista nie musi samodzielnie zarządzać pamięcią programu, co pozwala na większą swobodę i szybkość programowania. **GarbageCollector** działa poprzez przeglądanie sterty (*ang. heap*) programu, a następnie oznaczanie obiektów, które są do usunięcia.

GarbageCollector usuwa te obiekty, które są oznaczone w poprzednim kroku jako do usunięcia i zwalnia zajmowaną przez nie pamięć. Proces ten jest powtarzany w regularnych odstępach czasu, co pozwala na utrzymanie optymalnej wydajności programu. Warto zaznaczyć, że nie jest on idealnym mechanizmem i może wpływać na wydajność programu.

Gdy **GarbageCollector** usuwa niepotrzebne już obiekty, może zajmować więcej czasu, co prowadzi do spowolnienia działania programu. Aby zminimalizować ten problem, można wykorzystać kilka technik optymalizacyjnych. Pierwszą z nich jest unikanie alokowania dużej ilości obiektów. Im więcej obiektów jest alokowanych, tym więcej czasu **GarbageCollector** potrzebuje na ich usunięcie.

Dlatego warto unikać alokowania obiektów w pętlach i zwracać uwagę na to, jak często i jak wiele ich alokujemy w naszym kodzie. Kolejną techniką optymalizacyjną jest korzystanie z metod oznaczających obiekty do usunięcia, takich jak `Dispose()` lub `Finalize()`. Dzięki tym metodom można kontrolować moment, w którym `GarbageCollector` usunie obiekt, co pozwala na zoptymalizowanie procesu usuwania niepotrzebnych już obiektów.

Istnieje też inna technika optymalizacyjna, którą jest wykorzystanie profilera do analizy działania programu i znalezienia miejsc, w których **GarbageCollector** zajmuje najwięcej czasu. Dzięki temu można zlokalizować i zoptymalizować te fragmenty kodu, które powodują najwięcej problemów.

Ważne jest, aby zrozumieć, że **GarbageCollector** w języku C# działa nieco inaczej niż w innych językach programowania, ponieważ działa na stercie, a nie

na stosie (*ang. stack*). Sterta to obszar pamięci, w którym przechowywane są obiekty dynamiczne, a stos to inny obszar pamięci, w którym przechowywane są zmienne lokalne i argumenty funkcji. **GarbageCollector** działa w oparciu o generacje. Generacja 0 to pierwsza generacja, w której znajdują się najnowsze i najkrócej istniejące obiekty.

Generacja 1 to druga generacja, w której znajdują się obiekty, które przetrwały pierwszy cykl **GarbageCollectora**. Generacja 2 to trzecia generacja, w której znajdują się obiekty, które przetrwały dwa cykle **GarbageCollectora**. W języku C# można korzystać z kilku rodzajów kolektorów pamięci, takich jak np. kolektor pamięci konserwatywny, który działa na zasadzie przeglądania sterty i oznaczania obiektów, które są do usunięcia, a także kolektor pamięci z generacjami, który działa na zasadzie przeglądania poszczególnych generacji i usuwania obiektów z generacji, które są najdłużej niepotrzebne.

Pamięć jest jednym z najcenniejszych zasobów w programowaniu. Jeśli program zużywa za dużo pamięci, to może spowolnić działanie systemu operacyjnego i innych aplikacji. Dlatego warto dbać o optymalne wykorzystanie pamięci. Należy także pamiętać, że **GarbageCollector** działa nie tylko w języku C#. Podobne mechanizmy istnieją również w innych językach programowania, takich jak Java czy Python.

W każdym z tych języków, **GarbageCollector** działa nieco inaczej, ale zawsze ma na celu automatyczne zarządzanie pamięcią programu. Oprócz **GarbageCollector**, istnieją także inne mechanizmy, które umożliwiają zarządzanie pamięcią w programach. Na przykład, w języku C++ programista jest odpowiedzialny za ręczne zarządzanie pamięcią przy użyciu operatorów **`new`** i **`delete`**. Jest to bardziej skomplikowany proces niż **GarbageCollector**, ale daje programiście większą kontrolę nad wykorzystaniem pamięci.

6.2 Wskaźniki

W C# istnieją różne typy danych, które umożliwiają programistom manipulowanie wartościami i wykonywanie działań na nich. Jednym z tych typów są wskaźniki, które pozwalają na bezpośrednie manipulowanie pamięcią komputera.

Wskaźnik to zmienna, która przechowuje adres komórki pamięci komputera. Innymi słowy, wskaźnik wskazuje na miejsce w pamięci, gdzie znajduje się

wartość danej zmiennej. W C# wskaźniki są reprezentowane poprzez dodanie * po nazwie typu.

Dodatkowo, wskaźniki muszą być objęte blokiem **unsafe**. Przykładowo, jeśli mamy zmienną "**x**" typu **int**, to jej wartość będzie przechowywana w pewnym miejscu w pamięci. Możemy uzyskać adres tej zmiennej, używając operatora **&**. Na przykład:

```
static void Main(string[] args)
{
    unsafe
    {
        int x = 10;
        int* p = &x;
    }
}
```

Listing 6.1: Utworzenie wskaźnika p, przechowującego adres pamięci zmiennej x

W powyższym przykładzie zmienna **p** przechowuje adres zmiennej **x** w pamięci. Wskaźniki pozwalają na bezpośredni dostęp do pamięci komputera, co jest bardzo przydatne w niektórych sytuacjach. Jednakże, z powodu swojej natury, wskaźniki są też bardzo niebezpieczne, ponieważ jeśli zostaną źle użyte, mogą spowodować błędy i awarie programu.

Dlatego też, aby uniknąć niepożądanych skutków, C# narzuca pewne zasady dotyczące używania wskaźników. Na przykład, nie jest możliwe wykonywanie operacji arytmetycznych na wskaźnikach, co oznacza, że nie możemy dodawać, odejmować lub mnożyć wskaźników.

Kolejną zasada jest taka, że wskaźniki muszą mieć określony typ. Oznacza to, że nie możemy użyć wskaźnika typu **int**, aby wskazać na zmienną typu string. Wskaźniki są często używane do manipulowania tablicami. Możemy na przykład uzyskać adres pierwszego elementu tablicy i przesuwać go, aby uzyskać dostęp do innych elementów – taką operację można zrealizować w następujący sposób:

```
unsafe
{
    int[] array = new int[] { 1, 2, 3, 4, 5 };
    fixed (int* p = array)
    {
```

```
            for (int i = 0; i < array.Length; i++)
            {
                Console.WriteLine(*(p + i));
            }
        }
}
```

Listing 6.2: Użycie wskaźnika p, w celu uzyskania dostępu do kolejnych elementów tablicy

W powyższym przykładzie, używamy wskaźnika p, aby uzyskać dostęp do kolejnych elementów tablicy **`array`**. Aby uzyskać wartość elementu, musimy zrzutować wskaźnik na typ **`int*`**, a następnie odwołać się do wartości, używając operatora gwiazdki.

Należy zwrócić uwagę, że w kodzie oprócz bloku **`unsafe`**, znajduje się również blok **`fixed`**. Jest on jednym ze sposobów, aby zapewnić bezpieczeństwo typów w języku C# w przypadku pracy z niebezpiecznymi wskaźnikami, które umożliwiają bezpośredni dostęp do pamięci.

Głównym celem bloku **`fixed`** jest umożliwienie programiście zablokowania lokalizacji obiektu w pamięci podczas wykonywania określonych operacji z wykorzystaniem wskaźników, takich jak odczyt lub zapis wartości do pamięci.

Blok **`fixed`** służy do deklarowania zmiennej wskaźnika wskazującej na blok pamięci, który zostanie zablokowany w pamięci RAM i nie będzie przenoszony przez system śmieci. Dzięki temu, nawet jeśli **GarbageCollector** dokona innych przesunięć i przeniesie inne obiekty w pamięci, obiekt zadeklarowany w bloku **`fixed`** pozostanie na stałym miejscu w pamięci, co pozwala na bezpieczne korzystanie z wskaźników.

Wskaźniki są również przydatne w zarządzaniu pamięcią. Możemy użyć wskaźników do dynamicznego alokowania pamięci i zwalniania jej wtedy, gdy nie jest już potrzebna. Rozważmy poniższy przykład.

```
unsafe
{
    IntPtr p =
Marshal.AllocHGlobal(sizeof(int));

    // przypisanie wartości do zmiennej
    *((int*)p) = 10;
```

```
    // zwolnienie pamięci
    Marshal.FreeHGlobal(p);
}
```

Listing 6.3: Użycie wskaźnika p, do dynamicznej alokacji pamięci

W powyższym przykładzie, użyto funkcji **`Marshal.AllocHGlobal()`** do dynamicznego alokowania pamięci na zmienną typu **`int`**. Następnie, używając wskaźnika, przypisana została wartość do tej zmiennej, a na koniec, użyto funkcji **`Marshal.FreeHGlobal()`** do zwolnienia pamięci, gdy już nie jest potrzebna.

A co robi **`IntPtr`** w tym fragmencie kodu? To też jest wskaźnik, ale trochę innego typu. Typ **`IntPtr`** reprezentuje wskaźnik na dowolny obszar pamięci, zwykle adres w pamięci, ale może to być także indeks tablicy lub identyfikator innych obiektów systemu. Jest to typ bezpieczny i może być używany w kodzie bezpiecznym, ponieważ działa na zasadzie przekazywania wartości referencyjnych, zamiast bezpośredniego dostępu do pamięci. Natomiast typ **`int*`** reprezentuje niebezpieczny wskaźnik na wartość typu **`int`**.

Jest to typ niebezpieczny, ponieważ umożliwia bezpośredni dostęp do pamięci, co może prowadzić do błędów i nieoczekiwanego zachowania, w szczególności w przypadku dostępu do nieprawidłowych adresów pamięci. W przypadku kodu napisanego w C#, zaleca się unikanie bezpośredniego dostępu do pamięci i stosowanie typu **`IntPtr`**, który zapewnia bezpieczeństwo typu w języku C#.

Typ **`IntPtr`** może być używany do reprezentowania wskaźnika na adres pamięci, takiego jak adres elementu tablicy, a także innych typów wskaźników, takich jak identyfikator pliku lub identyfikator wątku. Czasem jednak, IDE samo doradza, jaki typ wskaźnika wybrać czy **`int*`** czy **`IntPtr`**.

Wskaźniki są również przydatne w przetwarzaniu binarnym. Można ich użyć do przetwarzania strumieni danych binarnych, takich jak pliki, porty szeregowe, zobaczmy, jak wygląda to w praktyce.

```
unsafe
{
    byte[] bytes = new byte[] { 0x01, 0x02,
0x03, 0x04 };
    fixed (byte* p = &bytes[0])
```

```
    {
        byte* temp = p;
        for (int i = 0; i < bytes.Length; i++)
        {
            Console.Write(*((byte*)temp) + " ");
            temp += 1;
        }
    }
}
```

Listing 6.4: Użycie wskaźnika, do przetworzenia danych binarnych w tablicy bytes

W powyższym przykładzie, używamy wskaźnika **p**, aby przetworzyć dane binarne w tablicy **bytes**. Aby uzyskać wartość bajtu, musimy zrzutować wskaźnik na typ **`byte*`**, a następnie odwołać się do wartości, używając operatora gwiazdki.

W powyższym kodzie nie da się bezpośrednio przesunąć wskaźnika **p** w pętli, ponieważ znajduje się on w bloku fixed. Dlatego została utworzona zmienna temp, która wskazuje na ten sam adres co **p**, a w pętli przesunięto **temp** o jeden **bajt**, aby wskazywała na kolejny element tablicy, by następnie odczytać wartość tego elementu przy użyciu operatora ***.

Wskaźniki w C# to potężne narzędzie, które pozwala na bezpośredni dostęp do pamięci komputera i umożliwia bardziej zaawansowane operacje w kodzie. Jednakże, należy zachować ostrożność podczas korzystania z wskaźników i zawsze stosować się do zasad bezpiecznego kodowania. W większości przypadków, w C# nie jest potrzebne użycie wskaźników, ponieważ język ten ma wbudowane funkcjonalności zarządzania pamięcią.

W złożonych aplikacjach, użycie wskaźników może znacznie przyspieszyć działanie kodu, szczególnie w przypadku operacji na dużych strukturach danych. W takich przypadkach, użycie wskaźników jest zasadne i może przynieść pozytywne rezultaty.

Wiedza o wskaźnikach w C# jest niezbędna dla programistów, którzy zajmują się tworzeniem oprogramowania dla systemów wbudowanych, sterowników lub innych aplikacji, w których ważne jest bezpośrednie zarządzanie pamięcią. Jednakże, dla większości programistów korzystających z C#, wiedza ta może być tylko dodatkiem do ich umiejętności programistycznych.

6.3 Struktury

W języku C# można użyć dwóch podstawowych typów wartościowych - typów prostych (np. **`int, bool, double`** itp.) oraz struktur. Struktury są tworzone przez programistów, aby reprezentować pojedyncze wartości, które mogą być przechowywane i przetwarzane jako jedna wartość, są one typami wartościowymi, co oznacza, że są przechowywane bezpośrednio w stosie, a ich instancje są kopiowane w całości podczas przypisania. Innymi słowy, struktury są przypisywane jako wartości, a nie jako wskaźniki.

Dzięki temu, struktury są szybsze niż klasy, ponieważ kopiowanie wartości jest szybsze niż kopiowanie wskaźnika do obiektu. Struktury są również bardziej wydajne, ponieważ nie muszą być zarządzane przez mechanizmy zarządzania pamięcią .NET Framework, takie jak **GarbageCollector**. Na przykład, w poniższym kodzie mamy dwie instancje struktury Point:

```
class Program
{
    static void Main(string[] args)
    {
        Point point1 = new Point();
        point1.x = 10;
        point1.y = 20;

        Point point2 = point1;
        point2.x = 30;

        Console.WriteLine(point1.x); // Wypisze
10
        Console.WriteLine(point2.x); // Wypisze
30

    }
}

struct Point
{
    public int x, y;
}
```

Listing 6.5: Struktura Point i jej dwie instancje

W tym przykładzie, zmiana wartości pola **x** w instancji **point2** nie ma wpływu na wartość pola **x** w instancji **point1**, ponieważ obie instancje są traktowane jako niezależne wartości. Struktury w C# definiujemy za pomocą słowa kluczowego **`struct`**. Jak widać, w tym przykładzie struktura znajduje się poza funkcją **`static void Main(string args[])`**, a nawet poza klasą Program –-i tak też powinno się implementować swoje własne struktury. Rozważmy teraz inny przykład:

```
public struct Person
{
    public string Name;
    public int Age;
}
```

Listing 6.6: Struktura Person zawierająca pola Name i Age

W powyższym przykładzie zdefiniowana została struktura **`Person`**, która składa się z dwóch pól: **`Name`** typu **`string`** i **`Age`** typu **`int`.** Te pola są publiczne, co oznacza, że mogą być odczytywane i zapisywane z zewnątrz struktury. Aby utworzyć nową instancję struktury, należy użyć słowa kluczowego **`new`** i przypisać wartości do jej pól. Na przykład:

```
Person person1 = new Person();
person1.Name = "Jan Kowalski";
person1.Age = 30;
```

Listing 6.7: Utworzenie nowej instancji struktury za pomocą operatora `new` oraz przypisanie wartości jej polom

Można również zainicjować wartości pól podczas tworzenia nowej instancji struktury, na przykład:

```
Person person2 = new Person
{
    Name = "Anna Nowak",
    Age = 25
};
```

Listing 6.8: Inicjacja pól podczas tworzenia nowej instancji struktury Person

Struktura składa się z pól, właściwości i metod. Przykładowa definicja struktury **`Point`** wygląda następująco:

```
struct Point
{
    public int x, y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public void Move(int dx, int dy)
    {
        x += dx;
        y += dy;
    }
}
```

Listing 6.9: Przykład struktury z metodami

W tej definicji struktury, mamy dwa pola – **x** i **y**, konstruktor i metodę **`Move()`**, która zmienia wartości pól **x** i **y**. Aby zadeklarować instancję struktury, używamy słowa kluczowego `new` i nazwy struktury, a następnie wywołujemy konstruktor struktury, jeśli istnieje, aby zainicjować wartości pól. Na przykład:

```
Point point = new Point(10, 20);
point.Move(5, 5);
```

Listing 6.10: Utworzenie instancji struktury za pomocą konstruktora i wywołanie na niej metody `Move()` z parametrami 5 i 5

W tym przykładzie, utworzono nową instancję struktury Point i przypisano ją do zmiennej point. Następnie wywołana została metoda **`Move()`**, która przesuwa punkt o **5** jednostek w poziomie i **5** jednostek w pionie. Warto wiedzieć, że można również przekazywać struktury jako argumenty do funkcji i zwracać je z nich. Rozważmy przykład z poniższego listingu:

```
public void DisplayPersonInfo(Person person)
{
    Console.WriteLine("Name: {0}, Age: {1}",
person.Name, person.Age);
}

public Person CreatePerson(string name, int age)
{
```

```
    Person person = new Person();
    person.Name = name;
    person.Age = age;
    return person;
}
```

Listing 6.11: Przekazanie struktur do funkcji i zwracanie struktur z nich

Ważne jest, aby pamiętać, że przekazywanie struktur jako argumentów do metod lub zwracanie ich z metod może prowadzić do kopii wartości. W przypadku dużych struktur może to prowadzić do spadku wydajności. Struktury nie obsługują dziedziczenia (nie były one jeszcze omawiane, bo to część programowania obiektowego, ale trzeba pamiętać o tym na przyszłość).

Struktury w C# nie mogą dziedziczyć po innych strukturach ani klasach, ani też nie mogą być dziedziczone przez inne struktury lub klasy. To ograniczenie może ograniczać elastyczność w projektowaniu aplikacji. Struktury nie obsługują dziedziczenia interfejsów – nie mogą dziedziczyć interfejsów ani być dziedziczone przez nie. Struktury są mniej elastyczne niż klasy.

W przeciwieństwie do klas, struktury nie mogą mieć konstruktora bezargumentowego ani destruktora. Są one zazwyczaj mniejsze od klas i w przeciwieństwie do klas, struktury są przechowywane bezpośrednio w stosie, co oznacza, że zajmują mniej miejsca w pamięci i są szybsze niż klasy. Ponieważ struktury są typami wartościowymi i są one przechowywane bezpośrednio w stosie to ich kopiowanie wartości jest szybsze niż kopiowanie wskaźnika do obiektu.

Struktury są również bardziej wydajne, ponieważ nie muszą być zarządzane przez mechanizmy zarządzania pamięcią **`.NET Framework`**, takie jak **`GarbageCollector`**.

Są one łatwe w użyciu ponieważ są typami wartościowymi łatwiejsze do zrozumienia, użycia i zazwyczaj mniejsze niż klasy, a także dobrze nadają się do przechowywania małych danych, takich jak współrzędne punktu, daty, liczby lub wartości boolowskie. Struktury są często stosowane w aplikacjach, gdzie konieczne jest przechowywanie danych w uporządkowany sposób, takim jak:

- **Przechowywanie współrzędnych** – struktury są często wykorzystywane do przechowywania współrzędnych w aplikacjach graficznych lub symulacjach.

- **Przechowywanie danych geograficznych** – struktury mogą być używane do przechowywania danych geograficznych, takich jak długość i szerokość geograficzna lub wysokość nad poziomem morza.
- **Przechowywanie daty i godziny** – struktury `DateTime` i `TimeSpan` w C# są często stosowane do przechowywania daty i godziny oraz wykonywania operacji na nich.
- **Przechowywanie danych finansowych** – struktury mogą być używane do przechowywania danych finansowych, takich jak kwota, waluta, itp.
- **Przechowywanie danych konfiguracyjnych** – struktury mogą być wykorzystywane do przechowywania danych konfiguracyjnych w aplikacjach, takich jak ustawienia, preferencje użytkownika, itp.

Struktury w C# obsługują wiele operatorów, takich jak:

- **Operator przypisania (=):** Przypisuje wartość jednej struktury do innej.
- **Operator równości (==):** Porównuje dwie struktury pod kątem równości.
- **Operator nierówności (!=):** Porównuje dwie struktury pod kątem nierówności.

Oprócz tych operatorów struktury obsługują również operatory arytmetyczne **(+, -, *, /)** oraz operatory logiczne **(&&, ||,!).** Dzięki temu można wykonywać różne operacje na strukturach, takie jak dodawanie lub porównywanie. Struktury to ważna część języka C#, którą warto poznać, ponieważ mogą one być bardzo przydatne w wielu zastosowaniach.

Dzięki swojej charakterystyce jako typów wartościowych, struktury są zazwyczaj szybsze niż klasy, co oznacza, że mogą być bardziej wydajne w niektórych zastosowaniach. Jednak warto pamiętać, że przekazywanie struktur jako argumentów do metod lub zwracanie ich z metod może prowadzić do kopii wartości, co może wpłynąć na wydajność w przypadku dużych struktur.

Dlatego należy wykorzystywać struktury wtedy, gdy potrzebujemy przechować małe zestawy danych lub gdy wydajność jest kluczowa. W przypadku większych zestawów danych, zwykle lepiej jest wykorzystać klasy, ponieważ klasy są typami referencyjnymi i mogą być bardziej wydajne w przypadku przekazywania ich jako argumentów do metod lub zwracania ich z metod.

6.3.1 Sprawdź się!

1. Zarządzaniem pamięcią w języku C# zajmuje się:
   a) Programista
   b) W języku C# pamięć zarządzana jest automatycznie, więc nie trzeba się tym martwić.
   c) Garbage Collector
   d) Specjalne klasy

2. Do przechowywania adresu pamięci komputera służy:
   a) Funkcja **`Console.WriteLine()`**
   b) Wskaźnik
   c) Zwykła zmienna
   d) Obiekt klasy **`MemoryAccess`**

3. Kiedy należy użyć bloku **`unsafe`**?
   a) Jest zawsze wymagany przy pracy z wskaźnikami w języku C#.
   b) Powinien być używany, gdy potrzebujemy szybkiego dostępu do danych w pamięci, ale nie jest to konieczne do pracy z wskaźnikami.
   c) Jest wymagany, gdy potrzebujemy odwoływać się do kodu napisanego w języku assemblera.
   d) Blok "**`unsafe`**" nie jest rekomendowany do użytku, a wskaźniki powinny być zawsze unikane w języku C#.

4. Wskaż różnicę między **`int*`** a **`IntPtr`**:
   a) **`int*`** to wskaźnik na typ **`int`**, podczas gdy **`IntPtr`** to typ wartości, który może przechowywać wskaźniki na różne typy danych.
   b) **`int*`** to typ wartości, który może przechowywać wskaźniki na różne typy danych, podczas gdy **`IntPtr`** to wskaźnik na typ **`int.`**
   c) **`int*`** i **`IntPtr`** to dwa różne sposoby deklarowania wskaźników

d) **`int*`** to wskaźnik na typ **`int`**, który jest używany w języku C++, podczas gdy **`IntPtr`** jest używany w języku C#.

5. Wskaż różnicę między klasą a strukturą:
   a) Struktura to typ wartości, podczas gdy klasa to typ referencyjny.
   b) Struktura i klasa to dwa różne sposoby deklarowania typów w języku C#.
   c) Struktura jest mniej wydajna niż klasa, ponieważ zajmuje więcej miejsca w pamięci.
   d) Struktura może dziedziczyć po innej strukturze lub klasie, podczas gdy klasa może dziedziczyć tylko po innej klasie.

6. Czy struktury obsługują mechanizm dziedziczenia?
   a) Tak
   b) Nie

6.3.2 Zadania praktyczne:

1. Napisz strukturę Point, która będzie przechowywać dwie współrzędne (x i y) jako liczby całkowite. Dodaj do struktury metodę, która obliczy odległość między dwoma punktami.
2. Napisz strukturę **`Rectangle`**, która będzie przechowywać pozycję lewego górnego rogu (x i y) jako liczby całkowite oraz długość i szerokość jako liczby rzeczywiste. Dodaj do struktury metodę, która obliczy pole powierzchni prostokąta.
3. Napisz strukturę **`Date`**, która będzie przechowywać datę jako trzy liczby całkowite (dzień, miesiąc i rok). Dodaj do struktury metodę, która sprawdzi, czy podana data jest poprawna.