Typy, operátory a proměnné

Typ v programování určuje, jaký druh hodnoty proměnná může obsahovat. Jinými slovy: typ definuje, jaká data jsou uložena v proměnné a jaké operace s nimi lze provádět.

Například typ může určit, že proměnná bude obsahovat celé číslo (např. 42), textový řetězec (např. "Ahoj světe"), nebo třeba desetinné číslo (např. 3.14).

Deklarace proměnné

Deklarace proměnné znamená, že vytvoříme proměnnou a přiřadíme jí nějaký typ. V C# se to dělá tak, že nejdříve napíšeme typ a potom název proměnné. Často se také rovnou uvede nějaká první hodnota.

```
int pocetDraku = 3;
string jmenoHrdiny = "Aragorn";
double silaMagie = 7.5;
```

- `int` znamená, že `pocetDraku` bude obsahovat celé číslo.
- `string` znamená, že `jmenoHrdiny` bude obsahovat text.
- `double` znamená, že `silaMagie` bude obsahovat desetinné číslo.

Jakmile proměnnou deklarujeme, můžeme ji dále používat v kódu a manipulovat s její hodnotou podle jejího typu. Pro přiřazení hodnoty do proměnné se používá operátor `=`.

```
int pocetDraku = 3;
pocetDraku = 5;
pocetDraku = 2;
pocetDraku = ( 1 + 7 ) / 2; // tj. 4
```

Primitivní typy

Primitivní typy jsou ty nejjednodušší a nejzákladnější typy, které jazyk C# nabízí. Jsou to typy, které nejsou složené z jiných typů a představují základní stavební kameny pro další typy a operace.

Zde jsou některé z nejčastěji používaných primitivních typů v C#:

Typ	Popis
`int`	Celé číslo. Např. 1, -42, 1000.
`double`	Desetinné číslo s dvojitou přesností. Např. 3.14, -0.001, 2.71828.
`char`	Jednotlivý znak. Např. 'A', 'z', '7'.
`bool`	Logická hodnota, která může být `true` (pravda) nebo `false` (nepravda).

NOTE	`string`, nebo-li textový řetězec, není tak docela primitivní typ, protože to je ve skutečnosti zamaskovaný char[] (pole znaků).

Operátory

Operátory jsou speciální symboly, které provádějí operace na jedné nebo více hodnotách. Pomocí operátorů můžeme provádět různé druhy výpočtů nebo logických operací.

Například:

- Sčítání (`+`)
- Odčítání (`-`)
- Porovnání hodnot (`==`, `>`, `<`, `>=`, `<=`)
- Logické operace (`&&`, `||`)

Výraz

Výraz je kombinace hodnot (konstant nebo proměnných) a operátorů, která se vyhodnotí na nějakou hodnotu. V praxi to znamená, že výraz provede nějakou operaci (např. sčítání) a vrátí výsledek.

Například: `2 + 3` je výraz, který používá operátor `+` k sečtení čísel `2` a `3`, a výsledkem je `5`.

Příklady různých výrazů

1. Výraz: `5 + 3` výsledek: `8`
2. Výraz: `10 - 2` výsledek: `8`
3. Výraz: `4 * 7` výsledek: `28`
4. Výraz: `20 / 5` výsledek: `4`
5. Výraz: `7 % 3` (operátor modulo, tj. zbytek po dělení celých čísel) výsledek: `1`
6. Výraz: `2 + 3 * 4` výsledek: `14` (násobení má vyšší prioritu než sčítání, takže se nejprve násobí 3 * 4 a potom přičte 2)
7. Výraz: `10 > 3` výsledek: `true` (10 je větší než 3)
8. Výraz: `5 == 5` výsledek: `true` (5 se rovná 5)
9. Výraz: `!(2 < 1)` (operátor `!` znamená negaci - převrácení hodnoty) výsledek: `true` (2 není menší než 1, takže výraz uvnitř závorek je `false`, negací se změní na `true`)
10. Výraz: `(3 + 2) * 4` výsledek: `20` (závorky mají vyšší prioritu, takže se nejprve sečte 3 + 2 a poté vynásobí 4)

Bool výrazy

Bool výrazy (logické výrazy) jsou výrazy, které se vyhodnocují na logickou hodnotu `true` (pravda) nebo `false` (nepravda). V C# používáme pro tyto výrazy typ `bool`.

Tyto výrazy často používají logické operátory a porovnávací operátory. Bool výrazy jsou klíčové v podmínkách, jako jsou if-else bloky, cykly a další logické rozhodování v kódu.

Operátory používané v bool výrazech

1. **Porovnávací operátory**:
   - `==` : Rovná se (např. `5 == 5` je `true`)
   - `!=` : Nerovná se (např. `5 != 3` je `true`)
   - `>` : Větší než (např. `10 > 2` je `true`)
   - `<` : Menší než (např. `2 < 10` je `true`)
   - `>=` : Větší než nebo rovno (např. `5 >= 5` je `true`)
   - `<=` : Menší než nebo rovno (např. `3 <= 4` je `true`)
2. **Logické operátory**:
   - `&&` : Logické AND - vrátí `true`, pokud jsou oba výrazy pravdivé (např. `true && true` je `true`)
   - `||` : Logické OR - vrátí `true`, pokud je alespoň jeden výraz pravdivý (např. `true || false` je `true`)
   - `!` : Logické NOT - neguje hodnotu výrazu (např. `!true` je `false`)

Příklady bool výrazů

1. Výraz: `5 > 3` výsledek: `true` (5 je větší než 3)
2. Výraz: `4 == 4` výsledek: `true` (4 se rovná 4)
3. Výraz: `6 != 7` výsledek: `true` (6 není rovno 7)
4. Výraz: `!(2 > 3)` výsledek: `true` (2 není větší než 3, takže uvnitř závorky je `false`, a negace `!` změní na `true`)
5. Výraz: `true && false` výsledek: `false` (oba výrazy musí být `true`, aby byl výsledek `true`)
6. Výraz: `true || false` výsledek: `true` (stačí, že jeden z výrazů je `true`)

Bool výrazy jsou důležité pro rozhodování v programu, protože nám umožňují kontrolovat, zda určité podmínky platí, a podle toho řídit tok programu.

Ve výrazech (nejen bool, ale i ve výpočtech apod.) můžeme samozřejmě pracovat i s proměnnými, nejen s pevnými hodnotami.

```
int vekHrdiny = 25;
bool jeKouzloAktivni = false;
string jmenoHrdiny = "Aragorn";
```

Bool výrazy s proměnnými:

```
bool jeDospely = vekHrdiny >= 18;
```

- Výraz: `vekHrdiny >= 18` výsledek: `true` (25 je větší nebo rovno 18, takže `jeDospely` bude `true`)

```
bool muzePouzitKouzlo = jeKouzloAktivni == true;
```

- Výraz: `jeKouzloAktivni == true` výsledek: `false` (hodnota `jeKouzloAktivni` je `false`, takže `muzePouzitKouzlo` bude `false`)

Je to ovšem totéž, jako kdybychom napsali:

```
bool muzePouzitKouzlo = jeKouzloAktivni;
```

Můžeme pracovat se string proměnnou:

```
bool jeSpravneJmeno = jmenoHrdiny == "Aragorn";
```

- Výraz: `jmenoHrdiny == "Aragorn"` výsledek: `true` (hodnota proměnné `jmenoHrdiny` je `"Aragorn"`, takže `jeSpravneJmeno` bude `true`)

```
bool jeDospelyAKouzloAktivni = jeDospely && jeKouzloAktivni;
```

- Výraz: `jeDospely && jeKouzloAktivni` výsledek: `false` (výraz `jeDospely` je `true`, ale `jeKouzloAktivni` je `false`, takže celý výraz bude `false`)

```
bool maSpravneJmenoNeboJeDospely = jeSpravneJmeno || jeDospely;
```

- Výraz: `jeSpravneJmeno || jeDospely` výsledek: `true` (alespoň jeden z výrazů `jeSpravneJmeno` nebo `jeDospely` je `true`, takže celý výraz bude `true`)

Když pracujeme s proměnnými v bool výrazech, hodnota výrazu závisí na hodnotách těchto proměnných. Kombinací proměnných a operátorů můžeme vytvářet složitější logické podmínky a rozhodovat se podle nich, co v programu dělat dál.

Někdy může být bool výraz velmi složitý. Vyplatí se ho rozebrat na menší kousky, ty si uložit do proměnných a z nich pak složit výsledek.

Funkce

Funkce (někdy se jim říká metody) jsou základní stavební kameny programování. Umožňují nám zabalit určitý kus kódu, který plní konkrétní úkol, do jednoho bloku. Tento blok pak můžeme opakovaně používat v programu, což zjednodušuje a zpřehledňuje náš kód.

Definice funkce

Definice funkce je místo v kódu, kde specifikujeme, co funkce dělá (resp. co BUDE dělat, až ji někdo

zavolá). Obsahuje:

1. Název funkce (pomocí kterého ji budeme volat).
2. Parametry (volitelně) - hodnoty, které můžeme funkci předat
3. Návratový typ (jaký druh hodnoty funkce vrací).
4. Tělo funkce - blok kódu, který se provede, když funkci zavoláme.

Příklad jednoduché funkce:

```
int SectiCisla(int a, int b)
{
    int vysledek = a + b;
    return vysledek;
}
```

- `int` před názvem funkce `SectiCisla` je návratový typ, což znamená, že funkce vrací celé číslo (`int`).
- `int a, int b` jsou parametry funkce, což jsou hodnoty, které funkci předáme.
- `return vysledek;` je příkaz, který ukončí funkci a vrátí hodnotu `vysledek` zpět tam, odkud byla funkce zavolána.

Rozdíl mezi definicí a zavoláním funkce

- **Definice funkce** je to, co jsme právě napsali výše - specifikujeme, co funkce udělá až ji zavoláme, jaké má parametry a co vrací.
- **Zavolání funkce** je to, když někde v programu tuto funkci použijeme, abychom provedli její kód a získali její výsledek.

Příklad zavolání funkce:

```
int soucet = SectiCisla(5, 3); // Zavolání funkce s parametry 5 a 3
```

V tomto případě se funkce `SectiCisla` provede s hodnotami `5` a `3`, a výsledkem bude číslo `8`, které se uloží do proměnné `soucet`.

Co znamená `return`?

Příkaz `return` v těle funkce:

1. Ukončí provádění funkce.
2. Vrátí hodnotu uvedenou za `return` zpět do místa, odkud byla funkce zavolána.

Příklad:

```
int Nasob(int a, int b)
```

```
{
    return a * b;
}
```

Když zavoláme funkci `Nasob(4, 5)`, funkce okamžitě vrátí hodnotu `20` (4 * 5) a ukončí se.

`return` na více místech ve funkci

Pokud je ve funkci více příkazů `return`, funkce se ukončí při prvním z nich, který se vykoná. To znamená, že se zbytek funkce už neprovede. To platí i v případě, kdy je return uveden uvnitř cyklu (for, while, ...).

Příklad:

```
int ZkontrolujCislo(int cislo)
{
    if (cislo < 0)
    {
        // Pokud je číslo záporné, vrátí -1 a funkce skončí
        return -1;
    }

    if (cislo == 0)
    {
        // Pokud je číslo 0, vrátí 0 a funkce skončí
        return 0;
    }
    // Pokud je číslo kladné, vrátí 1 a funkce skončí
    return 1;
}
```

V tomto příkladu se funkce ukončí po prvním `return`, který se provede. Například, když zavoláme `ZkontrolujCislo(-5)`, funkce vrátí `-1` a zbytek kódu se už neprovede. Neexistuje způsob, jak by se mohl return zavolat vícekrát.

Užití funkce ve výrazu

Funkci můžeme využít jako součást výrazu stejně jako běžné operátory nebo proměnné. Když funkci zavoláme ve výrazu, její výsledek se stane hodnotou, kterou můžeme dále použít.

Představme si funkci `Nasob`, kterou jsme definovali dříve:

```
int Nasob(int a, int b)
{
    return a * b;
}
```

Tuto funkci můžeme použít ve výrazech různými způsoby:

Použití funkce přímo při přiřazení hodnoty do proměnné

```
int vysledek = Nasob(4, 5);
```

- Výraz: `Nasob(4, 5)` výsledek: `20`
- Hodnota `20` se přiřadí do proměnné `vysledek`.

Použití funkce jako součást složitějšího výrazu

```
int soucet = Nasob(2, 3) + Nasob(4, 5);
```

- Výraz: `Nasob(2, 3) + Nasob(4, 5)` výsledek: `6 + 20 = 26`
- Hodnota `26` se přiřadí do proměnné `soucet`.

Použití funkce v podmínce

```
if (Nasob(3, 3) > 8)
{
    Console.WriteLine("Výsledek je větší než 8.");
}
```

- Výraz: `Nasob(3, 3) > 8` výsledek: `9 > 8`, což je `true`, takže se vykoná `Console.WriteLine`.

Použití funkce jako argument jiné funkce

Funkci `Nasob` můžeme použít jako argument pro další volání funkce (třeba i úplně jiné).

```
int vysledek = Nasob(Nasob(2, 3), 4);
```

- První volání: `Nasob(2, 3)` výsledek: `6` (2 * 3)
- Druhé volání: `Nasob(6, 4)` výsledek: `24` (6 * 4)
- Výsledná hodnota proměnné `vysledek` je `24`.

Funkce jsou tedy nejen způsobem, jak zabalit a znovu použít kód, ale také nástrojem pro stavbu složitějších výpočtů a logických kontrol v programu.

Pole

Pole (v angličtině "array") je datová struktura, která umožňuje uložit více hodnot stejného typu pod jedním názvem. Každá z těchto hodnot (tzv. prvek pole) je dostupná prostřednictvím indexu (čísla), který určuje pozici prvku v poli.

Pole je užitečné, když potřebujeme pracovat s více hodnotami stejného typu a chceme je mít pohromadě v jednom kontejneru. Například můžeme mít pole čísel, textů nebo objektů.

Deklarace pole

Pro deklaraci pole musíme specifikovat typ prvků, které pole bude obsahovat, a počet prvků.

```
int[] cisla = new int[5];
```

- `int[]` znamená, že pole `cisla` bude obsahovat prvky typu `int` (celá čísla).
- `new int[5]` vytvoří pole s 5 prvky, které jsou inicializovány na výchozí hodnoty (pro `int` je to `0`, ve všech prvcích bude `0`).

Inicializace pole

Pole můžeme rovnou naplnit hodnotami při jeho deklaraci:

```
int[] cisla = { 1, 2, 3, 4, 5 };
```

- Pole `cisla` obsahuje pět čísel: 1, 2, 3, 4, 5.

Přístup k prvkům pole

K jednotlivým prvkům pole přistupujeme pomocí jejich indexu. Indexy začínají od nuly, což znamená, že první prvek má index `0`, druhý prvek index `1`, a tak dále.

Příklad:

```
int prvniCislo = cisla[0]; // Získá první prvek pole, což je 1
cisla[2] = 10;             // Změní třetí prvek pole na hodnotu 10
```

- `cisla[0]` vrací hodnotu prvního prvku (v tomto případě `1`).
- `cisla[2] = 10` změní třetí prvek (který měl hodnotu `3`) na `10`.

Délka pole

Délku pole (tedy počet prvků, které pole obsahuje) zjistíme pomocí vlastnosti `Length`.

Příklad:

```
int delkaPole = cisla.Length; // Vrátí hodnotu 5, protože pole má pět prvků
```

Jako všechno - i pole můžeme použít při stavbě výrazu:

```
bool faktDlouhyPole = cisla.Length > 100; // faktDlouhyPole bude `false`,
                                          // protože prvků je v `cisla` jen 5.
```