PŘÍKAZY

Hlavní rozdíly v příkazech mezi C# a C++ jsou následující:

- V C# není příkaz `asm`.
- Pro příkazy `switch` a `goto` platí trochu odlišná pravidla.
- Pro výrazový příkaz platí silnější pravidla v C#.
- V C# jsou navíc příkazy `foreach`, `checked`, `unchecked`, `lock`, `using` a `yield`.
- V příkazech cyklu a v příkazu `if` nelze jako podmínku použít číslo nebo referenci.

Základní konstrukce

Výrazový příkaz

V C#, stejně jako v C++, platí, že připojením středníku na konec výrazu, se z výrazu stane výrazový příkaz.

Syntaxe:

výrazový_příkaz:

výraz **;**

Na rozdíl od C++ ovšem nelze použít jakýkoli výraz, ale pouze výraz, který má vedlejší efekt. To znamená, že např.

```
A+B;
5;
```

nejsou správné výrazové příkazy v C#. Mezi povolené výrazové příkazy patří volání metody, použití operátorů ++ a --, přiřazení atd.

Blok

V místě, kde syntaktická pravidla v C# povolují příkaz, lze použít jeden příkaz nebo *blok* neboli *složený příkaz*.

Syntaxe:

blok:

{ $příkazy_{nep}$ **}**

Příkazy – libovolné příkazy, včetně deklarací a vnořených bloků.

Na rozdíl od C++ nelze v C# ve vnořeném bloku deklarovat stejný identifikátor jako ve vnějším bloku. Následující úsek programu je v C# chybný:

```
{ // Začátek vnějšího bloku
   int i = 5;
   { // Začátek vnitřního bloku
      int i = 6; // Chyba
   }
}
```

Deklarace

Stejně jako v C++ se i v C# považuje deklarace lokální proměnné za plnohodnotný příkaz. Z toho vyplývá, že se deklarace lokální proměnné může objevit kdekoli mezi ostatními příkazy.

Rozhodování

Jazyk C# obsahuje stejně jako C++ dva rozhodovací příkazy (angl. *selection statements*): `if` a `switch`.

Příkaz if

Syntaxe:

příkaz_if:

if (*podmínka* **)** *příkaz_1*

if (*podmínka* **)** *příkaz_1* **else** *příkaz_2*

Podmínka – výraz typu `bool` nebo typu, který lze implicitně zkonvertovat na typ `bool` nebo typu, který má `operator true()`. Na rozdíl od C++ nelze použít čísla nebo reference a nelze zde deklarovat proměnnou.

Způsob použití příkazu `if` je stejný jako v C++.

Příkaz switch

Zjednodušená syntaxe:

příkaz switch:

switch (*výraz* **) {** *alternativy* **}**

alternativy:

alternativa

alternativa alternativy

alternativa:

návěští příkazy

návěští:

case *konstanta* **:**

case *konstanta* **:** *návěští*

default :

Alternativa – skupina příkazů, před níž je připojeno jedno nebo několik návěští.

Odlišnosti v příkazu `switch` v C# oproti C++:

- Jako konstanty v návěštích `case` lze použít i znakové řetězce a řídící *výraz* může být typu `string`.
- Program nesmí „samovolně propadnout“ z jedné *alternativy* do následující *alternativy*. Každá z *alternativ* musí končit některým z příkazů způsobujících přenos řízení. Zpravidla se jedná o příkaz `break`, ale může to být i příkaz `goto`, `return` nebo `throw`.
- Program může přejít na začátek jiné *alternativy* pomocí příkazu `goto` s uvedením *návěští*.

Příklad

```
class Program
{
   static void Main(string[] args)
   {
      string s;
      while (true) {
         Console.WriteLine("Zadej text");
         s = Console.ReadLine();
         switch (s) {
            case "while":
            case "switch":
               Console.WriteLine("Zadal jsi klíčové slovo");
               break;
            case "konec":
               return;
            case "if":
               goto case "while";
            case "":
               Console.WriteLine("Prázdný řádek");
               goto default;
            default:
               Console.WriteLine("Zadal jsi neznámý text.");
               break;
         }
      }
   }
}
```

Cykly

Jazyk C# obsahuje 4 příkazy cyklu (angl. *iteration statements*): `for`, `while`, `do-while` a `foreach`.

V příkazu cyklu lze stejně jako v C++ používat příkazy `break` a `continue`.

Příkaz while

Syntaxe:

příkaz_while:

while (*podmínka* **)** *příkaz*

Odlišnosti v C# oproti C++ se týkají *podmínky* a jsou stejné jako u příkazu `if`.

Příkaz for

Syntaxe:

příkaz_for:

for ($inicializace_{nep}$ **;** $podmínka_{nep}$ **;** $krok_{nep}$ **)** *příkaz*

V části *inicializace* lze podobně jako v C++ deklarovat proměnné. Lze zde také uvést několik příkazů oddělených čárkou.

Podmínka má stejná omezení jako u příkazu `if`.

Krok může být jeden nebo skupina příkazů oddělených čárkou.

Příkaz do-while

Syntaxe:

příkaz_do-while:

do *příkaz* **while (** *podmínka* **) ;**

Podmínka má stejná omezení jako u příkazu `if`.

Příkaz foreach

Příkaz slouží k iterování všech prvků v kolekci (kontejneru).

Syntaxe:

příkaz_foreach:

foreach (*typ identifikátor* **in** *výraz* **)** *příkaz*

Výraz – musí určovat nějakou kolekci.

Typ identifikátor – deklarace proměnné *identifikátor* typu *typ*. Do této proměnné se budou ukládat jednotlivé hodnoty prvků kolekce. Jedná se o *řídící proměnnou* cyklu.

Jako kolekci lze použít jakýkoli datový typ, který implementuje rozhraní `IEnumerable` nebo jeho generickou obdobu, tj. který obsahuje veřejnou metodu `GetEnumerator()`, jež vrací hodnotu `E`, kterou lze použít jako *enumerátor*. Dále jako kolekci lze použít metodu nebo vlastnost, která obsahuje *iterátor*. Iterátory byly zavedeny od verze .NET 2.0 a budou vysvětleny později.

`E` musí implementovat rozhraní `IEnumerator` nebo jeho generickou obdobu, tj. musí obsahovat tyto složky:

- Veřejná metoda `bool MoveNext()` – přejde na další prvek kolekce. Vrací `false`, pokud již v kolekci neexistuje další prvek.
- Veřejná metoda `Reset()` – nastaví enumerátor na počáteční pozici, což je místo před prvním prvkem kolekce.
- Veřejná vlastnost `Current` – poskytuje aktuální prvek kolekce.

Enumerátor může být jak referenčního, tak i hodnotového typu.

Cyklus `foreach` probíhá následovně:

- Vyhodnotí se *výraz* a tím se získá kolekce `K`. Má-li výraz hodnotu `null`, vznikne výjimka typu `System.NullReferenceException`.
- Pomocí metody `K.GetEnumerator()` se získá enumerátor `E`. Je-li `E` referenčního typu a má-li hodnotu `null`, vznikne výjimka typu `System.NullReferenceException`.
- Zavolá se metoda `E.MoveNext()`. Vrátí-li `true`, získá se aktuální hodnota v kolekci pomocí vlastnosti `E.Current`, převede se pomocí explicitní konverze na typ řídící proměnné a přiřadí se této proměnné. Pak se provede tělo cyklu *příkaz* a znovu se zavolá metoda `E.MoveNext()`.
- Provádění cyklu `foreach` skončí, když `E.MoveNext()` vrátí `false`.

V těle cyklu lze řídící proměnnou použít jen k získání hodnoty. Nelze do ní přiřadit jinou hodnotu.

Rozhraní `IEnumerable` implementují nejen kolekce, deklarované v prostoru jmen `System.Collections`, ale i pole včetně vícerozměrných.

Následující metoda vypíše prvky pole na obrazovku:

```
static void VypisPole(int[] pole)
{
   foreach (int a in pole) {
      Console.Write(a + " ");
   }
   Console.WriteLine();
}
```

Skoky

Skoky jsou příkazy, jež způsobují přenos řízení (angl. *jump statements*). V jazyce C# jsou to příkazy `break`, `continue`, `goto`, `return` a `throw`.

Příkaz `break`, `continue` a `return` má stejnou syntaxi a význam jako v C++.

Příkaz goto

Syntaxe:

příkaz_goto:

goto *identifikátor* **;**

goto default ;

goto case *konstanta* **;**

První možnost má stejný význam jako v C++. Další dvě možnosti se mohou objevit pouze v těle příkazu `switch` – viz kapitola „Příkaz switch“.

Příkaz throw

Slouží k vyvolání výjimky.

Syntaxe:

příkaz_throw:

throw $výraz_{nep}$ **;**

Výraz – odkaz na instanci třídy odvozené od třídy `System.Exception`. Pokud výsledkem výrazu je hodnota `null`, vznikne výjimka typu `System.NullReferenceException`.

Význam příkazu `throw` je stejný jako v C++.

Příkaz using

Příkaz `using` v jazyce C++ neexistuje. Slouží k řízení doby života některých objektů, které zpravidla spravují nějaké *prostředky* (angl. *resources*), jako jsou např. soubory.

Syntaxe:

příkaz_using:

using (*získání_prostředků* **)** *příkaz*

získání_prostředků:

deklarace_lokální_proměnné

výraz

Prostředek – instance třídy nebo struktury, která implementuje rozhraní `System.IDisposable`, které obsahuje pouze metodu `void Dispose()`.

Deklarace lokální proměnné – deklarace proměnné s inicializací, která představuje *prostředek*.

Výraz – výraz, jehož výsledkem je *prostředek*.

Příkaz – příkaz, v němž se nesmí přiradit jiná hodnota do proměnné deklarované v části *získání prostředků*.

Příkaz `using` ve formě

using (*výraz* **)** *příkaz*

je ekvivalentní k formě

using (R r = *výraz* **)** *příkaz*

kde `r` je proměnná typu *prostředku* `R`.

Je-li třída `R` třída prostředků, je příkaz

```
using (R r = new R()) {
   r.f();
}
```

zcela ekvivalentní konstrukci

```
R r = new R();
try {
   r.f();
}
finally {
   if (r != null) ((IDisposable)r).Dispose();
}
```

V příkazu `using` lze deklarovat i několik proměnných stejného typu oddělených čárkou. Např. příkaz

```
using (R r1 = new R(), r2 = new R()) {
   r1.f();
   r2.f();
}
```

je ekvivalentem příkazů

```
using (R r1 = new R()) {
   using (R r2 = new R()) {
      r1.f();
      r2.f();
   }
}
```

Příkaz lock

Příkaz `lock` nemá v C++ obdobu. Slouží k synchronizaci mezi *podprocesy* (angl. *threads*) programu pomocí vzájemně výlučných zámků, tzv. mutexů.

Syntaxe:

příkaz_lock:

lock (*výraz* **)** *příkaz*

Výraz – odkaz na referenční typ (v této souvislosti se nepoužije zabalení hodnotových typů).

Příkaz `lock` získá pro daný objekt mutex, provede *příkaz* a mutex potom uvolní.

Příkaz

```
lock (x) {
   /* ... */
}
```

má stejný význam jako

```
object obj = x;
System.Threading.Monitor.Enter(obj);
try {
  /* ... */
}
finally {
   System.Threading.Monitor.Exit(obj);
}
```

Přiřazení výrazu `x` do proměnné `obj` je uvedeno proto, aby se výraz `x` vyhodnotil pouze jednou.

Pro statické metody se často jako „uzamykací objekt“ používá instance třídy `Type`, získaná pomocí operátoru `typeof`. Pokud je např. potřebné vyloučit současné volání statických metod `Přidej` a `Odeber` třídy `SynchronizovanýObjekt` ze dvou různých podprocesů, lze napsat následující kód:

```
class SynchronizovanýObjekt
{
   public static void Přidej(object x)
   {
      lock(typeof(SynchronizovanýObjekt)) { /* tělo metody */ }
   }
   public static void Odeber(object x)
   {
      lock(typeof(SynchronizovanýObjekt)) { /* tělo metody */ }
   }
}
```

Příkaz yield

Příkaz `yield` v C++ neexistuje. Je používán v iterátoru – viz dále.

Příkaz try

Příkaz `try` je podobně jako v C++ pokusný blok `try` používán pro zachycení a ošetření výjimek – viz kapitola „Výjimky“.

OPERÁTORY A VÝRAZY

Přehled operátorů

Obsahuje-li výraz několik operátorů, operátory se vyhodnocují v pořadí daném prioritou operátorů. Operátory s vyšší prioritou se vyhodnocují dříve. Jestliže výraz obsahuje více operátorů se stejnou prioritou, určuje se pořadí vyhodnocení operátorů podle jejich asociativity. Operátory mohou být asociativní zleva doprava nebo naopak.

V jazyce C# jsou binární operátory asociativní zleva doprava, zatímco unární operátory, přiřazovací operátory, ternární operátor `?:` a operátor nulového sjednocení `??` jsou asociativní zprava doleva.

Pořadí vyhodnocování operátorů ve výrazu lze změnit pomocí kulatých závorek, které mají nejvyšší prioritu.

Seznam operátorů je uveden v následující tabulce. Vyšší priorita má v tabulce nižší hodnotu. Sloupec P reprezentuje prioritu, A asociativitu. Je-li ve sloupci U křížek, daný operátor se smí použít pouze v nezabezpečeném kódu.

Operátor	P	A	U	Význam
Primární operátory:				
`.` (tečka)	1	→		Přístup ke složce instance: `a.b`
`[ ]`	1	→		Indexování (přístup k prvku pole): `a[b]`
`(`*výraz*`)`	1	→		Uzávorkování výrazu: `(a + b) * c`
`( )`	1	→		Volání metody: `f(a)`
`++`	1	→		Postfixová inkrementace: `a++`
`--`	1	→		Postfixová dekrementace: `a--`
`new`	1	→		Volání konstruktoru
`stackalloc`	1	→	x	Alokace pole na zásobníku
`typeof`	1	→		Určení typu: `typeof(a)`
`sizeof`	1	→	x	Určení velikosti hodnotového typu: `sizeof(int)`
`checked`	1	→		Zapnutí kontroly přetečení: `checked(a + b)`
`unchecked`	1	→		Vypnutí kontroly přetečení: `unchecked(a + b)`
`->`	1	→	x	Přístup ke složce struktury prostřednictvím ukazatele: `a->b`
Unární operátory:				
`+`	2	←		Unární plus: `+a`
`-`	2	←		Unární minus: `-a`
`!`	2	←		Negace logické hodnoty: `!a`
`~`	2	←		Bitový doplněk: `~a`
`++`	2	←		Prefixová inkrementace: `++a`
`--`	2	←		Postfixová inkrementace: `--a`
`(`*typ*`)`	2	←		Přetopování (konverzní operátor): `(int)a`
`&`	2	←	x	Získání adresy: `&a`
`*`	2	←	x	Dereferencování ukazatele: `*a`

Operátor	P	A	U	Význam
Multiplikativní operátory:				
*	3	→		Násobení: `a * b`
/	3	→		Dělení: `a / b`
%	3	→		Modulo (zbytek po dělení): `a % b`
Aditivní operátory:				
+	4	→		Sčítání: `a + b`
-	4	→		Odečítání: `a - b`
Operátory posunu:				
<<	5	→		Bitový posun doleva: `a << b`
>>	5	→		Bitový posun doprava: `a >> b`
Relační operátory a operátory testování typu:				
<, >, <=, >=	6	→		Menší, větší, menší nebo rovno, větší nebo rovno: `a < b`
is	6	→		Test příslušnosti k typu: `a is string`
as	6	→		Přetypování: `a as string`
Operátory rovnosti:				
==	7	→		Rovná se: `a == b`
!=	7	→		Nerovná se: `a != b`
Operátor úplného logického AND:				
&	8	→		Logická konjunkce s úplným vyhodnocením nebo logická konjunkce po bitech: `a & b`
Operátor logického XOR:				
^	9	→		Logická nonekvivalence nebo logická nonekvivalence po bitech: `a ^ b`
Operátor úplného logického OR:				
|	10	→		Logická disjunkce s úplným vyhodnocením nebo logická disjunkce po bitech: `a | b`
Operátor neúplného logického AND:				
&&	11	→		Logická konjunkce s neúplným vyhodnocením: `a && b`
Operátor neúplného logického OR:				
||	12	→		Logická disjunkce s neúplným vyhodnocením: `a || b`
Operátor nulového sjednocení:				
??	13	←		Nulové sjednocení: `a ?? b`
Podmíněný operátor:				
?:	14	←		Podmíněný operátor: `a ? b : c`
Přiřazovací operátory:				
=	15	←		Přiřazení: `a = b`
*= /= %= += -= <<= >>= &= ^= |=	15	←		Složená přiřazení: `a += b`

Poznámky:

- V C# se za operátor považují i kulaté závorky použité ve výrazu za účelem změny pořadí vyhodnocování, zatímco v C++ se považují za oddělovač. Tento syntaktický rozdíl nemá pro programátora žádný praktický význam.
- Operátor `new` je v C# operátorem volání konstruktoru, v C++ je alokátorem. U struktur se v C# totiž alokace neprovádí.
- V C# chybí operátory přetypování `static_cast`, `dynamic_cast`, `reinterpret_cast` a `const_cast`. Všechna přetypování se provádí pomocí operátoru (*typ*) nebo `as` (viz dále).
- V C# dále chybí operátory `.*`, `->*` (operátory pro třídní ukazatel), `typeid`.
- Některé operátory C# lze využít pouze v nezabezpečeném kódu: `stackalloc` (alokace na zásobníku), `sizeof` (velikost hodnotového typu), `->` (přístup ke složce struktury), `&` (získání adresy), `*` (dereferencování ukazatele).

Číselná rozšíření

Ve výrazech je často potřebné kombinovat různé typy operandů. Pro číselné typy se přitom uplatňují automatické konverze, označované jako *unární* a *binární rozšíření* (angl. *unary and binary numeric promotions*). Tato pravidla se liší od analogických pravidel v C++.

Unární rozšíření

Unární rozšíření jsou implicitní konverze, které se uplatňují pro operandy předdefinovaných (nikoli uživatelem přetížených) unárních operátorů `+`, `-` a `~` (bitový doplněk). Operandy typu `sbyte`, `byte`, `short`, `ushort` a `char` se při nich převedou na typ `int`. V případě unárního operátoru `-` (minus) se navíc operand typu `uint` převede na typ `long`.

To znamená, že následující přiřazení je chybné:

```
sbyte a = -1;
a = -a; // Chyba – nelze implicitně konvertovat int na sbyte, v C++ OK
```

Binární rozšíření

Binární rozšíření se týkají operandů, na než se použijí předdefinované binární operátory `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `==`, `!=`, `<`, `>`, `<=`, `>=`. Binární rozšíření převedou oba operandy na společný typ a tento typ bude v případě nerelačních operátorů také typem výsledku a v případě relačních operátorů je výsledek typu `bool`.

Na binární rozšíření se uplatňují pravidla v následujícím pořadí:

1. Je-li jeden z operandů typu `decimal`, převede se druhý také na typ `decimal`. Je-li druhý operand typu `float` nebo `double`, nastane chyba, protože neexistuje implicitní konverze z těchto typů na typ `decimal`.
2. Jinak, je-li jeden z operandů typu `double`, převede se druhý na typ `double`.
3. Jinak, je-li jeden z operandů typu `float`, převede se druhý na typ `float`.
4. Jinak, je-li jeden z operandů typu `ulong`, převede se druhý na typ `ulong`. Je-li druhý operand typu `sbyte`, `short`, `int` nebo `long`, nastane chyba, protože neexistuje implicitní konverze z typů se znaménkem na typ `ulong`.
5. Jinak, je-li jeden z operandů typu `long`, převede se druhý na typ `long`.
6. Jinak, je-li jeden z operandů typu `uint` a druhý typu `sbyte`, `short` nebo `int`, převedou se oba operandy na typ `long`.

7. Jinak, je-li jeden z operandů typu `uint`, převede se druhý na typ `uint`.
8. Jinak se oba operandy převedou na typ `int`.

Některé operátory

Operátor tečka

Specifikuje přístup ke statickým a nestatickým složkám tříd, struktur, ale i prostorů jmen a výčtových typů. Nahrazuje operátory tečka, `->` a `::` z C++.

Zjišťování typu: typeof a is

Operátor `typeof` má podobný význam jako operátor `typeid` z C++. Lze jej ale použít pouze na jméno typu, nikoli na výraz. Vrací instanci třídy `System.Type`, která popisuje daný typ.

Instanci třídy `System.Type`, která popisuje typ výrazu, lze získat pomocí metody `GetType()`, zděděné ze třídy `object`. Lze tedy napsat např.

```
if ((a + b).GetType() == typeof(int)) { /* ... *// }
```

Třída `System.Type` má podstatně širší možnosti než třída `type_info` v C++.

Operátor `is` slouží také pro zjišťování typu. Má následující syntaxi:

použití operátoru is:

výraz **is** *typ*

Výraz – výraz referenčního typu.

Typ – jméno typu třídy, struktury nebo rozhraní.

Operátor `is` vrací `true`, pokud lze *výraz* přetypovat na *typ*.

Pro hodnotové typy lze tento operátor použít jen v případě, kdy *výraz* je typu `object`.

Přetypování

C# má dva operátory přetypování, a to operátor (*typ*) a operátor `as`. Na rozdíl od C++ se v C# přetypování mezi objektovými typy vždy kontroluje, zda má smysl.

Operátor (*typ*) se používá stejně jako v C++. Pokud je už při překladu jasné, že požadované přetypování nelze provést, ohlásí překladač chybu. Jinak se kontrola provede za běhu programu, a pokud přetypování není povoleno, vyvolá se výjimka `System.InvalidCastException`.

Operátor `as` se používá pouze k přetypování mezi referenčními typy nebo při zabalení. Syntaxe jeho použití je následující:

použití operátoru as:

výraz **as** *typ*

Výraz – reference na instanci, která se má přetypovat nebo hodnota, která se má zabalit.

Typ – cílový referenční typ.

Je-li přetypování možné, je výsledkem odkaz na cílový typ, jinak je výsledkem hodnota `null`. Např.:

```
static void Vypis(object r)
{
   string s = r as string;
   if (s != null) Console.WriteLine(s);
}
```

Bitové posuny

Pro levý operand operátorů << a >> se uplatňují podobné konverze jako pro unární operátory:

- Levý operand musí být celočíselného typu. Je-li typu `sbyte`, `byte`, `short`, `ushort` nebo `char`, převede se na typ `int`.
- Pravý operand musí být typu `int` nebo musí být možné ho na typ `int` implicitně zkonvertovat.

Výsledkem je hodnota typu levého operandu.

Rovná se, nerovná se

Operátory `==` a `!=` lze použít i k porovnávání referencí. Implicitně porovnávají, zda se dvě reference odkazují na stejný objekt. To lze ale změnit jejich přetížením. Např. pro třídu `string` jsou tyto operátory přetíženy tak, že porovnávají, zda dvě instance obsahují tytéž řetězce znaků.

Pokud se porovnávají dvě reference, o nichž je v době překladu známo, že odkazují na instance různých typů, ohlásí překladač chybu.

Při porovnání instancí delegátů platí, že dvě instance delegátů jsou si rovny, když jsou stejného typu a:

- oba obsahují `null`, nebo
- oba ukazují na tutéž statickou metodu, nebo
- oba ukazují na tutéž nestatickou metodu a týž objekt, nebo
- oba mohou ukazovat na více metod a obsahují stejné metody (podle pravidel v předchozích dvou odrážkách) a ve stejném pořadí.

Podmíněný operátor

Na rozdíl od C++ smí být první operand podmíněného operátoru `?:` pouze typu `bool` nebo typu, který lze implicitně zkonvertovat na typ `bool`, nebo typu, který obsahuje `operator true`.

Typ výsledku výrazu `A ? X : Y` se určuje takto:

- Je-li typ `X` a typ `Y` stejný, bude tento společný typ i typem výsledku.
- Existuje-li implicitní konverze z typu výrazu `X` na typ výrazu `Y` a neexistuje-li implicitní konverze z typu `Y` na typ `X`, bude typem výsledku typ `Y`.
- Existuje-li implicitní konverze z typu výrazu `Y` na typ výrazu `X` a neexistuje-li implicitní konverze z typu `X` na typ `Y`, bude typem výsledku typ `X`.
- Jinak nelze typ výsledku určit a překladač oznámí chybu.

Při vyhodnocování podmíněného výrazu `A ? X : Y` se nejprve vyhodnotí podmínka `A` a potom operand `X` nebo `Y`, ale ne oba.

PREPROCESOR

Na rozdíl od C++ není součástí překladu programu v C# zpracování zdrojového textu preprocesorem. Přesto jazyk C# obsahuje skupinu direktiv, které mají podobný účel jako direktivy preprocesoru v jazyce C++. Zpracovává je ale překladač. Přesto se o nich v literatuře (včetně normy) hovoří jako o „direktivách preprocesoru“ (angl. pre-processing directives).

Podmíněné symboly

V C# neexistuje možnost definovat makra s parametry nebo bez nich. Lze ale definovat tzv. podmíněné symboly, které lze později použít v direktivách `#if` a v atributu `Conditional` (viz dále) pro podmíněný překlad.

Symbol #define

K definování podmíněného symbolu slouží direktiva `#define`.

Syntaxe:

direktiva_define:

#define *identifikátor*

Tato direktiva musí být v souboru před jakýmikoli příkazy, které nejsou direktivami preprocesoru, tedy i před direktivou `using`.

Direktiva zavádí *identifikátor* podmíněného symbolu, který je definován od místa této direktivy do konce zdrojového souboru. Identifikátor podmíněného symbolu lze použít pouze v direktivách preprocesoru, a proto se může i shodovat s identifikátorem jiného objektu programu (např. proměnné).

Např.

```
#define LADĚNÍ
using System;
// další příkazy programu
```

Definice podmíněného symbolu pro všechny překládané části sestavení: menu **Project | Properties**, část **Build**, pole **Conditional compilation symbols**.

Symbol #undef

Direktiva `#undef` slouží ke zrušení podmíněného symbolu zavedeného direktivou `#define` nebo definovaného ve vlastnostech projektu.

Syntaxe:

direktiva_undef:

#undef *identifikátor*

Identifikátor je identifikátor rušeného podmíněného symbolu. *Identifikátor* není definován od místa výskytu této direktivy do konce zdrojového souboru.

Touto direktivou lze zrušit i symbol, který není definován. V takovém případě se nic nestane.

Direktiva `#undef` musí být v souboru před jakýmikoli příkazy, které nejsou direktivami preprocesoru.

Podmíněný překlad

Pro podmíněný překlad části zdrojového souboru slouží direktiva `#if`. Používá se podobně jako v C++.

Schéma jejího použití:

#if *podmíněný_výraz_1*
    zdrojový_text_1
#elif *podmíněný_výraz_2*
    zdrojový_text_2
`// ...`
#else
    zdrojový_text_n
#endif

Část `#elif` se může vícekrát opakovat. Části `#elif` a `#else` lze vynechat.

Má-li *podmíněný_výraz_1* hodnotu `true`, přeloží se *zdrojový_text_1* a ostatní části zdrojového textu až po direktivu `#endif` se vynechají. Jinak se testují postupně další *podmíněné výrazy* za direktivou `#elif`. Má-li některý z nich hodnotu `true`, přeloží se odpovídající *zdrojový text* a zbývající části zdrojového textu až po direktivu `#endif` se vynechají.

Nemá-li žádný z *podmíněných výrazů* hodnotu `true` a je-li uvedena část `#else`, přeloží se *zdrojový text* za `#else`. Jinak se nepřeloží žádný *zdrojový text* mezi direktivami `#if` a `#endif`.

Direktivy `#if` lze do sebe vnořovat.

Významná odlišnost direktivy `#if` od analogické direktivy v C++ se týká *podmíněného výrazu*. Jeho základem jsou podmíněné symboly. Identifikátor podmíněného symbolu, který je definován, představuje hodnotu `true`. Identifikátor podmíněného symbolu, který není definován, představuje hodnotu `false`.

Nejjednodušší použití direktivy:

```
#define LADĚNÍ
// ...
#if LADĚNÍ
Console.WriteLine("x = " + x);
#endif
```

Pokud se direktiva `#define LADĚNÍ` zruší, výpis proměnné `x` se neprovede.

Složitější *podmíněné výrazy* mohou obsahovat operátory `==`, `!=`, `||` a `&&`, konstanty `true` a `false` a kulaté závorky. Např.:

```
#if LADĚNÍ == false
Console.WriteLine("x = " + x);
#endif
#if (LADĚNÍ || TESTOVÁNÍ)
Console.WriteLine("y = " + y);
#endif
```

Výpis proměnné `x` se provede, pokud podmíněný symbol `LADĚNÍ` není definován. Výpis proměnné `y` se provede, pokud je definován podmíněný symbol `LADĚNÍ` nebo `TESTOVÁNÍ`.

Na rozdíl od C++ nemohou *podmíněné výrazy* obsahovat číselné konstanty ani konstanty definované pomocí `const`.

Pokud je program v prostředí Visual Studio 2005 překládán v ladícím režimu (v poli Solution Configuration je zvolena položka Debug), je automaticky definován symbol `DEBUG`.