BASIX — dokumentace

Tato dokumentace vychází z technické specifikace jazyka BASIX a vysvětluje jazyk srozumitelně, krok za krokem, s příklady. Cílem je, aby si i úplný začátečník dokázal BASIX osahat a psát v něm jednoduché programy.

Obsah

---

1. Co je BASIX a jak funguje

BASIX je jednoduchý programovací jazyk se speciální syntaxí (příkazy vypadají jako volání funkcí, např. `var(...)`, `set(...)`, `if(...)`). Program napsaný v BASIXu se **nepouští přímo** — nejdřív se **přeloží (zkompiluje) do C++**, a teprve výsledný C++ kód se zkompiluje běžným C++ překladačem (GCC nebo Clang) do spustitelného programu.

Pro vás jako programátora BASIXu to znamená hlavně jednu věc: nemusíte se starat o C++ vůbec — píšete jen v BASIXu a překladač (napsaný v Pythonu) za vás vyrobí odpovídající C++ kód.

Proč je to důležité vědět: C++ má spoustu míst, kde se program při chybě (např. dělení nulou, přetečení čísla) chová *nedefinovaně* — může spadnout, vypsat nesmysl, nebo se "jen" chovat nevyzpytatelně. Tomu se říká **UB** (undefined behavior). BASIX je navržený tak, aby se **nikdy nic podobného nestalo** — místo toho má pro každou takovou situaci pevně dané, předvídatelné chování. To je tak důležité, že je to první ze dvou hlavních zásad celého jazyka (viz další kapitola).

---

2. Dvě klíčové myšlenky, které je dobré pochopit hned na začátku

Celá specifikace BASIXu se drží dvou zásad. Pochopení těchto dvou vět vám usnadní pochopit spoustu "podivných" pravidel dál v textu.

Zásada 1 — žádné nedefinované chování

Cokoliv, co by v C++ vedlo k UB (dělení nulou, přetečení čísla, přístup mimo pole…), má v BASIXu **přesně definované** chování — typicky se hodnota prostě **ořízne (saturuje)** na nejbližší povolenou mez, místo aby program "spadl" nebo se choval nepředvídatelně.

Zásada 2 — obě čísla mají stejný rozsah, `2^53`

BASIX má dva číselné typy, `int` a `real`, a **oba mají stejný rozsah hodnot**: `[-2^53, 2^53]` (tj. přibližně od −9 007 199 254 740 992 do 9 007 199 254 740 992). Díky tomu se dá mezi `int` a `real` převádět bez překvapení — jediné, co se při převodu `real → int` ztrácí, je desetinná část čísla, nikdy se neztrácí rozsah.

> **Proč zrovna `2^53`?** Protože `real` je uvnitř reprezentován jako `double`, a `double` umí přesně (bez zaokrouhlovací chyby) reprezentovat celá čísla právě do velikosti `2^53`. Tím pádem je převod `int → real` vždy bezeztrátový.

---

3. Datové typy: `int` a `real`

BASIX má jen **dva** datové typy pro čísla:x

Typ	Rozsah hodnot	Co reprezentuje	Příklad zápisu
`int`	`[-2^53, 2^53]`, obě meze včetně	celé číslo	`42`, `-5`
`real`	`[-2^53, 2^53]`, obě meze včetně	desetinné číslo	`3.14`, `-0.5`

Žádný jiný základní typ (`bool`, `string`, `char`…) BASIX nemá. Řetězec je ve skutečnosti obyčejné **pole typu `int`**, kde každé číslo je kód znaku (viz kapitola o polích).

3.1 Pravdivost (místo `bool`)

BASIX nemá typ `bool`. Kdekoliv se hodnota používá jako podmínka (`if`, `while`…), platí jednoduché pravidlo, stejné pro `int` i `real`:

- `0` (resp. `0.0`) → **nepravda** (false)
- cokoliv jiného → **pravda** (true)

3.2 Co se stane při přetečení — saturace

Když výpočet vyjde mimo rozsah `[-2^53, 2^53]`, BASIX hodnotu **neoseká nesmyslně jako v C++**, ale “přilepí” ji na nejbližší povolenou mez. Tomu se říká **saturace**.

```
var(int, x, [99999999999999999999])
```

Toto obrovské číslo se hned při kompilaci saturuje na horní mez `2^53`. Stejně tak i výsledek běžného výpočtu za běhu programu (např. násobení dvou velkých čísel) saturuje, místo aby přeteklo do nesmyslné hodnoty.

3.3 Dělení nulou nikdy nehavaruje

Na rozdíl od C++ (kde je celočíselné dělení nulou UB) BASIX definuje:

- `x / 0` → výsledkem je `x` beze změny,
- `x % 0` → výsledkem je nula (`0`, resp. `0.0`).

Takže `5 / 0` dá `5`, a `5 % 0` dá `0`. Žádná chyba, žádný pád programu.

3.4 Modulo (%)

`%` funguje pro `int` i `real` (na rozdíl od C++, kde `%` funguje jen pro celá čísla). Výsledek má vždy znaménko **dělence** (levého operandu):

```
-7 % 2   →  -1
7 % -2   →   1
5.3 % 2.0  →  1.3
-5.3 % 2.0 → -1.3
```

3.5 Implicitní (automatická) konverze mezi `int` a `real`

BASIX umí automaticky převádět mezi `int` a `real`, kdykoliv je to potřeba (např. při uložení

hodnoty jednoho typu do proměnné jiného typu):

Směr	Co se stane	Ztrácí se něco?
`int → real`	hodnota se přenese přesně	ne
`real → int`	ořízne se desetinná část (`3.7 → 3`)	jen desetinná část

3.6 Hierarchie typů — když se v operaci potkají `int` a `real`

Pokud sečtete (nebo jinak zkombinujete) `int` a `real` hodnotu, `int` se nejdřív automaticky převede na `real` a operace celá proběhne v `real`:

```
real > int

var(int, a, [5])
var(real, b, [1.5])
var(real, c, [a + b])   // a se povýší na real, výsledek je 6.5
```

- U **aritmetických** operátorů (`+ - * / %`) se výsledek ještě saturuje na meze `real`.
- U **porovnávacích** operátorů (`< <= > >= == !=`) se saturace neuplatní — výsledkem je vždy `int` hodnota `0` nebo `1`.

Logické operátory (`and`, `or`, `not`) se touto hierarchií vůbec neřídí — každý operand se posoudí jen podle pravdivosti (0/nenula), bez ohledu na typ. Výsledek `and`/`or`/`not` je vždy `int` `0`, nebo `1`.

`and` a `or` se vyhodnocují se **zkráceným vyhodnocením** (short-circuit) — stejně jako &&/|| v C++: u `a and b` se `b` vůbec nevyhodnotí, pokud `a` už samo o sobě je nepravda; u `a or b` se `b` nevyhodnotí, pokud `a` je už pravda.

3.7 Priorita a asociativita operátorů

Od nejvyšší priority k nejnižší:

Priorita	Operátory	Asociativita
1 (nejvyšší)	unární `-`, `not`	zprava doleva
2	`*`, `/`, `%`	zleva doprava
3	binární `+`, `-`	zleva doprava
4	`<`, `<=`, `>`, `>=`	zleva doprava
5	`==`, `!=`	zleva doprava
6	`and`	zleva doprava
7 (nejnižší)	`or`	zleva doprava

Pozor, past pro začátečníky: relační operátory (`<`, `<=`, `>`, `>=`) mají *vyšší* prioritu než rovnostní (`==`, `!=`). Zápis:

```
a < b == c < d
```

se vyhodnotí jako `(a < b) == (c < d)`, ne jak by se zdálo na první pohled.

BASIX **nemá bitové operátory** (`& | ^ ~ << >>`) — v jazyce prostě neexistují.

---

4. Zápis výrazů [. . .]

Toto pravidlo platí naprosto všude v BASIXu, proto si ho zapamatujte hned:

> **Kdekoliv se v kódu použije hodnota výrazu, musí být obalená hranatými závorkami [. . .].**

Platí to i pro obyčejnou proměnnou nebo číselný literál samotný:

```
var(int, x, [5])       // [5] je výraz
set(x, [x + 1])        // [x + 1] je výraz
if([x > 3])            // [x > 3] je výraz
    ...
```

Kulaté závorky `()` k tomuto obalu **nepatří** — ty se používají jen ke klasickému seskupování uvnitř výrazu (`[(a + b) * c]`), stejně jako v jiných jazycích.

4.1 Explicitní (ruční) konverze typu

Kromě běžného `[výraz]` existuje i zápis s ručním určením cílového typu:

```
[typ; výraz]
```

Nejdřív se vyhodnotí `výraz` úplně normálně (ve svém vlastním typu), a teprve **poté** se výsledek převede na `typ`:

```
[int; 3.7]     // 3.7 je real, ořízne se na int → 3
[real; 5]      // 5 je int, převede se na real → 5.0
```

Je to užitečné, když chcete mít kontrolu nad tím, *kdy přesně* se ořízne desetinná část:

```
var(real, w, [int; 3.7])   // 3.7 -> ořízne se na int (3) -> zpět
na real (3.0)
                            // výsledek: w = 3.0 (ne 3.7!)
```

---

5. Proměnné: `var` a `set`

5.1 Deklarace proměnné — `var`

Proměnná v BASIXu vzniká vždy rovnou i s hodnotou:

```
var(typ, jmeno, [výraz])

var(int, vek, [25])
var(real, cena, [19.99])
```

Existuje i zkrácená forma bez hodnoty — proměnná pak dostane výchozí nulu (`0` pro `int`, `0.0` pro `real`):

```
var(typ, jmeno)

var(int, a)     // a = 0
var(real, b)    // b = 0.0
```

Pokud se typ výrazu liší od typu proměnné, hodnota se automaticky převede (viz 3.5):

```
var(int, z, [3.7])    // z dostane 3 (ořízne se desetinná část)
```

5.2 Přiřazení do existující proměnné — `set`

`set` **nezakládá** novou proměnnou, jen mění hodnotu té, která už existuje:

```
set(jmeno, [výraz])

var(int, x, [10])
set(x, [x + 5])    // x je teď 15
```

Typ proměnné se `set`em nikdy nemění — je navždy daný jejím `var(...)`.

5.3 Identifikátory (pojmenování)

- Povolené znaky: `a`–`z`, `A`–`Z`, `0`–`9`, `_`.
- Nesmí začínat číslicí.
- Jsou **case-sensitive** (`promenna` ≠ `Promenna`).
- BASIX nemá konstanty — všechno vytvořené přes `var` je proměnlivé.
- Nesmí se shodovat s žádným klíčovým slovem jazyka (`var`, `if`, `int`…).

5.4 Oddělování příkazů

BASIX používá **jeden příkaz na řádek** — žádný středník na konci není potřeba, konec řádku sám ukončuje příkaz.

---

6. Bloky kódu a `break: code`

`code` seskupí více příkazů do jednoho pojmenovaného bloku:

```
code(label)
    tělo
end code
```

`label` je jméno bloku, které slouží k jeho případnému předčasnému ukončení příkazem `break(label)`:

```
var(int, x, [0])
code(muj_blok)
    set(x, [x + 1])
    break(muj_blok)      // okamžitě ukončí "muj_blok"
    set(x, [x + 100])    // tohle už se neprovede
end code
// x je tady 1
```

Odsazení (mezery navíc před příkazy) je čistě pro čitelnost — na fungování programu nemá vliv. Blok je vymezený jen řádky `code(...)`/`end code`.

`break(label)` může ukončit i **vzdálenější** obklopující blok, ne jen ten nejbližší — pak se ukončí i všechny bloky mezi tím:

```
code(vnejsi)
    code(vnitrni)
        break(vnejsi)   // ukončí OBA bloky najednou
    end code
end code
```

Vnořené bloky nesmí mít stejný `label` jako blok, ve kterém leží (byl by nejednoznačný, který z nich se má ukončit) — ale dva bloky vedle sebe (sourozenci) stejný label mít mohou, protože nikdy neběží současně.

---

7. Cykly: `while`, `loop`, `for`, `dowhile`

BASIX má čtyři druhy cyklů. Všechny podporují nepovinný `label` (se středníkem, když je uveden).

7.1 `while` — cyklus s podmínkou na začátku

```
while([podmínka])
    tělo
end while
```

Podmínka se testuje **před každou iterací** (i tou první). Pokud je hned napoprvé nepravdivá, tělo se neprovede ani jednou.

```
var(int, i, [0])
while([i < 3])
    set(i, [i + 1])
end while
// i = 3 (tělo proběhlo 3x)
```

7.2 `dowhile` — cyklus s podmínkou na konci

Stejné jako `while`, ale podmínka se testuje **až po** proběhnutí těla — takže tělo proběhne **vždy alespoň jednou**:

```
dowhile([podmínka])
    tělo
end dowhile

var(int, x, [0])
dowhile([0])           // podmínka je hned nepravdivá...
    set(x, [x + 1])
end dowhile
// ...ale tělo přesto proběhlo jednou: x = 1
```

7.3 `loop` — cyklus s krokováním (jako klasický `for` z jiných jazyků)

```
loop(promenna, (od, do, krok))
    tělo
end loop
```

- `promenna` **musí být deklarovaná dřív** přes `var(...)`.
- `od`, `do` jsou povinné, `krok` je nepovinný (výchozí je `1`).
- Meze `od`, `do`, `krok` se vyhodnotí **jen jednou**, na začátku.
- Je-li `krok >= 0`, testuje se `promenna <= do`; je-li `krok < 0`, testuje se `promenna >= do`. Obě meze jsou včetně (inclusive).

```
var(int, i, [0])
loop(i, ([0], [10], [2]))
    print([i])
```

```
    print(" ")
end loop
// vypíše: 0 2 4 6 8 10
```

Proměnná `promenna` je normální proměnná — pokud si ji `tělo` samo změní přes `set`, projeví se to i na dalším průběhu cyklu (žádné skryté počítadlo).

7.4 `for` — cyklus přes prvky pole

```
for(promenna, pole)
    tělo
end for
```

Projde postupně všechny indexy pole `pole`, od `1` do jeho velikosti, a ukládá je do `promenna` (musí být předem deklarovaná):

```
var((array, int), a, {10, 20, 30})
var(int, i, [0])
var(int, soucet, [0])

for(i, a)
    set(soucet, [soucet + a[i]])
end for
// soucet = 60
```

Protože pole nikdy nemá velikost 0, `for` vždy provede tělo **alespoň jednou**.

7.5 Srovnání cyklů

Cyklus	Kdy testuje podmínku	Provede tělo alespoň 1x?	Typický účel
`while`	před	ne	opakuj, dokud platí podmínka
`dowhile`	po	ano	opakuj alespoň jednou
`loop`	před, s pevnou aritmetickou posloupností	závisí na mezích	klasický počítaný cyklus
`for`	— (jde vždy přes celé pole)	ano	procházení pole

8. `break` a `continue` — společná pravidla

Sedm druhů bloků v BASIXu — `code`, `while`, `loop`, `for`, `dowhile`, `if`, `case` — sdílí **jednu společnou množinu labelů**: vnořený blok nesmí použít label, který už používá kterýkoliv z jeho obklopujících bloků (kolize = chyba kompilace). Sourozenecké (vedle sebe stojící) bloky ale label sdílet mohou.

8.1 `break`

- **`break(label)`** — ukončí konkrétní pojmenovaný blok (a všechny bloky mezi místem `break` a ním).
- **`break()`** bez labelu — ukončí **nejbližší obklopující cyklus** (`while`/`loop`/`for`/`dowhile`), přeskočí přitom libovolné mezilehlé `code`/`if`/`case` bloky. Použití mimo jakýkoliv cyklus je chyba kompilace.

8.2 `continue`

Přeskočí zbytek **aktuální iterace** (na rozdíl od `break`, který cyklus celý ukončí):

- u `while`/`dowhile` → skočí rovnou na (znovu)vyhodnocení podmínky,
- u `loop`/`for` → nejdřív proběhne krokování, pak se testuje podmínka.

`continue(label)`/`continue()` může cílit **jen na cyklus** (`while`, `loop`, `for`, `dowhile`) — nikdy na `code`, `if`, nebo `case`, protože ty nemají pojem “iterace”.

```
var(int, i, [0])
var(int, soucet, [0])
while([i < 10])
    set(i, [i + 1])
    if([i % 2 == 0])
        case(true)
            continue()      // přeskoč sudá čísla
        end case
    end if
    set(soucet, [soucet + i])
end while
// soucet = 1+3+5+7+9 = 25
```

---

9. Podmínky: čtyři varianty `if`

BASIX má **čtyři** varianty páru `if`/`case`, odlišené tím, co je druhým argumentem `if`.

9.1 Dvouhodnotová varianta — klasické if/else

```
if([podmínka])
    case(true)
        // provede se, pokud je podmínka pravdivá
    end case
    case(false)
        // provede se, pokud je podmínka nepravdivá
    end case
end if
```

- Smí mít **1 nebo 2** `case` bloky, s hodnotou `true`, nebo `false` (nikdy obě stejné).
- Na pořadí `case` bloků nezáleží — vybere se ten, jehož hodnota odpovídá.
- Pokud odpovídající `case` chybí, neprovede se nic.

```
var(int, x, [5])
var(int, y, [0])
if([x > 3])
    case(true)
        set(y, [1])
    end case
    case(false)
        set(y, [-1])
    end case
end if
// y = 1
```

9.2 Řetězená varianta — `if(...; first)` (jako if/else if/else)

```
if(first)
    case([podmínka1])
        ...
    end case
    case([podmínka2])
        ...
    end case
    case()              // volitelný "catch-all", musí být
poslední
        ...
    end case
end if
```

Prochází se case bloky **shora dolů**, provede se **první**, jehož podmínka vyjde pravdivá, a `if` hned poté končí (žádný další case se netestuje ani nevyhodnocuje).

```
var(int, skore, [72])
var(int, znamka, [0])
```

```
if(first)
    case([skore >= 90])
        set(znamka, [1])
    end case
    case([skore >= 75])
        set(znamka, [2])
    end case
    case([skore >= 60])
        set(znamka, [3])
    end case
    case()
        set(znamka, [4])
    end case
end if
// znamka = 3 (72 >= 60, ale první dvě podmínky neplatily)
```

9.3 Vícenásobná varianta — `if(...; every)`

```
if(every)
    case([podmínka1])
        ...
    end case
    case([podmínka2])
        ...
    end case
    case()              // volitelný, provede se vždy, musí být
poslední
        ...
    end case
end if
```

Na rozdíl od `first` se tu **vyhodnotí a případně provedou úplně všechny** case bloky, jejichž podmínka vyjde pravdivá — žádné “vítězství” prvního. Koncový `case()` (bez podmínky) se provede vždy.

```
var(int, n, [15])
var(int, pocet, [0])
if(every)
    case([n % 3 == 0])
        set(pocet, [pocet + 1])
    end case
    case([n % 5 == 0])
        set(pocet, [pocet + 1])
    end case
end if
// pocet = 2, protože 15 je dělitelné 3 i 5
```

9.4 Postupná varianta — `if(...; dive)`

```
if(dive)
    case([podmínka1])
        ...
```

```
    end case
    case([podmínka2])
        ...
    end case
    case()
        ...
    end case
end if
```

Podobné `and` řetězení: pokračuje se dál, **dokud** každá další podmínka vychází pravdivě. Jakmile jedna vyjde nepravdivě, `if` **okamžitě celý skončí** (na rozdíl od `every`, kde se pokračuje dál).

```
var(int, x, [7])
var(int, y, [0])
if(dive)
    case([x > 0])
        set(y, [y + 1])     // provede se
    end case
    case([x > 5])
        set(y, [y + 10])    // provede se
    end case
    case([x > 100])
        set(y, [y + 100])   // NEPRAVDA -> if tady hned končí
    end case
    case()
        set(y, [y + 1000])  // tohle se už nedostane ke slovu
    end case
end if
// y = 11
```

9.5 Přehled rozdílů mezi variantami

	dvouhodnotová	`first`	`every`	`dive`
druhý argument `if`	`[výraz]`	`first`	`every`	`dive`
hodnota `case`	`true`/`false`	libovolný výraz / catch-all	libovolný výraz / catch-all	libovolný výraz / catch-all
max. počet `case`	2	neomezeno	neomezeno	neomezeno
kolik se provede	max. 1	max. 1 (první, co sedí)	libovolný počet	pokud řetěz drží
co dělá nepravda	vybírá druhou větev	jde dál	přeskočí, jde dál	**ukončí celý if**

Tělo `if` smí obsahovat výhradně `case(...) ... end case` bloky — žádný jiný příkaz přímo v `if` psát nejde, musí být uvnitř nějakého `case`.

---

10. Pole: `array`

Pole v BASIXu:

- má **pevnou velikost** (danou při deklaraci, dál se nemění),
- všechny prvky mají **stejný typ** (`int`, nebo `real`),
- indexuje se **od 1** (ne od 0 jako v C++!),
- přístup mimo meze **nikdy nespadne** — index se saturuje na nejbližší platný.

10.1 Deklarace — varianta A (výchozí nuly, velikost jako výraz)

```
var((array, typ), jmeno, [velikost])

var((array, int), a, [4])     // pole velikosti 4, samé nuly:
[0,0,0,0]
```

Pokud vyjde velikost menší než 1, saturuje se na 1 (pole nikdy nemá 0 prvků).

10.2 Deklarace — varianta B (výčet hodnot, velikost dle počtu)

```
var((array, typ), jmeno, {hodnota1, hodnota2, ...})

var((array, int), c, {10, 20, 30})   // pole velikosti 3
```

Do `{}` lze vkládat čtyři druhy položek:

1. **číselný literál** (`10`),
2. **řetězcový literál** v uvozovkách (`"ab"`) — přispěje **jednou hodnotou na znak**, znaky se ukládají jako jejich Unicode kód,
3. **výraz** obalený `[]`, (`[x + 1]`),
4. **jméno jiného pole** obalené `[]`, (`[jine_pole]`) — přispěje všemi jeho prvky (jako zřetězení).

```
var((array, int), jmeno, {"ab"})      // jmeno = [97, 98]
('a'='97', 'b'='98')
var(int, x, [5])
var((array, int), a, {[x + 1], 5})    // a = [6, 5]
```

BASIX nemá typ `string` — text je prostě pole `int` s Unicode kódy znaků. Řetězcový literál `"ahoj"` je jen pohodlnější způsob, jak takové pole naplnit.

10.3 Čtení a zápis prvku

```
jmeno[index]                    // čtení
set(jmeno[index], [výraz])      // zápis

var((array, int), c, {10, 20, 30})
var(int, x, [c[1]])             // x = 10 (indexy začínají od 1!)
set(c[2], [99])                 // c = [10, 99, 30]
```

10.4 Index mimo meze — saturace

```
set(c[-5], [77])       // -5 je moc malé -> uloží se na index 1
var(int, x, [c[100]])  // 100 je moc velké -> vezme se poslední
prvek
```

Pro začátečníky nejdůležitější rozdíl oproti jiným jazykům: BASIX nikdy nespadne na "index out of range" — vždy jen sáhne po nejbližším platném prvku.

10.5 Zápis celého pole najednou (`set`)

`set` umí i tři speciální kombinace zdroj/cíl na celém poli (bez indexu):

A) číslo → pole (zformátuje se jako text):

```
var((array, int), buf, [5])
set(buf, [42])   // buf dostane text "42": [52, 50, 0, 0, 0]
```

B) pole → skalár (rozparsuje se jako číslo):

```
var((array, int), text, {"123"})
var(int, n, [0])
set(n, [text])   // n = 123
```

C) pole → pole (kopie hodnot, prvek po prvku):

```
var((array, int), a, {1, 2, 3})
var((array, int), b, [5])
set(b, [a])      // b = [1, 2, 3, 0, 0]
```

Ve všech třech případech: je-li zdroj kratší než cíl, zbytek cíle se doplní nulami; je-li delší, přebytek se zahodí. Velikost cíle se nikdy nemění.

10.6 Porovnání polí (`==`, `!=`)

Porovnání dvou polí porovnává **jen velikost** (konkrétně velikost posledního rozměru), **ne obsah**:

```
var((array, int), a, {1, 2, 3})
var((array, real), b, {9.0, 9.0, 9.0})
[a == b]   // 1 (obě mají velikost 3, obsah se vůbec neřeší)
```

---

11. Vícerozměrná pole

Zobecnění pole na víc rozměrů (matice, kostky čísel…). Základní vlastnosti jsou stejné jako u 1D pole (pevný tvar, stejný typ prvků, indexace od 1, saturace mimo meze).

11.1 Deklarace

```
var((array, typ), jmeno, ([vyraz1], [vyraz2], ..., [vyrazD]))

var((array, real), matice, ([2], [3]))   // 2×3 matice, samé nuly
```

11.2 Čtení a zápis

```
matice[1][1]                                  // čtení
set(matice[1][2], [2.5])                      // zápis
```

Počet indexů musí přesně odpovídat počtu rozměrů — u 2D pole musíte vždy zadat oba indexy, `matice[i]` samo o sobě není povolené.

11.3 `for` přes vícerozměrné pole

```
for((i, j), matice)
    tělo
end for
```

Chová se jako vnořené cykly, kde **poslední** proměnná (`j`) se mění nejrychleji (“row-major” pořadí) — přesně jako klasické vnořené cykly `for i { for j { ... } }`.

11.4 Praktický příklad — součet matice

```
var(int, radky, [2])
var(int, sloupce, [3])
var((array, real), matice, ([radky], [sloupce]))

set(matice[1][1], [1.5])
set(matice[1][2], [2.5])
set(matice[1][3], [3.0])
set(matice[2][1], [4.0])
set(matice[2][2], [5.5])
set(matice[2][3], [6.0])

var(real, soucet, [0.0])
var(int, i, [1])
var(int, j, [1])

for((i, j), matice)
    set(soucet, [soucet + matice[i][j]])
end for
// soucet = 22.5
```

12. Výstup: `print`

`print` vypisuje text. Má tři varianty:

```
print([výraz])                                  // A: jedna hodnota nebo
celé pole
print({polozka1, polozka2, ...})                // B: série hodnot za
sebou
print()                                         // C: bez argumentu
```

12.1 Co se jak vypíše

- **skalární hodnota** (`int`/`real`) → vypíše se jako číslo (`42`, `3.5`),
- **jméno pole** (`[jmeno_pole]`) → vypíše se **jako text** (znaky podle Unicode kódu prvků),
- v `{}` navíc můžete přímo psát **číselný literál** a **řetězcový literál** v uvozovkách.

```
var(int, x, [42])
var((array, int), jmeno, {"Ahoj"})

print([jmeno])                                  // vypíše: Ahoj
print()                                         // odřádkuje
print({[x], " je odpověď"})                     // vypíše: 42 je
odpověď
```

12.2 `real` se vypisuje vždy s desetinnou tečkou

`3.0` se vypíše jako `"3.0"` (ne `"3"`) — nikdy nezmizí vizuální rozdíl mezi `int` a `real`.

12.3 `print` nikdy sám nepřidává mezery ani odřádkování

```
print({[x], " + ", [y]})   // žádné automatické mezery navíc!
```

Pokud chcete oddělovač nebo nový řádek, musíte ho napsat explicitně (buď jako součást řetězcového literálu, např. `" "` nebo `"\n"`, nebo úplně prázdným voláním `print()`, které vypíše jen jeden konec řádku).

> **Typická chyba začátečníka:** zapomenout na `print()` mezi výpisy a divit se, proč je všechno na jednom řádku slepené k sobě.

---

13. Vstup: `input`

```
input(promenna)
```

`promenna` musí být **předem deklarovaná** — buď skalár (`int`/`real`), nebo jednorozměrné pole.

13.1 Vždy se čte celý řádek

Každé volání `input` přečte a **spotřebuje jeden celý řádek** vstupu — i kdyby proměnná nakonec využila jen jeho část, zbytek řádku propadá a další `input` čte až od dalšího řádku.

13.2 Vstup do skaláru

Přeskočí úvodní bílé znaky a přečte **nejvýše jednu hodnotu** (číslo); zbytek řádku se zahodí.

```
var(int, a, [0])
var(int, b, [0])
input(a)   // vstup "5 10" -> a = 5, zbytek řádku (" 10") se
ZAHODÍ
input(b)   // čte NOVÝ řádek, ne zbytek předchozího!
```

13.3 Vstup do pole

Načte celý řádek jako text, znak po znaku, **až do velikosti pole**:

```
var((array, int), retezec, [5])
input(retezec)   // vstup "Ahoj svete" -> uloží se jen "Ahoj" (5
znaků)
```

Kratší text → zbytek pole se doplní nulami. Delší text → přebytek se zahodí.

13.4 Neplatný nebo chybějící vstup → nula, ne chyba

Nejde-li vstup rozpoznat jako platné číslo (nebo je konec vstupu), skalár prostě dostane výchozí `0`/`0.0` — žádná chyba za běhu.

---

14. Vestavěné funkce `max`/`min`

`max`/`min` mají čtyři různé podoby podle toho, co jim předáte:

Zápis	Co dělá
`max([výraz])` / `min([výraz])`	vrátí horní/dolní mez typu daného výrazu (`2^53`, resp. `-2^53`)
`max(pole)` / `min(pole)`	vrátí nejvyšší/nejnižší platný **index** pole (u `min` je to vždy `1`)
`max((pole, [rozměr]))` / `min(...)`	totéž, ale pro konkrétní rozměr vícerozměrného pole
`max(obj_instance)` / `min(...)`	vrátí počet členů `obj` / vždy `1`

14.1 Nejčastější použití — délka pole

BASIX nemá funkci `length`. Místo toho slouží `max(jmeno)`, protože pole je indexováno od 1, takže nejvyšší index = počet prvků = délka:

```
var((array, int), c, {10, 20, 30})
var(int, i, [min(c)])          // i = 1 (start)
while([i <= max(c)])           // max(c) = 3 (délka pole)
    print([c[i]])
    print()
    set(i, [i + 1])
end while
```

To je velmi užitečné, pokud nechcete velikost pole “natvrdo” vypisovat do kódu.

14.2 Skalární podoba — jen ukázka mezí typu

```
var(int, a, [max([5])])    // a = 9007199254740992 (2^53), protože
[5] je int
```

Tahle podoba se v praxi hodí spíš zřídka (např. jako inicializační “nekonečno” pro hledání minima/maxima v poli).

---

15. Vlastní datový typ: `obj`

`obj` je obdoba `struct` z C++ — vlastní pojmenovaný typ složený z pojmenovaných členů.

15.1 Definice typu

```
obj(jmeno_typu)
    var(typ, jmeno_clenu, [vyraz])
    var(typ, jmeno_clenu2)
end obj

obj(bod)
    var(int, x)
    var(int, y)
end obj
```

Členem `obj` může být `int`, `real`, pole (jednorozměrné i vícerozměrné), nebo i jiný, dříve definovaný `obj`.

15.2 Vytvoření instance (proměnné)

```
var((obj, typ), jmeno)

var((obj, bod), p)   // p.x i p.y jsou hned nastavené na 0
```

Žádná další hodnota se nezadává — instance se vždy naplní výchozími hodnotami členů podle jejich definice v `obj`.

15.3 Čtení a zápis členů (tečková notace)

```
jmeno_instance.jmeno_clenu                // čtení
set(jmeno_instance.jmeno_clenu, [vyraz]) // zápis

var((obj, bod), p)
set(p.x, [10])
set(p.y, [20])
var(int, soucet, [p.x + p.y])   // 30
```

Tečky se dají řetězit, pokud je člen sám typu `obj`:

```
obj(usecka)
    var((obj, bod), zacatek)
    var((obj, bod), konec)
end obj

var((obj, usecka), u)
set(u.zacatek.x, [1])
set(u.konec.x, [4])
```

15.4 Výchozí hodnota se počítá znovu při každé instanci

```
var(int, pocitadlo, [10])

obj(citac)
    var(int, hodnota, [pocitadlo])
end obj

var((obj, citac), c1)     // c1.hodnota = 10
set(pocitadlo, [20])
var((obj, citac), c2)     // c2.hodnota = 20 (pocitadlo se mezitím
změnilo!)
```

15.5 `obj` se nekonvertuje na nic jiného

Mezi `obj` a čísly ani poli **neexistuje žádná konverze** — pokud byste ji zkusili použít (např. `set` mezi `obj` a `int`), nic se tiše neprovede (hodnota zůstane beze změny), ale nejde o chybu kompilace.

---

16. Funkce: `fun`

16.1 Definice funkce

```
fun(jmeno_funkce)
    ret(typ, jmeno_navratove_hodnoty)
    par(typ, jmeno_parametru1)
    par(typ, jmeno_parametru2)
    tělo
end fun
```

- `ret(...)` je **povinný** a musí být vždy **první** deklarace (i s výchozí hodnotou, stejnou syntaxí jako `var`).
- `par(...)` je libovolný počet parametrů (i nula), vždy až za `ret`.

```
fun(soucet)
    ret(int, vysledek, [0])
    par(int, a)
    par(int, b)
    set(vysledek, [a + b])
end fun
```

16.2 Návrat hodnoty — `return`

```
return
```

Holé klíčové slovo, bez argumentu. Okamžitě ukončí funkci a jako návratová hodnota se použije **aktuální hodnota `ret`** v tu chvíli. Pokud program nikdy `return` nepoužije, funkce prostě doběhne až na `end fun` — i to je platný, rovnocenný způsob návratu (návratová hodnota je pak hodnota `ret` na konci těla).

16.3 Volání funkce

```
jmeno_funkce([vyraz_pro_ret], [vyraz_pro_par1], [vyraz_pro_par2])
```

První argument jde vždy do `ret` (jeho počáteční hodnota), **druhý a další** postupně do jednotlivých `par` (podle pořadí jejich deklarace).

```
soucet([0], [3], [4])    // zavolá funkci, výsledek se ale nikam neuloží
```

Chybějící argumenty dostanou výchozí hodnotu svého typu — volání úplně bez argumentů (`jmeno_funkce()`) je vždy platné.

16.4 Zachycení návratové hodnoty

Volání funkce je vždy **samostatný příkaz** — nedá se vložit doprostřed výrazu jako `[foo(1) + 2]`. Pro uložení výsledku existuje speciální forma `set`:

```
set(cil, jmeno_funkce([vyraz1], [vyraz2], ...))
```

```
var(int, vysledek, [0])
set(vysledek, soucet([0], [3], [4]))   // vysledek = 7
```

16.5 Předávání parametrů — hodnotou vs. odkazem

- **`int`/`real`** parametry se předávají **hodnotou** — funkce dostane vlastní kopii, změny uvnitř funkce se venku vůbec neprojeví.
- **pole a `obj`** parametry se předávají **odkazem** — funkce pracuje přímo s tou samou instancí jako volající; jakákoliv změna prvku pole nebo členu `obj` uvnitř funkce se hned projeví i venku.

```
fun(vynuluj)
    ret(int, x, [0])
    par((array, int), pole)
    var(int, i, [min(pole)])
    while([i <= max(pole)])
        set(pole[i], [0])
        set(i, [i + 1])
    end while
end fun

var((array, int), a, {1, 2, 3})
vynuluj([0], [a])
// a je teď [0, 0, 0] - funkce ho měnila přímo, protože pole jde
odkazem!
```

Pro argumenty typu pole/`obj` se místo `[výraz]` píše zvláštní tvar `[jmeno]` — musí to být **holé jméno** proměnné (žádná indexace, operace ani výraz) a jejího typu se musí **přesně** shodovat s typem parametru.

16.6 Rekurze

Funkce smí volat sama sebe (přímá rekurze), i vzájemně jiné funkce (nepřímá rekurze) — všechny funkce v BASIXu žijí na jedné, společné globální úrovni a **nezáleží na pořadí**, v jakém jsou v souboru napsané:

```
fun(faktorial)
    ret(int, vysledek, [1])
    par(int, n)
    if([n > 1])
        case(true)
            var(int, dalsi, [0])
            set(dalsi, faktorial([1], [n - 1]))
            set(vysledek, [n * dalsi])
        end case
    end if
end fun
```

Každé (i rekurzivní) volání má vlastní, nezávislou sadu `ret`/`par` — vnořené volání nijak neovlivní hodnoty ve volání, ze kterého bylo zavoláno.

16.7 Kde smí funkce žít

Funkce se smí definovat **jen na nejvyšší (globální) úrovni** programu — nikdy uvnitř jiného bloku (`while`, `if`…) ani uvnitř jiné funkce.

---

17. Souhrn klíčových slov a rychlá reference

Klíčová slova s vlastní (

`var`, `set`, `code`, `while`, `loop`, `for`, `dowhile`, `if`, `case`, `break`, `continue`, `obj`, `max`, `min`, `input`, `print`, `fun`, `ret`, `par`

Holá klíčová slova (bez ()

`end`, `return`, `true`, `false`

Klíčová slova typu

`int`, `real`

Rychlá tabulka příkazů

Co chci udělat	Zápis
nová proměnná	`var(typ, jmeno, [výraz])`
změna hodnoty	`set(jmeno, [výraz])`
pojmenovaný blok	`code(label) ... end code`
ukončit blok/cyklus	`break(label)` nebo `break()`
přeskočit iteraci	`continue(label)` nebo `continue()`
cyklus s podmínkou	`while([výraz]) ... end while`
cyklus s podm. na konci	`dowhile([výraz]) ... end dowhile`
počítaný cyklus	`loop(prom, ([od],[do],[krok])) ... end loop`
cyklus přes pole	`for(prom, pole) ... end for`
klasický if/else	`if([výraz]) case(true)...case(false)...end if`
if/else if/else	`if(first) case([v])...case()...end if`
provést všechny platné	`if(every) case([v])...end if`
řetěz podmínek (AND)	`if(dive) case([v])...end if`
nové pole (nuly)	`var((array, typ), jmeno, [velikost])`
nové pole (hodnoty)	`var((array, typ), jmeno, {v1, v2, ...})`
čtení prvku pole	`jmeno[index]`
zápis prvku pole	`set(jmeno[index], [výraz])`
délka pole	`max(jmeno)`
výpis	`print([výraz])`, `print({...})`, `print()`
načtení vstupu	`input(promenna)`
nový typ	`obj(jmeno) ... end obj`
instance typu	`var((obj, typ), jmeno)`
člen instance	`jmeno.clen`
nová funkce	`fun(jmeno) ret(...) par(...) ... end fun`
návrat z funkce	`return`
volání funkce	`jmeno([v1], [v2], ...)`
volání se zachycením	`set(cil, jmeno([v1], ...))`

18. Nejčastější chyby začátečníků

1. **Zapomenutí hranatých závorek kolem výrazu.** `var(int, x, 5)` — správně: `var(int, x, [5])`.
2. **Indexace od 0.** V BASIXu se pole indexují **od 1**, ne od 0 jako v C++/Javě/Pythonu. `pole[0]` je platný zápis, ale saturuje na `pole[1]` — takže se tato chyba nezhroutí, jen tiše udělá něco jiného, než jste čekali.
3. **Očekávání pádu programu při chybě.** BASIX nikdy "nespadne" na dělení nulou nebo indexu mimo pole — vždy vrátí definovanou hodnotu. Pokud program dělá něco divného, hledejte saturaci, ne chybovou hlášku.
4. **Volání funkce uvnitř výrazu.** `[foo(1) + 2]` je neplatné — volání funkce musí být samostatný příkaz. Použijte `set(x, foo([1]))` a teprve pak `[x + 2]`.
5. **Zapomenutý `print()` mezi výpisy.** `print` nepřidává žádné mezery ani konce řádku sám od sebe — pokud potřebujete oddělit výstupy, musíte to napsat explicitně.
6. **Míchání variant `if`.** Nelze v jednom `if(...)` kombinovat `[výraz]`/`first`/`every`/`dive` — vyberte si jednu variantu a držte se jí v rámci daného `if`.
7. **Předávání pole "hodnotou".** Pole a `obj` se do funkcí předávají **odkazem** — pokud funkce uvnitř pole změní, projeví se to i po návratu z funkce. Pokud chcete pracovat s kopií, musíte si ji sami vytvořit přes `set(kopie, [original])`.
8. **Zapomenutí, že `promenna` u `loop`/`for` musí existovat předem.** Na rozdíl od `var`, `loop` a `for` **nezavádí** novou proměnnou — musí být deklarovaná už dřív přes `var(...)`.
9. **Očekávání, že `real` se vypíše bez `.0`.** Hodnota `3.0` se vždy vypíše jako `"3.0"`, nikdy jako `"3"` — je to úmyslné, aby šlo na výstupu poznat typ hodnoty.