# 3 Principy vyšších programovacích jazyků

Kreslíková, 22. 9. 2023





V akademickém roce 2023/2024 je přednáška od kapitoly 3.5, (**zvýrazněno červeně**.)

## 3.1 Úvod

Každý program pracuje s daty. Bez nich by neměl smysl. V této kapitole si ukážeme, s jakými základními typy dat můžeme pracovat a jak se v paměti počítače ukládají. Ukážeme si to na datových typech jazyka C, protože jeho kolekce základních typů poměrně přesně kopíruje to, co máme k dispozici na úrovni hardwaru, tedy to, s čím pracuje přímo procesor.

Dále se podíváme, z jakých základních elementů se skládá programovací jazyk, jak se deklarují a definují proměnné, a jak se dají z proměnných skládat výrazy pomocí základních operátorů. Tyto pojmy jsou demonstrovány na jazyce C, ale mějte stále na paměti, týkají se i jiných programovacích jazyků i přesto, že jejich syntaxe může být odlišná.

Všechna data, která se na počítači zpracují, se musí nějakým způsobem do počítače uložit. Uložená data jsou nakonec dekomponována na jednotlivé bity, ale psát programy, jež zpracovávají výlučně bity je nepohodlné. Zavedení ***datového typu*** nám umožní určit, jakým způsobem budeme konkrétní sady bitů používat a operátory (funkce) nám umožní určit operace, jež budeme nad daty provádět.

## 3.2 Datové struktury

Organizace dat pro zpracování je při vývoji počítačového programu významným krokem. Pro mnoho aplikací je výběr správné datové struktury nejvýznamnějším rozhodnutím celé implementace. Jakmile je tento výběr proveden, jsou již příslušné algoritmy jednoduché. Pro stejná data vyžadují některé struktury více nebo méně místa než jiné. Pro stejné operace nad daty vedou některé struktury k více či méně efektivním algoritmům než jiné. Volba algoritmu a ***datové struktury*** jsou navzájem propojeny a správnou volbou se snažíme najít cesty k úspoře času a místa. Datová struktura není pasivním objektem. Musíme přihlížet k operacím, které nad ní mají být provedeny (a na algoritmy použité pro tyto operace). Tento koncept je formalizován v pojmu datový typ.

Programy zpracovávají informace odvozené od popisu světa, v němž žijeme, ať již matematických nebo popisů v přirozené řeči. Podle toho pak výpočetní prostředí vyžaduje poskytnutí zabudované podpory pro základní prvky takových popisů, tj. pro čísla a znaky.
Pro reprezentaci čísel používáme pevné počty bitů. Rozsah čísel pak závisí na konkrétním počtu bitů, jež použijeme k jejich reprezentaci. Zobrazitelná celá čísla jsou podmnožinami čísel přirozených nebo celých.

Zobrazitelná čísla v racionálních datových typech jsou podmnožinou čísel racionálních. V literatuře, zabývající se programováním, se můžeme setkat s poněkud nepřesným tvrzením, že pomocí těchto datových typů zobrazujeme reálná čísla (v některých jazycích je trochu nešťastně racionální datový typ označen identifikátorem **real**).

Čísla v plovoucí řádové čárce se blíží reálným číslům a počet bitů nutných k jejich reprezentaci souvisí s přesností, s níž chceme reálné číslo vyjádřit. V jazyce C se tento typ jmenuje **float** (od floating point).

**Vlastnosti datových struktur**
Datová struktura je:

- **homogenní** - všechny komponenty dané struktury jsou téhož typu.
  *Příklad:* Pole záznamů (i když záznam sám o sobě je heterogenní)
- **heterogenní** - komponenty struktury nejsou téhož typu.
  *Příklad:* Záznam.
- **statická** - nemůže měnit v průběhu výpočtu počet svých komponent ani způsob uspořádání.
- **dynamická** - může měnit v průběhu výpočtu počet svých komponent a způsob uspořádání struktury.

*Příklad*: uspořádaný binární strom lze transformovat např. na obousměrně vázaný lineární seznam a naopak.

Vždy si však musíme uvědomit, že jak homogennost/heterogennost, tak statičnost/dynamičnost se posuzuje vždy jako vlastnost na jisté úrovni abstrakce. Snížíme-li se na úroveň stroje, pak jsou data vždy homogenní a statická.

## *3.3 Datové typy*

Návrh datových struktur je v moderních programovacích jazycích podporován koncepcí datových typů a tzv. typovou kontrolou. ***Datový typ*** je množina hodnot a množina operací nad těmito hodnotami. Každý objekt (konstanta, proměnná, výraz, funkce) v těchto jazycích přísluší právě k jednomu datovému typu. Typ proměnných se zavádí v definici a v deklaraci proměnné. Typ výrazu je dán operátory a operandy, které jsou použity ve výrazu. Příslušnost k typu je přitom syntaktickou vlastností objektu, takže pravidla jazyka určují, v jakých souvislostech lze používat objekty kterých typů. Když zapisujeme operaci, musíme se ujistit, že její operandy i výsledek jsou správného typu.

V některých situacích se provádí ***implicitní konverze typu*** (nutno se seznámit s pravidly pro implicitní konverze). Jindy používáme přetypování neboli ***explicitní konverzi***.

*Příklad*: *x* a *n* jsou celá čísla, výraz: *((float)x) / n* - přetypování, implicitní konverze.

V moderním programování uvažujeme o datových typech více v souvislosti s potřebami programů, než s možnostmi stroje, s důrazem na přenositelnost programů.

*Příklad*: uvažujeme o **short int** jako o datovém objektu, jenž může zaručeně pojmout hodnoty mezi -32768 a 32767, a ne jako o šestnáctibitovém objektu. Navíc koncept celého čísla zahrnuje operace, jež nad ním provádíme: součet, násobení atd.

### 3.3.1 Vlastnosti datových typů

**Datový typ je určen:**

- názvem (**logical, Boolean, int, real, float,** *den, pohlaví*).
- množinou hodnot, kterých mohou nabývat konstanty, proměnné, výrazy, funkce určitého typu.
  *Příklad*: **true, false,** 358, 2.5, "Ahoj", sobota, 'a'.
  Počet různých hodnot příslušejících typu T se nazývá ***kardinalita typu*** T.
- množinou operací (každý operátor nebo funkce předpokládá operandy stanoveného typu a dává výsledek stanoveného typu.
  Tři úrovně dostupných operací:
  1. operace se standardními datovými typy (aritmetické operace, logické operace),
     *Příklad:* **%, and, or, not, eq, xor, +, /, *, in**
  2. operace dostupné v knihovnách funkcí (např. funkce pro práci s lineárními seznamy, vektory, frontou, zásobníkem, atd. ),
     *Příklad: listInit, copyFirst, deleteFirst*
  3. operace definované uživatelem (programátorem).
     *Příklad: vycisliPolynom*

Prostudujte si: *Hornerovo schéma*

Jestliže operátor připouští operandy několika typů (např. / pro dělení jak čísel reálných, tak čísel celých), pak se typ operace / určí podle dalších pravidel jazyka.

*Příklad:*
**int** / **int** … celočíselné dělení – výsledkem je **int**
**int** / **float** … dělení v pohyblivé řádové čárce – výsledkem je **float**
**float** / **int** … dělení v pohyblivé řádové čárce – výsledkem je **float**
**float** / **float** … dělení v pohyblivé řádové čárce – výsledkem je **float**

- množinou atributů (které vlastnosti objektů daného typu jsou programově dosažitelné).
  *Příklad:* *FIRST*, *LAST* - pro datový typ **integer** v Adě.
  Deklarujeme-li v Adě proměnnou *A*: **integer**; pak nejvyšší zobrazitelné celé číslo přiřadíme do této proměnné použitím atributu *LAST* :
  `A := integer LAST;`

  V jazyce C pro celočíselné typy: knihovna <limits.h> [C99, str. 441]

| | | | |
|---|---|---|---|
| *CHAR_BIT* | *SCHAR_MIN* | *SCHAR_MAX* | *UCHAR_MAX* |
| *CHAR_MIN* | *CHAR_MAX* | *MB_LEN_MAX* | *SHRT_MIN* |
| *SHRT_MAX* | *USHRT_MAX* | *INT_MIN* | *INT_MAX* |
| *UINT_MAX* | *LONG_MIN* | *LONG_MAX* | *ULONG_MAX* |
| *LLONG_MIN* | *LLONG_MAX* | *ULLONG_MAX* | |

  **int** *a* = *INT_MAX* – přiřazení maximální dosažitelné hodnoty do proměnné typu **int** (v daném prostředí).
  V jazyce C pro reálné typy: knihovna <float.h>

*Úkol:* zjistěte ve vašem vývojovém prostředí všechny maximální dosažitelné hodnoty celočíselných a racionálních typů.

### 3.3.2 Definování datových typů.

Ve většině případů se nový datový typ definuje pomocí dříve definovaných datových typů. Hodnoty takových typů jsou obvykle složeny z hodnot komponent dříve definovaných ustavujících (***konstitučních***) typů a proto těmto složeným hodnotám říkáme ***strukturované***. Jsou-li všechny hodnoty téhož konstitučního typu, pak tomuto typu říkáme ***bázový typ***. Konstituční typy mohou být opět samy strukturovány a tak lze vytvářet hierarchicky uspořádané struktury. Komponenty na nejnižší úrovni však jsou již atomické - dále nedělitelné. Proto je nutné, aby byly zavedeny také primitivní (jednoduché), nestrukturované typy.

Hodnoty primitivních typů se mohou stanovit vyjmenováním (výčtem, enumerací). Mezi primitivními typy jsou tzv. ***standardní typy***, které jsou předdefinovány. V programovacích jazycích primitivní typy obvykle zahrnují čísla, logické hodnoty a znaky. V případě deklarace vyjmenováním jsou jejich hodnoty seřazeny podle pořadí v posloupnosti vyjmenování. S těmito nástroji lze definovat primitivní typy a vytvářet složité datové struktury do libovolné vnořené úrovně. S ohledem na praktické problémy reprezentace dat a jejich využití je nutné, aby vyšší programovací jazyky měly několik metod strukturování, které se liší operátory, jimiž se konstruují hodnoty struktur a operátory, jimiž se vybírají komponenty ze strukturované hodnoty. Strukturované hodnoty se vytvářejí ze svých komponent pomocí tzv. ***konstruktorů***. Hodnotu jedné komponenty lze ze strukturované hodnoty získat pomocí ***selektoru***.

Mezi základní metody strukturování patří:

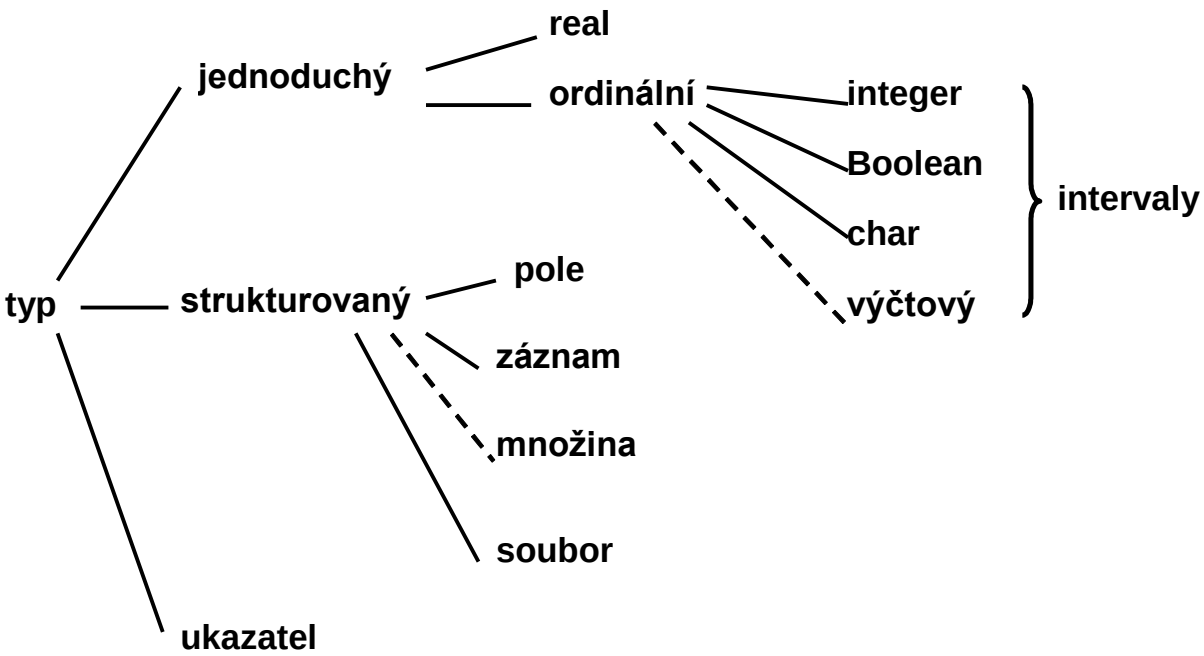
- pole,
- záznam,
- množina,
- soubor.

Strukturované datové typy budou probrány později.

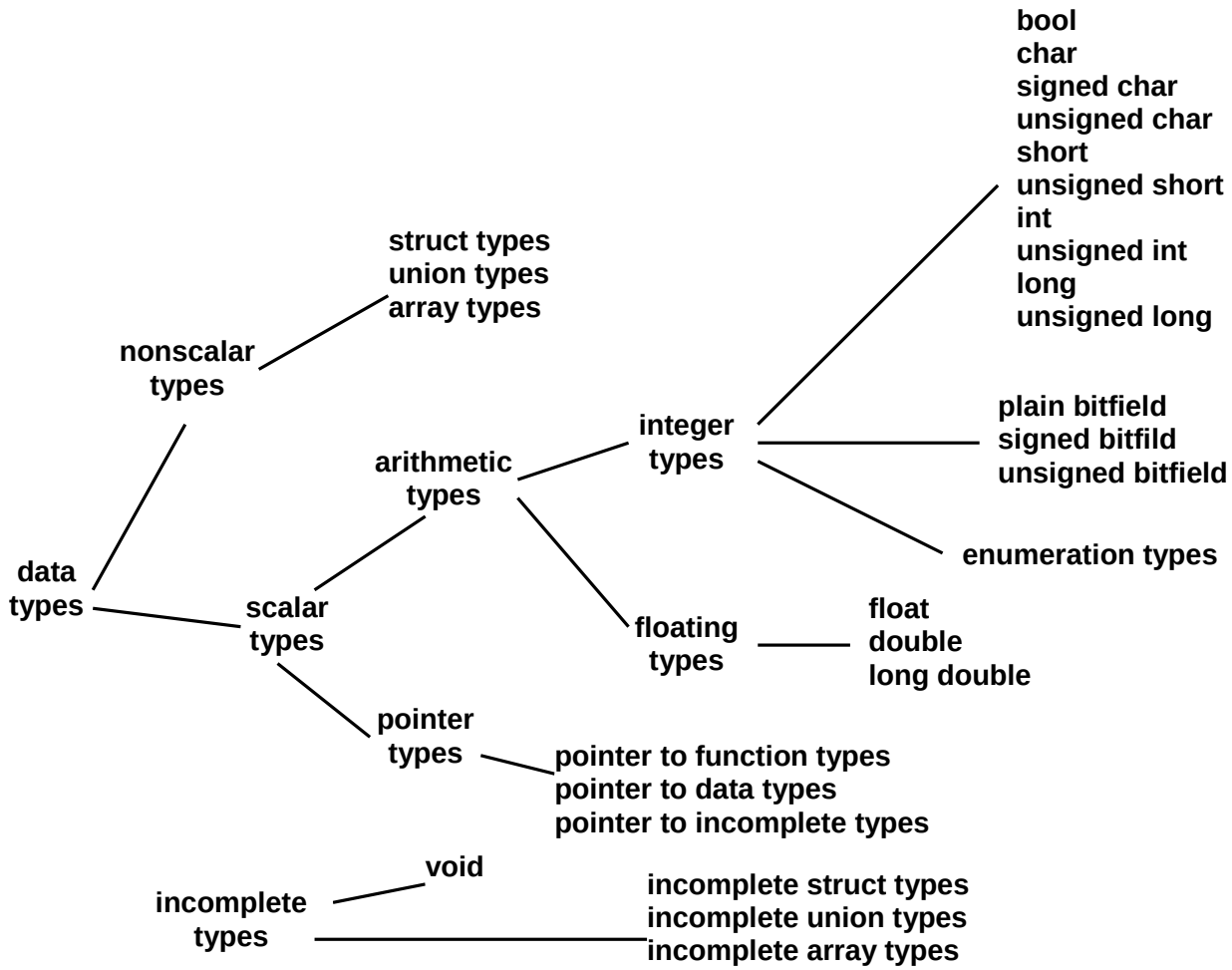
### 3.3.3 Základní operátory datových typů

- ekvivalence - je definována nad primitivními typy, množinami a textovými řetězci (v jazyce C datový typ množina neexistuje, ale v některých jiných jazycích ano).
- přiřazení - operace přiřazení je definována obvykle nad všemi datovými typy; musíme si však uvědomit, že pro rozsáhlé a hluboce strukturované typy může taková operace představovat značné množství strojových instrukcí. U strukturovaných typů může být realizace operace přiřazení realizována různým způsobem (např. záznam lze přiřadit pouze po jednotlivých složkách).
- transformační operátory - mapují jeden datový typ do jiného (přetypování).

### 3.3.4 Rozdělení datových typů




**Obrázek 1.:** Datové typy v jazyce Pascal.




**Obrázek 2.:** Klasifikace typů v jazyce C. [C99]

**Signed (**znaménkový**) / unsigned (**neznaménkový**)**

Rozdíl mezi signed a unsigned je v rozsahu čísla:

**unsigned** od 0 do $2^n - 1$ pro **char** 0 až 255
**signed** od $-2^{n-1}$ do $2^{n-1} - 1$ pro **signed char** -128 až +127

| bit číslo | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| MSB | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | LSB |

MSB – **m**ost **s**ignificant **b**it
LSB – **l**east **s**ignificant **b**it

Standard C zaručuje, že jednotlivé typy z hlediska alokace paměti jsou ve vztahu:

**sizeof(char)** = 1 Bajt
**sizeof(short int)** ≤ **sizeof(int)** ≤ **sizeof(long int)**
**sizeof(unsigned int)** = **sizeof(signed int)**
**sizeof(float)** ≤ **sizeof(double)** ≤ **sizeof(long double)**

| datový typ | počet bitů | význam |
|---|---|---|
| **char, unsigned char, signed char** | 8 | znak |
| **short**, **unsigned short, signed short** | 16 | krátké celé číslo |
| **int**, **unsigned int, signed int** | 16 nebo 32 | celé číslo |
| **long**, **unsigned long, signed long, long long** | 32\|64 | dlouhé celé číslo |
| **enum** | 8\|16\|32 | výčtový typ |
| **float** | 32 | racionální číslo |
| **double** | 64 | racionální číslo s dvojitou přesností |
| **long double** | 80\|96 | dlouhé racionální číslo |
| pointer | 16\|32\|64 | ukazatel |

**Tabulka 1.:** Přehled datových typů v jazyce C, spolu s paměťovými nároky a jejich českým významem.

### 3.3.5 Jednoduché datové typy

***Jednoduché datové typy*** se nazývají jednoduchými proto, že patří k nejzákladnějším typům dat, se kterými se dá manipulovat. Složitější datové typy jsou ve skutečnosti jen kombinacemi jednoduchých datových typů. Jednoduché datové typy jsou stavebními kameny, ze kterých se ostatní typy budují. Jednoduché datové typy představují opravdový základ veškerých výpočtů. Patří k nim znaky, čísla a logické hodnoty (Booleovský typ - **bool**). Jednoduché datové typy mají často přímou souvislost s možnostmi stroje (velikost registrů, organizace čísel na paměťovém médiu). Nad všemi jednoduchými datovými typy jsou definovány operace přiřazení, relace ekvivalence a uspořádání.

**Standardní jednoduché datové typy**

- **Typ int (integer)**
  Tento typ reprezentuje množinu celých čísel, která je v daném programovacím jazyce k dispozici jako standardní datový typ.
  Na rozdíl od matematického chápání množiny celých čísel jako množiny, která je nekonečná a je podmnožinou reálných čísel, datový typ **integer** není nekonečná množina.
  Konečnost kterékoliv množiny čísel reprezentované v počítači je nutným důsledkem konečnosti jeho paměti.

  Aritmetické operátory: +, -, *, /, %
  Relační operátory: ==, !=, <, ≤, ≥, >

  Přeplnění výsledku operace:
  Obecně neplatí asociativní zákon. Je-li např. typ **integer** definován jako množina celých čísel |x| <= 100, + operace sečítání,

  pak: (x + y) + z = x + (y + z), by nemuselo platit - záleží na implementaci
  50 60 -40

- **Typ float (real)**
  Skutečnost, že v počítači reprezentované obory hodnot jsou vždy konečnými množinami, má silný dopad na reprezentaci reálných čísel. Na rozdíl od typu **integer**, kde předpokládáme přesné výsledky operací, jsou výsledky s operandy typu **float** vždy pouze přibližné, aproximující přesné hodnoty. Příčinou je vlastnost množiny reálných čísel (kontinua) - libovolně malý interval na reálné ose obsahuje nekonečně mnoho reálných čísel.
  V počítači typ **float** proto nereprezentuje nekonečnou a nespočetnou množinu reálných čísel, ale konečnou množinu reprezentativních intervalů reálného kontinua - konečnou podmnožinu racionálních čísel.

  *Příklad:* Předpokládejme, že máme reálné číslo reprezentovat pouze prostřednictvím čtyř číslic. Pak číslo:
  1.345 reprezentuje interval <1.345, 1.346)
  1035. - " - <1035.;1036.)

**Pohyblivá řádová čárka**
Číslo x můžeme vyjádřit ve tvaru: $x = m \times B^e$, kde $-E < e < E$, $-M < m < M$,
m je mantisa, e je exponent, B, E, M charakterizují danou reprezentaci reálných čísel na konkrétním počítači.
Kanonický nebo normalizovaný tvar se definuje podmínkou:
$M / B \leq m < M$ nebo $1 / B \leq m / M < 1$
Aritmetické operátory: +, -, *, /
Realizace elementárních operací na množině čísel typu **float** musí zaručovat splnění určitých (minimálních) podmínek :

**Axiomy aritmetiky v pohyblivé řádové čárce**
A1. Typ **float** P je konečnou podmnožinou reálných čísel R. $P \subset R$
A2. Každému číslu $x \subset R$ odpovídá číslo $x' \subset P$; x' se nazývá reprezentant čísla x.
A3. Existuje hodnota max taková, že pro všechna x, pro něž $|x| > max$ jsou x' nedefinovány.
Je-li výsledkem operace číslo x, pro které $|x| > max$, pak stejně jako v případě čísel typu **integer** mluvíme o chybě přeplnění (overflow).
**A4**. symetrie základních operací vzhledem k 0:
$X - Y = X + ( -Y ) = - ( Y - X )$
$( -X ) * Y = X * ( -Y ) = - ( X * Y )$
**A5**. komutativnost:
$X + Y = Y + X$
$X * Y = Y * X$
**A6**. monotónnost:
Pokud $0 \leq X \leq A$ a $0 \leq Y \leq B$, pak
$X + Y \leq A + B$
$X * Y \leq A * B$
$X - B \leq A - Y$

Potenciálně nebezpečnými operacemi aritmetiky čísel typu **float** jsou operace odčítání a dělení.
U všech aritmetických operací je nutné také počítat s chybou zaokrouhlení.
Jsou-li odečítána dvě velmi blízká malá čísla, pak se ve výsledku mohou ztratit téměř nebo úplně.

*Příklad:* $2e^{-40} - 1e^{-40} \rightarrow 0.000000$

Podobně je-li dělitel velmi malý, pak snadno dojde k přeplnění výsledku dělení.
Nevhodný výraz : $abs(t/x) \leq eps$, ale je nutné použít:

$abs(t) \leq eps \times abs(x)$

Převodní funkce **b** --> **integer**: **int** (x) např. 3.545 → 3
Zaokrouhlení[1]: *round (x) ÷ trunc (x + 0.5)*, pro x ≥ 0, např. 3.545 → 4
*round(x) ÷ - trunc (0.5 - x)*, pro x < 0, např. -3.545 → -4

- **Typ character (char)**
  Zahrnuje konečnou a uspořádanou množinu znaků (kódování znaků).
  Existují standardizované soubory znaků:
  - Kódy ISO (International Organisation for Standardisation)
    Mezinárodní standars ISO/IEC 10646:2017, definuje Universal Character Set (UCS).
    [on line, cit. 2017-09-25]
  - ASCII (American Standard Code for Information Interchange)**,** [on line, cit. 2018-09-24]
    tabulka ASCII: http://www.labo.cz/mft/matasciit.htm, [on line, cit. 2018-09-24]
  - EBCDIC (Extended Binary Coded Decimal Interchange), [on line, cit. 2019-09-25]
  - Unicode: http://cs.wikipedia.org/wiki/Unicode, [on line, cit. 2019-09-24] je tabulka znaků všech existujících abeced.
    Unicode standard: http://cs.wikipedia.org/wiki/Unicode,
    Poslední verze: https://www.unicode.org/versions/Unicode14.0.0/
    [on line, cit. 2021-09-14]
  - UTF: http://cs.wikipedia.org/wiki/UTF-8, [on line, cit. 2018-09-24] je zkratka pro UCS Transformation Format. Je to způsob kódování řetězců znaků Unicode/UCS do sekvencí bajtů. [on line, cit. 2018-09-24]
  - Proč Unicode? - http://www.cestina.cz/kodovani/unicode.html, [on line, cit. 2018-09-24]

- **Typ bool (Boolean)**
  Hodnoty typu **bool** = (**false**,**true**)
  Operace následník(**false**) = **true**, předchůdce(**true**) = **false**,
  pořadí(**false**) = 0, pořadí(**true**) = 1
  Operátory **&&(AND)**, || (**OR**), **!** (**NOT**).

  Výsledek typu **Boolean** dávají také všechny relační operátory (==, !=, <, ≤, ≥, >). (Pozor v jazyce C – tam je výsledkem relačních operátorů typ **int**!)

- **Typ interval**
  V jazyce C tento datový typ není zaveden.
  Lze definovat pouze nad ordinálními typy (v některých programovacích jazycích).
  **type** *T* = min .. max
  **type** *Rok* = 1900 .. 1999
  Využití redundantní informace skýtané typem interval pro detekci chyb je jednou ze základních vlastností některých vyšších programovacích jazyků.

- **Typ definovaný výčtem (jednoduchý, ordinální typ)**
  V jazyce C tento typ není zaveden (datový typ **enum** nelze považovat za plnohodnotný datový typ).
  **type** *den* = ($C_1$,$C_2$,.... $C_n$), kde Ci je vždy identifikátor i-té hodnoty typu, n je kardinalita typu.

[1] Operací **round** zde rozumíme aritmetické zaokrouhlení: round(2.6) = 3. Operace **trunc** znamená oříznutí desetinné části čísla: trunc(2.6) = 2.

Objektem $C_i$ nikdy nemůže být číslo, znak či textový řetězec.
*Příklad:* **type** *den* = (*pondeli,utery,streda,ctvrtek,patek,sobota,nedele*)

Operace pořadí (ord), v některých programovacích jazycích lze vnutit vlastní pořadí.
ord :{ $i_1, i_2, \ldots i_n$ } → { 0,1, ... , n-1 }
ord($i_k$) = k-1, pro k = 1, .... n
Predikát ekvivalence a nonekvivalence: (=, ≠),
Predikáty uspořádání: (<, >, ≤, ≥ ).
Operace přiřazení: (=).

*Příklad*: následník (*utery*) je *streda* pořadí (*pondeli*) je 0
předchůdce (*patek*) je *ctvrtek*

*Poznámka*: uspořádání hodnot některých typů nemusí mít v reálném prostředí žádný význam. V tom případě nezáleží na pořadí, v jakém uvedeme jednotlivé hodnoty v definici typu.
**type** stav = (*pevny,tekuty,plynny*)

Strukturované datové typy budou probrány později.

### 3.3.6 Typový systém programovacího jazyka

***Typovým systémem*** jazyka rozumíme soubor pravidel, která přiřazují výrazům v daném jazyce typ. Máme-li například výraz 2 + 3, a budeme mluvit o jazyce C, typový systém přidělí tomuto výrazu nejspíše typ **int**. Výrazu 2 * (3.14 - 2) přidělí nejspíše typ **double**. Pro výraz 7 + "ABCD" však typový systém typ nenajde[2].

Říkáme proto, že typový systém akceptuje výraz, pokud pro něj nalezne typ a zamítne výraz, pokud k němu typ nenalezne. Výrazy zamítnuté typovým systémem představují chybu a nemohou být vyhodnoceny. Samotný proces přiřazování typu výrazům nazýváme též ***typovou kontrolou výrazů***. Typová kontrola má značný význam z hlediska praxe programátora, neboť umožňuje odhalit značné množství programátorských chyb. Ve zkratce lze říci, že typovou kontrolou zjišťujeme, zda typy jednotlivých operandů jsou slučitelné s typy jednotlivých operátorů.

Rozeznáváme dva druhy typových systémů: ***typový systém silný*** a ***slabý***. Typový systém nazveme silným, pokud akceptuje pouze bezpečné výrazy. Bezpečné výrazy jsou přitom takové, které nemohou za běhu programu způsobit chybu. Typový systém, který není silný, se nazývá slabý. Pro praktické použití je však tato definice příliš striktní, protože například překladač jazyka C nedokáže odhalit některé výrazy, které způsobují chyby, přesto se považuje za jazyk se silným typovým systémem. Praktičtější bude rozlišovat tyto systémy podle smyslu jejich existence: silný typový systém slouží k odhalování chyb v programu, kdežto slabý typový systém má za úkol odhalit pouze technicky neproveditelné nebo logicky nesmyslné výrazy (např. násobení čísla textovým řetězcem).

Oba systémy mají své výhody a nevýhody. Slabý typový systém může dovolit provedení výrazů, které nejsou bezpečné a hrozí tak nouzové zastavení programu chybou za běhu výpočtu (nebo ještě hůře – počítání s chybnou hodnotou bez jakékoli výstrahy). Jedná-li se o program řídící provoz jaderné elektrárny, může mít takováto chyba dosti vážné následky. Silný typový systém tento problém nemá, na druhé straně zase zakazuje provedení některých výrazů, které jsou bezpečné. Extrémním případem silného typového systému by byl systém, zamítající všechny výrazy. Takovýto systém by však nebyl příliš užitečný.

[2] Zde hovoříme o „klasických“ programovacích jazycích typu C, C++, Pascal, Java, atd. V některých speciálních jazycích může typový systém přijmout i takovýto výraz. Většinou jde o jazyky interpretované (či přímo skriptovací) s velmi uvolněným typovým systémem (Perl, Bash).

## *3.4 Prvky programovacích jazyků (lexikální jednotky)*

V této kapitole se budeme zabývat lexikálními prvky jazyka C. Většina obecných pojmů jako atom jazyka, identifikátor, výraz, atd. má ovšem stejný význam i v jiných programovacích jazycích. Znalost těchto pojmů se tedy bude hodit i v případě, že místo jazyka C budeme v praxi používat jiný programovací jazyk.

### 3.4.1 Základní elementy jazyka

Znaky, které tvoří program v jazyce C, se spojují do takzvaných lexikálních atomů. Překladač vždy sestavuje z přicházejících znaků zleva doprava nejdelší možný atom, i když výsledek nemusí tvořit platný program v jazyce C:

*Příklad*:

| **Znaky** | **Atomy jazyka** |
|---|---|
| forwhile | forwhile |
| b>x | b,>,x |
| b->x | b,->,x |
| b--x | b,--,x |
| b---x | b,--,-,x |

Existuje 5 tříd atomů:

- rezervované slovo
- identifikátor
- konstanta
- řetězcové literály
- oddělovač

Některé sousedící atomy mohou být od sebe odděleny libovolným počtem „bílých znaků“, tedy i nulovým (a + b, a+b). Gramatika jazyka předepisuje, že mezi některými atomy musí být minimálně jeden „bílý znak“ a to v případě, že by tyto atomy pak nešlo rozlišit například od identifikátoru (**do if**, *doif*).

***Bílý znak*** je jeden z následujících symbolů: mezera, tabelátor, nový řádek, posun řádku (LF), návrat vozíku (CR), nová stránka a vertikální tabelátor. Bílé znaky spolu s operátory a oddělovači stojí mezi identifikátory, klíčovými slovy, řetězci a konstantami použitými ve zdrojovém textu programu. Překladač považuje rovněž komentář za bílý znak (nahrazuje ho mezerou).

**Použitá notace:**
Syntaktické kategorie (nonterminály) jsou indikovány *kurzívou.*
Základní symboly (terminály) jsou označeny **tučně.**
Dvojtečka (:), která následuje nonterminál uvádí jeho definici.
Alternativní definice jsou uvedeny na novém řádku, některým předchází slova „one of“.
Volitelný symbol je indikován „opt“.
*Příklad:* **{** $expression_{opt}$ **}** znamená volitelný expression uzavřený ve složených závorkách.

*Poznámka**:***
V následujícím textu budou prvky programovacích jazyků demonstrovány na gramatice jazyka C. Pokud nepůjde o konstrukci jazyka C, bude to zvlášť zdůrazněno.
Jazyk C, jako každý průmyslově používaný jazyk, podléhá snahám o standardizaci. Protože se standard průběžně vyvíjí, je zvykem označovat nové vlastnosti jazyka zkratkou té verze normy, která tyto novinky zavedla. Původní verze jazyka se označuje podle jeho tvůrců K&R (Kernighan a Ritchie). Tato verze se v současné době nepoužívá. Významnou úpravu jazyka přinesla norma ANSI C (v podstatě totéž co ISO C90), která je dnes nejvíce používaná. Další rozšíření přinesla norma ISO C99. Dá se říci, že ISO C99 je nadmnožinou ANSI C. Původní verze K&R není s těmito verzemi plně kompatibilní. Vlastnosti jazyka, které v dalším textu nejsou nijak speciálně označeny vyhovují normě ANSI C, vyznačeny budou novinky v ISO C99.

NV-C99 - označuje novou vlastnost ve verzi ISO C99 [C99]

**Lexikální elementy**

- Rezervovaná (klíčová) slova
- Identifikátory
- Konstanty
- Řetězcové literály
- Oddělovače (interpunkce)

*token:*
*keyword*
*identifier*
*constant*
*string-literal*
*punctuator*

*preprocessing-token:*
*header-name*
*identifier*
*pp-number*
*character-constant*
*string-literal*
*punctuator*

### 3.4.2 Rezervovaná slova

***Rezervovaná slova*** jsou vyhrazené identifikátory. Nelze uživatelsky definovat identifikátor shodný s nějakým rezervovaným slovem. [C99]

*keyword:* one of

| | | | |
|---|---|---|---|
| **auto** | **enum** | **restrict** | **unsigned** |
| **break** | **extern** | **return** | **void** |
| **case** | **float** | **short** | **volatile** |
| **char** | **for** | **signed** | **while** |
| **const** | **goto** | **sizeof** | **_Bool** |
| **continue** | **if** | **static** | **_Complex** |
| **default** | **inline** | **struct** | **_Imaginary** |
| **do** | **int** | **switch** | |
| **double** | **long** | **typedef** | |
| **else** | **register** | **union** | |

Jde o elementy definované přímo jako součást gramatiky jazyka. Uživatel nemá možnost je měnit – jsou zakomponována přímo do překladače. Různé implementace mohou tento seznam dále rozšiřovat.

### 3.4.3 Identifikátory

Identifikátor je jednoznačný název (pojmenování) používaný v programovacím jazyce k označení podprogramů, proměnných, datových typů, konstant, atd. Identifikátor může být libovolně dlouhý, ale dle konkrétní implementace se rozlišuje pouze prvních n (např. 31) znaků, (ISO C99 – 63 znaků) [C99]. Identifikátory musí být souvislé. Nelze je přerušovat bílými znaky.

*identifier:*
*identifier-nondigit*
*identifier identifier-nondigit*
*identifier digit*

*identifier-nondigit:*
*nondigit*
*universal-character-name*

*nondigit:* one of
**_ a b c d e f g h i j k l m**
**n o p q r s t u v w x y z**
**A B C D E F G H I J K L M**
**N O P Q R S T U V W X Y Z**

*digit:* one of
**0 1 2 3 4 5 6 7 8 9**

NV-C99
*universal-character-name:*
**\u** *hex-quad*
**\U** *hex-quad hex-quad*

*hex-quad:*
*hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit*
*Poznámka:*
Pro jednotlivé jazyky existuje tabulka povolených univerzálních znaků v identifikátoru.

Jazyk C je tzv. ***case sensitive*** jazyk, to znamená, že rozlišuje velká a malá písmena.
*Příklad: prom Prom PROM* jsou tři různé identifikátory.

Znak _ (podtržítko) není vhodné používat libovolně:
*_prom* - raději nepoužívat, takto bývají tvořeny systémové identifikátory,
*prom_x* - používá se poměrně často, zpřehledňuje text,
*prom_* - nepoužívat, protože se na konci snadno přehlédne.

*Příklad:*
**int** cislo;
| ^-- identifikátor
^-------- klíčové slovo

**float** sectiMatice(Tmatice *m1, Tmatice *m2)
| ^------------^--------^-------^--------^--------identifikátory
^----- klíčové slovo

Zásady:

- Jména objektů (proměnných, funkcí) psát malými písmeny s využitím znaku podtržení nebo kombinace malých a velkých písmen. Není však vhodné oba způsoby kombinovat v jednom programu. Programátor by si měl vybrat jeden styl a ten potom konzistentně používat.

  *Příklad: velmi_dlouhy_identifikator velmiDlouhyIdentifikator*

- Nepoužívat podobné identifikátory, např. *systst* a *sysstst*.

- Nepoužívat dva stejné identifikátory odlišené jen typem písma, např. *PROM* a *prom*.

## 3.4.4 Konstanty (literály)

***Konstanty*** jsou symboly, reprezentující neměnnou číselnou nebo jinou hodnotu. Překladač jazyka jim přiřadí typ, který této hodnotě odpovídá.
*constant:*
*integer-constant*
*floating-constant*
*enumeration-constant*
*character-constant*

### 3.4.4.1 Celočíselné konstanty

Celočíselná konstanta se zapisuje v desítkové, osmičkové nebo šestnáctkové soustavě.

**Syntaktická definice celočíselné konstanty**
*integer-constant:*
*decimal-constant integer-suffix*$_{opt}$
*octal-constant integer-suffix*$_{opt}$
*hexadecimal-constant integer-suffix*$_{opt}$

*decimal-constant:*
*nonzero-digit*
*decimal-constant digit*

*octal-constant:*
    **`0`**
    *octal-constant octal-digit*

*hexadecimal-constant:*
    *hexadecimal-prefix hexadecimal-digit*
    *hexadecimal-constant hexadecimal-digit*

*hexadecimal-prefix:* one of
    **`0x 0X`**

*nonzero-digit:* one of
    **`1 2 3 4 5 6 7 8 9`**

*octal-digit:* one of
    **`0 1 2 3 4 5 6 7`**

*hexadecimal-digit:* one of
    **`0 1 2 3 4 5 6 7 8 9`**
    **`a b c d e f`**
    **`A B C D E F`**

*integer-suffix:*
    *unsigned-suffix long-suffix*$_{opt}$
    *unsigned-suffix long-long-suffix*
    *long-suffix unsigned-suffix*$_{opt}$
    *long-long-suffix unsigned-suffix*$_{opt}$

*unsigned-suffix:* one of
    **`u U`**

*long-suffix:* one of
    **`l L`**

*long-long-suffix:* one of
    **`ll LL`**

- Začíná-li celočíselná konstanta znaky **`0x`** nebo **`0X`**, pak se jedná o zápis v šestnáctkové soustavě se znaky **`a`** až **`f`** (**`A`** až **`F`**) ve významu hodnot 10 až 15.
- Začíná-li číslicí **`0`**, pak se jedná o zápis v osmičkové soustavě. (Pozor! Člověk snadno přehlédne, že nejde o desítkové číslo.)
- Jinak se jedná o zápis v desítkové soustavě.
- Za celočíselnou konstantou může bezprostředně následovat jedno z písmen **`l`** nebo **`L`**, které označuje konstantu typu **long**. ANSI C dovoluje použití písmena **`u`** nebo **`U`** pro označení konstanty bez znaménka.

*Příklady:*

`123  255  -45` – desítkové konstanty

`0173  0377  -055` – osmičkové konstanty

`0x7b 0XFF -0x2D` – hexadecimální konstanty

`128U 256u` – konstanta bez znaménka

`128L 256LL` – **long, long long**

`56 LU 56ull` – (!)

*Zásady:*
Nepoužívat přípony malé `l` a `ll` – jsou snadno zaměnitelné s číslicí 1 nebo 11. Použití `L` nebo `LL` je přehlednější.

### 3.4.4.2 Racionální konstanty (float, double, ...)

***Racionální konstanty*** umožňují zapsat číselnou konstantu, která nemusí být celočíselná. Vnitřně je reprezentována ve tvaru, který obsahuje mantisu a exponent, obojí s případným znaménkem. Někdy se těmto číslům říká čísla s pohyblivou řádovou čárkou nebo nepřesně reálná čísla (i když matematicky to neodpovídá).

Racionální konstanta se zapisuje s desetinnou tečkou, s exponentem nebo oběma způsoby současně. Racionální konstanta se interpretuje v desítkové soustavě pokud není uvedeno jinak. ANSI C dovoluje koncové písmeno pro rozlišení konstanty typu **float** a **long double**. Neuvede-li se přípona, typ konstanty je **double**.

**Syntaktická definice racionální konstanty**

*floating-constant:*
    *decimal-floating-constant*
    *hexadecimal-floating-constant*

*decimal-floating-constant:*
    *fractional-constant exponent-part*$_{opt}$ *floating-suffix*$_{opt}$
    *digit-sequence exponent-part floating-suffix*$_{opt}$

*hexadecimal-floating-constant:*
    *hexadecimal-prefix hexadecimal-fractional-constant*
    *binary-exponent-part floating-suffix*$_{opt}$
    *hexadecimal-prefix hexadecimal-digit-sequence*
    *binary-exponent-part floating-suffix*$_{opt}$

*fractional-constant:*
    *digit-sequence*$_{opt}$ **.** *digit-sequence*
    *digit-sequence* **.**

*exponent-part:*
    **e** *sign*$_{opt}$ *digit-sequence*
    **E** *sign*$_{opt}$ *digit-sequence*

*sign:* one of
    **+ -**

*digit-sequence:*
*digit*
*digit-sequence digit*

*hexadecimal-fractional-constant:*
*hexadecimal-digit-sequence*$_{opt}$ **.**
*hexadecimal-digit-sequence*
*hexadecimal-digit-sequence* **.**

*binary-exponent-part:*
**p** *sign*$_{opt}$ *digit-sequence*
**P** *sign*$_{opt}$ *digit-sequence*

*hexadecimal-digit-sequence:*
*hexadecimal-digit*
*hexadecimal-digit-sequence hexadecimal-digit*

*floating-suffix:* one of
**f l F L**

*Příklad:*
`0.  .0  1.0  1.0f  3e1  1.0E-3  .00034  1.0e67L`
(`0xFF.Fp8` – specialita, hexadecimální zápis desetinné konstanty)

*Zásady:*
Při porovnávání proměnné typu **float** s konstantou je lepší používat desetinné konstanty:
`if (determinant < 0.0)`

### 3.4.4.3 Znakové konstanty

***Znaková konstanta*** obsahuje jeden znak, přičemž znakem rozumíme jeden prvek z používané znakové sady. Nejjednodušším případem znakových konstant jsou ty, které obsahují některý ze zobrazitelných znaků, mimo znaků apostrof (**'**) a obrácené lomítko (\) . Znaky apostrof, obrácené lomítko a tzv. „neviditelné znaky“ zapisujeme pomocí jejich znakových kódů (nejčastěji kódů v tabulce ASCII). Některé často používané znaky (zpětné lomítko, apostrof, nový řádek, …) mají předdefinovaný zápis pomocí zpětného lomítka ('\\', '\'', '\n', ...)

Konstanty uvozené znakem \ se také nazývají ***escape sekvence***, protože jejich zápis je uvozen zpětným lomítkem (escape character). Některé často používané escape sekvence mají kromě numerických kódů i znakový ekvivalent.

**Syntaktická definice znakové konstanty**
*character-constant:*
**'** *c-char-sequence* **'**
**L'** *c-char-sequence* **'**

*c-char-sequence:*
*c-char*
*c-char-sequence c-char*

*c-char:*
any member of the source character set except
the single-quote **'**, backslash ****, or new-line character
*escape-sequence*

*escape-sequence:*
*simple-escape-sequence*
*octal-escape-sequence*
*hexadecimal-escape-sequence*
*universal-character-name*

*simple-escape-sequence:* one of
**\' \" \? **
**\a \b \f \n \r \t \v**

*octal-escape-sequence:*
**** *octal-digit*
**** *octal-digit octal-digit*
**** *octal-digit octal-digit octal-digit*

*hexadecimal-escape-sequence:*
**\x** *hexadecimal-digit*
*hexadecimal-escape-sequence hexadecimal-digit*

*Příklad:*

| | |
|---|---|
| `'J'` | znak J |
| `'\112'  '\12'  '\15'` | jde o osmičkové hodnoty! |
| `'\x4A'  '\x4a'` | šestnáctkové hodnoty |
| `'\X4A'` | nelze, správně: `'\x4A'` |

*Pozor:* Neplést s řetězcem – `'A'` není totéž co `"A"`

### 3.4.5 Řetězcové literály

Pod pojmem ***řetězec*** se v programovacích jazycích myslí posloupnost znaků. V jazyce C se řetězce uzavírají do dvojice uvozovek ("Toto je ukázka řetězcové konstanty v jazyce C "). V řetězcových konstantách lze používat češtinu a znaky s diakritickými znaménky (pozor ale na nastavení jazykového prostředí v operačním systému, např. Windows používají pro češtinu zároveň dvě znakové sady Win1250 a cp852.

**Syntaktická definice řetězce**
*string-literal:*
**"** *s-char-sequence*$_{opt}$ **"**
**L"** *s-char-sequence*$_{opt}$ **"**

*s-char-sequence:*
*s-char*
*s-char-sequence s-char*

*s-char:*
any member of the source character set except
the double-quote **"**, backslash ****, or new-line character
*escape-sequence*

Velmi často se řetězcová konstanta nevejde na jednu řádku zdrojového kódu. V takovém případě ji můžeme rozdělit na více řádků:

*Příklad:*
```
"Některé řetězce se těžko vejdou na jednu řádku"
"Některé řetězce se těžko\
 vejdou na jednu řádku"
"Některé řetězce se těžko"
 "vejdou na jednu řádku"
"Některé řetězce se těžko"          "vejdou na"
 "jednu řádku"
"Řetězec může obsahovat i více řádků, které \n
jsou odděleny znakem nového řádku.\n"
"Je přehlednější to psát takto.\n"
```

Jazyk C nemá žádný standardní datový typ pro přímé uložení řetězce. Je nutné to řešit pomocí pole. Řetězec uložený v paměti zabírá vždy o jeden bajt více než je počet jeho znaků, protože na konec řetězce je (automaticky) připojen nulový znak (`'\0'`).

### 3.4.6 Pojmenované konstanty

Konstanty můžeme pojmenovat. Konstantám s vhodně zvolenými identifikátory dáváme přednost například před přímým uvedením konstantní hodnoty jako meze cyklu nebo dimenze pole. Modifikace programu pak probíhá velmi snadno změnou hodnoty konstanty. Odpadá obtížné uvažování, zda-li ta či jiná hodnota má být modifikována, či nikoliv. Pro tyto účely slouží tzv. ***pojmenované konstanty***.

Většina běžných jazyků má jednotný způsob, jak vytvářet pojmenované konstanty. Typicky se to dělá pomocí klíčového slova **const**. V jazyce C, vlivem historického vývoje, lze tyto konstanty vytvářet trojím způsobem. Prvním způsobem je definovat konstantu pomocí příkazu preprocesoru *#define* **.**

```
#define RETEZEC "Konstantni retezec."  // zde se nepíše = ani ;
#define PI 3.141592653589793
```

Konstanty definované pomocí *#define* nemají specifikován datový typ. Jde pouze o textové zkratky, kdy preprocesor před překladem programu nahradí v textu všechna jména symbolických konstant jejich skutečnou hodnotou. Tento způsob definice konstant vlastně není součástí gramatiky jazyka C, protože je zpracovává externí program – preprocesor. Jde vlastně o tzv. ***metapříkazy***, které samy o sobě s jazykem nesouvisí.

Další způsob jak definovat konstanty je použití výčtového datového typu **enum**. Definice konstant pomocí datového typu **enum** se používá v případech, kdy spolu mají konstanty nějakým způsobem souviset. Používá se například pro specifikaci chybových kódů:

```
enum ErrFileCodes {E_OK, E_BADFILE, E_READ, E_WRITE};
enum ErrValues {EV_NEGATIVE = -1, EV_TOO_BIG = 1024};
```

Všechny takto definované konstanty jsou typu **int**. Při definici jim lze přidělit hodnotu. Pokud žádnou hodnotu přidělenu nemají, mají hodnotu od nuly a podle pořadí definic každá další o jedničku větší (*E_OK == 0, E_BADFILE == 1, ...*)

Třetím typem konstant jsou ***konstantní proměnné*** (od ISO C90). Definujeme je pomocí klíčového slova **const** následovaného specifikací datového typu, jménem a inicializací. Nejde ovšem o konstanty v pravém slova smyslu, překladač například nedovolí její použití při definici pole na globální úrovni. Jde o proměnné označené modifikátorem **const**, který říká překladači, že má hlídat všechny pokusy o modifikaci. Příklady konstantních proměnných:

```
const int KONSTANTA = 123;
const float PLANCK = 6.6256e-34F; // bez F na konci dojde k double => float
const char MALE_A = 'a';
const char *RETEZEC = "Konstantni retezec."
const char RETEZEC[] = "Konstantni retezec." // lepší
```

Takto můžeme definovat konstantní proměnné všech základních datových typů. Výhodou je, že takto vytvořené konstanty mají specifikován datový typ, proto překladač může při jejich použití uplatnit typovou kontrolu. Nevýhodou je jejich omezená použitelnost.

*Zásady:*

- Nepoužívat tzv. „magické konstanty“ - nepojmenované konstanty uvnitř kódu. Přispívají ke vzniku chyb a nepřehlednosti kódu při opravách. Místo nich je lepší používat pojmenované konstanty (*#define* nebo konstantní proměnné).
- Názvy konstant se píší vždy velkými písmeny, aby je bylo možné v kódu odlišit od běžných identifikátorů.
- Kde to jde používat konstantní proměnné, pro významově spřízněné celočíselné konstanty používat výčtový typ, v ostatních případech *#define* konstanty.

### 3.4.7 Interpunkce (oddělovače)

Interpunkcí se v jazyce C myslí operátory. Blíže viz kapitolu Operátory a výrazy.

**Syntaktická definice oddělovačů** [C99]
*punctuator:* one of

```
[ ] ( ) { } . ->
++ -- & * + - ~ !
/ % << >> < > <= >= == != ^ | && ||
? : ; ...
= *= /= %= += -= <<= >>= &= ^= |=
, # ##
<: :> <% %> %: %:%:
```

Norma myslí i na uživatele, kteří mají speciální klávesnice, kde chybí znaky jako [,],{,}, apod. Pro tyto účely je možné využívat kombinace <:, :>, <%. %>. Obecně se ale nedoporučuje je používat, protože to je pro většinu programátorů velmi nezvyklé a takový kód je potom prakticky nečitelný.

### 3.4.8 Komentáře

***Komentář*** je text, který můžeme napsat kamkoliv do zdrojového kódu a nemá žádný vliv na překlad. Nejčastěji se používá kvůli vysvětlení určité části kódu. Velmi doporučujeme je vhodně používat. Patří to k dobré programátorské kultuře.

Komentář se může vyskytnout všude tam, kde je povolen bílý znak.

```
/* Ukázka komentáře */
// komentář do konce řádku (podle nové normy)
```

Není možné vnořovat komentáře /* */:

```
/* Ukázka komentáře, v němž je vnořen /*další komentář*/ */
```
!! nelze

```
/*  Ale toto lze: // pořád uvnitř komentáře */
```
tady už je kód - lze

```
// /* v komentáři */  pořád v komentáři
// komentář do konce řádku ať je v něm cokoli /* */ //
```
-- lze ...

Existují systémy, které umí z komentářů ve zdrojovém souboru vygenerovat hypertextovou programátorskou dokumentaci (např. Doxygen pro C, C++, javadoc pro Javu). Proto se vyplatí psát dostatečně kvalitní komentáře už v průběhu psaní kódu.

*Doporučení:*

- Psát komentáře průběžně při psaní zdrojového kódu a ne až nakonec až zbude čas.
- Napsat komentář hned na začátek každého zdrojového souboru a uvést název programu, jméno autora, datum a verzi.
- Vytvářet funkce od úvodního komentáře – ujasníte si, co od vytvářené funkce očekáváte.

```
/*********************************************************/
/* * *       Práce s maticemi v jazyce C               * * */
/* * *                                                 * * */
/* * *               Verze:1                           * * */
/* * *                                                 * * */
/* * *            Jaroslav  Nový                       * * */
/* * *              říjen 2018                         * * */
/*********************************************************/
```

Styl vhodný pro zpracování dokumentačním systémem Doxygen:

```
/**
 * Funkce počítá součet dvou matic stejných rozměrů.
 * Výsledek ukládá do …
 */
```

## 3.5 Proměnné a deklarace

Při programování potřebujeme uchovávat data. Mohou to být:

- Údaje zadané uživatelem (jeho jméno, věk, adresa, posloupnost čísel, kterou chce uživatel seřadit, …).
- Údaje načtené z diskového souboru (textové, číselné, …).
- Různé dočasné a pomocné hodnoty (počet opakování).
- Operandy, výsledky operací.

***Datový objekt*** je obecné označení jakéhokoliv údaje uloženého v operační paměti (nezaměňovat s pojmem objekt v objektově orientovaném programování).
Paměťová místa, ve kterých uchováváme data, označujeme jako ***proměnné***. Proměnná je datový objekt určitého typu, hodnota tohoto objektu se může (za běhu programu měnit).
***Deklarace proměnné*** je konstrukce, která přidělí proměnné jméno a typ, ale nevytvoří ji. Najdeme je někdy v hlavičkových souborech, i když jejich použití tímto způsobem je neobvyklé.
***Definice proměnné*** je v jazyce C zároveň deklarací a kromě jména a datového typu přidělí proměnné i paměťový prostor.

*Příklady:*
```
int i, j;
char znak;
float podil,odmocnina;
```

**Inicializace proměnné**
Velmi často už při definování proměnné víme, jakou bude mít zpočátku hodnotu. V tom případě je vhodné proměnnou ***inicializovat***. Vyhýbáme se tím nepříjemným chybám, protože proměnné mohou mít po definici bez inicializace náhodnou hodnotu.

*Příklady:*
```
char znak = 'a';
int rozmer = 100;
```

### 3.5.1 Syntaktická definice deklarace

*declaration:*
*declaration-specifiers init-declarator-list*$_{opt}$ **;**

*declaration-specifiers:*
*storage-class-specifier declaration-specifiers*$_{opt}$
*type-specifier declaration-specifiers*$_{opt}$
*type-qualifier declaration-specifiers*$_{opt}$
*function-specifier declaration-specifiers*$_{opt}$

*init-declarator-list:*
*init-declarator*
*init-declarator-list* **,** *init-declarator*

*init-declarator:*
*declarator*
*declarator* **=** *initializer*

*storage-class-specifier:*
**typedef**
**extern**
**static**
**auto**
**register**

*type-specifier:*
**void**
**char**
**short**
**int**
**long**
**float**
**double**
**signed**
**unsigned**
**_Bool**
**_Complex**
**_Imaginary**

*struct-or-union-specifier*
*enum-specifier*
*typedef-name*

*struct-or-union-specifier:*
*struct-or-union identifier*$_{opt}$ **{** *struct-declaration-list* **}**
*struct-or-union identifier*

*struct-or-union:*
**struct**
**union**

*struct-declaration-list:*
*struct-declaration*
*struct-declaration-list struct-declaration*

*struct-declaration:*
*specifier-qualifier-list struct-declarator-list* **;**

*specifier-qualifier-list:*
*type-specifier specifier-qualifier-list*$_{opt}$
*type-qualifier specifier-qualifier-list*$_{opt}$

*struct-declarator-list:*
*struct-declarator*
*struct-declarator-list* **,** *struct-declarator*

*struct-declarator:*
*declarator*
*declarator*$_{opt}$ **:** *constant-expression*

*enum-specifier:*
**enum** *identifier*$_{opt}$ **{** *enumerator-list* **}**
**enum** *identifier*$_{opt}$ **{** *enumerator-list* **,** **}**
**enum** *identifier*

*enumerator-list:*
*enumerator*
*enumerator-list* **,** *enumerator*

*enumerator:*
*enumeration-constant*
*enumeration-constant* **=** *constant-expression*

*type-qualifier:*
**const**
**restrict**
**volatile**

*function-specifier:*
**inline**

*declarator:*
*pointer*$_{opt}$ *direct-declarator*

*direct-declarator:*
*identifier*
**(** *declarator* **)**
*direct-declarator* **[** *type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ **]**
*direct-declarator* **[** **static** *type-qualifier-list*$_{opt}$ *assignment-expression* **]**
*direct-declarator* **[** *type-qualifier-list* **static** *assignment-expression* **]**
*direct-declarator* **[** *type-qualifier-list*$_{opt}$ ***** **]**
*direct-declarator* **(** *parameter-type-list* **)**
*direct-declarator* **(** *identifier-list*$_{opt}$ **)**

*pointer:*
***** *type-qualifier-list*$_{opt}$
***** *type-qualifier-list*$_{opt}$ *pointer*

*type-qualifier-list:*
*type-qualifier*
*type-qualifier-list type-qualifier*

*parameter-type-list:*
*parameter-list*
*parameter-list* **, ...**

*parameter-list:*
*parameter-declaration*
*parameter-list* **,** *parameter-declaration*

*parameter-declaration:*
*declaration-specifiers declarator*
*declaration-specifiers abstract-declarator*$_{opt}$

*identifier-list:*
*identifier*
*identifier-list* **,** *identifier*

*type-name:*
*specifier-qualifier-list abstract-declarator*$_{opt}$

*abstract-declarator:*
*pointer*
*pointer*$_{opt}$ *direct-abstract-declarator*

*direct-abstract-declarator:*
**(** *abstract-declarator* **)**
*direct-abstract-declarator*$_{opt}$ **[** *assignment-expression*$_{opt}$ **]**
*direct-abstract-declarator*$_{opt}$ **[** ***** **]**
*direct-abstract-declarator*$_{opt}$ **(** *parameter-type-list*$_{opt}$ **)**

*typedef-name:*
    *identifier*

*initializer:*
    *assignment-expression*
    **{** *initializer-list* **}**
    **{** *initializer-list* **,** **}**

*initializer-list:*
    *designation*$_{opt}$ *initializer*
    *initializer-list* **,** *designation*$_{opt}$ *initializer*

*designation:*
    *designator-list* **=**

*designator-list:*
    *designator*
    *designator-list designator*

*designator:*
    **[** *constant-expression* **]**
    **.** *identifier*

Jednotlivé typy deklarací budou průběžně vysvětleny.

## 3.6 Operátory a výrazy

Tyto pojmy jsou dostatečně známy z matematiky (ostatně počítač je matematický stroj). V programovacích jazycích se tyto pojmy většinou zobecňují na všechny dostupné datové typy. Dále tedy budeme tyto pojmy chápat takto:
***Výraz*** – konstrukce jazyka, která má hodnotu (nějakého datového typu).
***Operand*** – je částí výrazu na kterou je aplikován jeden z legálních operátorů. Má také hodnotu konkrétního datového typu. Někdy se operandům říká podvýrazy, protože operandem může být i složitější výraz, např. `(a+b)*(c+d)`.
***Operátor*** – určuje, jakým způsobem se z operandů získá hodnota výrazu. Pořadí vyhodnocování podvýrazů ve výrazu je dáno prioritami jednotlivých operátorů nebo závorkami, pokud byly použity.

Datové typy operandů určují, které operátory je možné v daném okamžiku použít. Na konkrétním typovém systému jazyka záleží, zda se aplikace operátoru na různé datové typy vyhodnotí jako chybná konstrukce nebo zda je možno tímto spojením zkonstruovat smysluplnou hodnotu.

*Příklad:*

```
    x = a + b * c;
operátor: --------^------^
podvýraz: ---^
podvýraz: ----------^^^^^^
výraz: -------^^^^^^^^^^^
```

Jazyk C poskytuje operátory jen pro zabudované datové typy. Pokud chce uživatel provádět operace nad vlastními datovými typy, musí použít vlastní funkce. Některé jazyky (C++) umožňují předefinování sémantiky základních operátorů pro uživatelské typy (tato konstrukce je ovšem

problematická z pohledu bezpečnosti a přehlednosti programování, protože umožňuje vytvářet zákeřné vedlejší efekty).

**Výraz × příkaz**
Rozdíl mezi výrazem a příkazem je v tom, že výraz představuje dále použitelnou hodnotu, kdežto primárním úkolem příkazu je vykonat nějaký kód. Pokud je výsledkem kódu v příkazu nějaká hodnota a nejde o výraz s přiřazením, hodnota se zahodí (nepoužije, bude se ignorovat).
V jazyce C se z výrazu stane příkaz tím, že jej ukončíme středníkem (samozřejmě v místě, kde to dává smysl; uprostřed podmínky cyklu to samozřejmě nejde). Samotný středník (;) představuje ***prázdný příkaz*** (null statement), který se někdy používá například v cyklech.

*Příklad:*
```
x = 64 // výraz s přiřazením
x = 64; // příkaz
printf("Hello world!"); // příkaz
getchar(); // příkaz – výsledek funkce se „zahodí"/nepoužije
a + b; // příkaz – výsledek výrazu se „zahodí"/nepoužije – tzv. mrtvý kód,
       // programátor by se mu měl vyhýbat
```

### 3.6.1 Operátor přiřazení, L-hodnota a P-hodnota

Operátor přiřazení (=) kopíruje hodnotu výrazu na své pravé straně do proměnné na levé straně. V této souvislosti se mluví o L-hodnotě a P-hodnotě (v anglických textech L-value a R-value).
***L-hodnota*** je objekt v paměti, kterému lze přiřadit hodnotu. V praxi může být L-hodnotou proměnná, prvek pole nebo paměť odkazovaná přes ukazatel.
***P-hodnota*** je výraz, který má vždy hodnotu a který vystupuje na pravé straně přiřazovacího operátoru.
*Příklad:*
```
float x = -c/b;
pole[i] = 25;
*cislo = 4;
souradnice->x = 17;

(a + b) = 68; // nelze!
faktorial(5) = 120; // nelze!
```

**Přiřazení**
V jazyce C je přiřazení definováno jako operátor (=). V praxi to znamená, že operace přiřazení má vlastní hodnotu, kterou lze použít dále ve složitějším výrazu. Operátor přiřazení se tedy může vyskytovat i v P-výrazech.
*Příklad:*
```
x1 = x2 = -c/b;
```

*Zásada:*
V praxi používat opatrně, raději jen v obecně zažitých konstrukcích (např. čtení ze souboru pomocí cyklu `while((znak = fgetc()) != EOF))`.

Jazyk C ještě poskytuje celou kolekci rozšířených přiřazovacích operátorů, které zjednodušují zápis často používaných výrazů typu:
*L-hodnota = L-hodnota operátor výraz;*

Místo toho lze napsat:
*L-hodnota operátor= výraz;*

V jazyce C jsou definovány tyto rozšířené přiřazovací operátory:
+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=

*Příklad:*

```
x *= a + b; je totéž jako x = x * (a + b); // závorky jsou nutné!
x * = a + b;  // syntaktická chyba – mezi operátorem a rovnítkem
              // (=) nesmí být mezera
```

**Syntaktická definice výrazu** [C99]

*primary-expression:*
- *identifier*
- *constant*
- *string-literal*
- **(** *expression* **)**

*postfix-expression:*
- *primary-expression*
- *postfix-expression* **[** *expression* **]**
- *postfix-expression* **(** *argument-expression-list*$_{opt}$ **)**
- *postfix-expression* **.** *identifier*
- *postfix-expression* **->** *identifier*
- *postfix-expression* **++**
- *postfix-expression* **--**
- **(** *type-name* **)** **{** *initializer-list* **}**
- **(** *type-name* **)** **{** *initializer-list* **,** **}**

*argument-expression-list:*
- *assignment-expression*
- *argument-expression-list* **,** *assignment-expression*

*unary-expression:*
- *postfix-expression*
- **++** *unary-expression*
- **--** *unary-expression*
- *unary-operator cast-expression*
- **sizeof** *unary-expression*
- **sizeof** **(** *type-name* **)**

*unary-operator:* one of
- **& * + - ~ !**

*cast-expression:*
- *unary-expression*
- **(** *type-name* **)** *cast-expression*

*multiplicative-expression:*
- *cast-expression*
- *multiplicative-expression* ***** *cast-expression*
- *multiplicative-expression* **/** *cast-expression*
- *multiplicative-expression* **%** *cast-expression*

*additive-expression:*
- *multiplicative-expression*

*additive-expression* **+** *multiplicative-expression*
*additive-expression* **–** *multiplicative-expression*

*shift-expression:*
*additive-expression*
*shift-expression* **<<** *additive-expression*
*shift-expression* **>>** *additive-expression*

*relational-expression:*
*shift-expression*
*relational-expression* **<** *shift-expression*
*relational-expression* **>** *shift-expression*
*relational-expression* **<=** *shift-expression*
*relational-expression* **>=** *shift-expression*

*equality-expression:*
*relational-expression*
*equality-expression* **==** *relational-expression*
*equality-expression* **!=** *relational-expression*

*AND-expression:*
*equality-expression*
*AND-expression* **&** *equality-expression*

*exclusive-OR-expression:*
*AND-expression*
*exclusive-OR-expression* **^** *AND-expression*

*inclusive-OR-expression:*
*exclusive-OR-expression*
*inclusive-OR-expression* **|** *exclusive-OR-expression*

*logical-AND-expression:*
*inclusive-OR-expression*
*logical-AND-expression* **&&** *inclusive-OR-expression*
*logical-OR-expression:*
*logical-AND-expression*
*logical-OR-expression* **||** *logical-AND-expression*

*conditional-expression:*
*logical-OR-expression*
*logical-OR-expression* **?** *expression* **:** *conditional-expression*

*assignment-expression:*
*conditional-expression*
*unary-expression assignment-operator assignment-expression*

*assignment-operator:* one of
**= *= /= %= += -= <<= >>= &= ^= |=**

*expression:*

*assignment-expression*
*expression* , *assignment-expression*

*constant-expression:*
*conditional-expression*

## 3.6.2 Aritmetické operátory

Aritmetické operátory jsou v jazyce C definovány nad číselnými datovými typy. Můžeme je rozdělit na tři skupiny – ***unární*** a ***binární aritmetické operátory*** a na ***speciální unární operátory***.

[HePa13, str. 35]

**Unární operátory**

| Unární plus | Unární mínus |
|---|---|
| + | - |

Používají se běžným způsobem ke změně znaménka výrazu. Je možné je použít jak s celočíselnými tak i s racionálními typy.

**Binární aritmetické operátory**

| sčítání | odčítání | násobení | dělení | modulo |
|---|---|---|---|---|
| + | - | * | / | % |

**Modulo**
Operátor modulo (%) - zbytek po dělení je možné použít pouze s celočíselnými typy. Pozor! V jazyce C může operace modulo produkovat i záporné hodnoty `(-2%4 == -2)`.

**Dělení**
Operátor dělení funguje kontextově podle datového typu operandů:
**int** op **int** … celočíselné dělení – výsledkem je **int**
**int** op **float** … dělení v pohyblivé řádové čárce – výsledkem je **float**
**float** op **int** … dělení v pohyblivé řádové čárce – výslekem je **float**
**float** op **float** … dělení v pohyblivé řádové čárce – výsledkem je **float**

**Priorita** [HePa13, str. 270]
Multiplikativní operátory (*, /, %) mají vyšší prioritu než aditivní operátory (+, -), takže matematické výrazy můžeme zapisovat přirozeným způsobem:

*Příklad:*
```
x = 10 + 2*3; // výsledkem je 16
```

**Speciální unární operátory**
Jazyk C poskytuje dva speciální celočíselné unární operátory.

| Unární přičtení jedničky | Unární odečtení jedničky |
|---|---|
| ++ | -- |

Oba existují jako předpony (prefix) a jako přípony (suffix). Obě verze se liší svým významem:

L-hodnota++
Provede se inkrementace po použití hodnoty. Nejprve vrátí původní hodnotu (pokud je použit ve výrazu) a poté přičte k L-hodnotě jedničku.

++L-hodnota
Provede se inkrementace před použitím hodnoty. Nejprve k L-hodnotě přičte jedničku a teprve potom vrátí hodnotu výsledku (pokud je použit ve výrazu).

Tyto operátory lze použít výhradně na L-hodnotu, protože do ní ukládají výsledek.
*Příklady:*
```
28++    // nelze
--(a + b)   // nelze
i++; // inkrementace proměnné i
i = i++; // POZOR! Nedefinovaná hodnota – není jasné, v jakém pořadí se bude
         // modifikovat hodnota proměnné i.
```

### 3.6.3 Logické a relační operátory

***Logické a relační operátory*** pracují s logickými hodnotami. V jazyce C však produkují výsledky typu **int**, protože jazyk C interpretuje hodnotu 0 jako nepravdu jakoukoli nenulovou hodnotu jako pravdu. [HePa13, str. 48]

*Poznámka:* Norma ISO C99 přinesla do jazyka C datový typ **bool** s hodnotami **true** a **false**. Tento typ je však kompatibilní s datovým typem **int**. Pro práci s hodnotami **true** a **false** je nutné dovézt rozhraní <stdbool.h>. Je si však potřeba dávat pozor na to, že ne každá hodnota, která bude interpretována jako pravda je shodná s hodnotou **true**. Identifikátor **true** je totiž definován jako konstanta s hodnotou 1.

**Logické operátory**

| Logický součin | Logický součet | Logická negace |
|---|---|---|
| && | \|\| | ! |

**Relační operátory**

| Rovnost | Nerovnost | Menší než | Menší nebo rovno | Větší než | Větší nebo rovno |
|---|---|---|---|---|---|
| == | != | < | <= | > | >= |

**Zkrácené vyhodnocování** [HePa13, str. 48]
Jazyk C používá při vyhodnocování logického součinu a součtu ***zkrácené vyhodnocování*** (short circuit). Logické výrazy se vyhodnocují zleva doprava a jakmile je jisté, jaký bude mít výraz výsledek, vyhodnocování se ukončí.

*Příklad:*
V jazyce C je naprosto legální tato konstrukce: `if (x != 0 && y / x < z)`

Nikdy v ní nedojde k dělení nulou, protože v případě, že *x* je rovno nule, po vyhodnocení podvýrazu *x != 0* je jasný výsledek celého výrazu a druhá část už se nebude vyhodnocovat.
V jazyce C je priorita aritmetických a relačních operátorů vyšší než priorita logických operátorů, takže není potřeba logické výrazy přehnaně závorkovat.

### 3.6.4 Bitové operátory

Někdy je při programování potřeba pracovat s čísly na úrovni jednotlivých bitů. Jazyk C k tomuto účelu poskytuje šest ***bitových operátorů***. [HePa13, str. 262]

| Bitový součin (AND) | Bitový součet (OR) | Bitová negace (NOT) | Bitový exkluzivní součet (XOR) | Bitový posuv vlevo | Bitový posuv vpravo |
|---|---|---|---|---|---|
| & | \| | ~ | ^ | << | >> |

Tyto operátory mají odlišnou funkci od logických operátorů. Výsledkem aplikace bitových operátorů není logická hodnota, ale hodnota číselná.

*Příklady:*
```
0xF0 && 0x88  // logický součin – výsledkem je logická pravda (1)
0xF0 & 0x88 // bitový součin – výsledkem je 0x80
```

### 3.6.5 Adresové operátory

Tyto operátory se používají ve spojení s ukazateli. [HePa13, str. 144]

| Referenční operátor | Dereferenční operátor |
|---|---|
| *&jmeno* | **ukazatel* |

***Referenční operátor*** umožňuje získat adresu proměnné. ***Dereferenční operátor*** naopak umožňuje získat hodnotu paměťového místa, na něž odkazuje ukazatel.

*Příklady:*
```
int *uCislo = &cislo;  // (int *) je datový typ ukazatel na int,
                       //  ne dereferenční operátor
int hodnota = *uCislo; // nyní jde o dereferenční operátor
```

### 3.6.6 Operátor přetypování

*cast-expression:*
*unary-expression*
**(** *type-name* **)** *cast-expression*

Tento operátor slouží pro explicitní změnu datového typu výrazu. Programátor jeho použitím přebírá zodpovědnost za chování typového systému jazyka. Nevhodné použití explicitních konverzí způsobuje problémy. Programátor by si měl být vědom, jaké dopady bude mít tato násilná změna datového typu.
Přetypování se často používá ve spojitosti s ukazateli, zejména s obecným ukazatelem (**void** *).
[HePa13, str. 154]

*Příklady často používaných konverzí:*
```
(char)ordinální_hodnota //získání znaku z ordinální hodnoty
```

```
(int)výraz_float  // oříznutí desetinné části (Pozor! Problémy s
                  // velkými čísly. Raději trunc(), round(), …)
(float)výraz_int //převod na racionální typ
```

### 3.6.7 Operátor sizeof

**sizeof** *unary-expression*
**sizeof ( ***type-name* **)**

Jak už bylo řečeno dříve, norma jazyka nezaručuje přesné rozměry číselných datových typů. Pokud potřebujeme s jistotou zjistit, kolik prostoru zabírá konkrétní datový typ, používáme k tomu operátor **sizeof**. [HePa13, str. 161] Jako parametr tohoto operátoru můžeme zadat jakýkoli datový typ nebo výraz. Operátor vrací velikost v bajtech.

*Příklady:* Použití operátoru **sizeof.**

```
sizeof(int) // == 4 nebo 2 nebo 8 – záleží na architektuře
            // počítače

sizeof(promenna) // rozměr datového typu proměnné promenna
```

Zvláštní význam má tento operátor u polí. V tomto případě vrací velikost pole jakožto součet velikostí jeho jednotlivých položek.

*Příklad:* Použití operátoru **sizeof** s polem.

```
int pole[10];
sizeof(pole) // == 10*sizeof(int) == 40  – za předpokladu,
             // že sizeof(int) == 4
```

### 3.6.8 Podmíněný operátor (ternární operátor)

***Podmíněný operátor***, nebo také ***ternární operátor*** [HePa13, str. 50], slouží pro vytváření podmíněných výrazů, jejichž výsledek závisí na vstupní podmínce. Syntaxe tohoto výrazu vypadá takto:

*conditional-expression:*
*logical-OR-expression*
*logical-OR-expression* **?** *expression* **:** *conditional-expression*

*podmínka ? varianta_true : varianta_false*

Pokud je zadaná *podmínka* pravdivá, má výraz hodnotu *varianta_true*, v opačném případě má výraz hodnotu *varianta_false*.

*Příklad:*

```
int max = (a > b)? a : b; // maximum z a, b
```

*Doporučení:*

- Používat pouze ve výrazech.
- Nezneužívat jako náhradu podmíněného příkazu **if**.
- Nevnořovat ternární operátory do sebe – vede k nesrozumitelnému kódu.

### 3.6.9 Operátor volání funkce

*jméno_funkce()*
*jméno_funkce(parametry)*

V jazyce C je pro zavolání funkce nutné použít ***operátor volání funkce***, a to i pro funkce, které nemají žádné parametry. Uvedení identifikátoru funkce bez závorek má význam ukazatele na tuto funkci.

*Příklady:* Volání funkce.
```
getchar();
printf("HELLO.");
```

### 3.6.10 Přístupové operátory

***Přístupové operátory*** slouží ke zpřístupnění složek složitějších datových typů, jako jsou pole, struktury, či dynamické struktury.

| Indexování | Přístup k prvku struktury | Přístup k prvku struktury přes ukazatel na strukturu |
|---|---|---|
| [] | . | -> |

*Příklady:* Přístupové operátory
```
pole[i+2] = pole[i+1]+pole[i];
souradnice.x = 10;
souradnice.y = 20;
```

`(*ukazatel).prvek` je ekvivalentní se zápisem `ukazatel->prvek`

### 3.6.11 Operátor čárka

Tento operátor zřetězuje dva výrazy a zajišťuje přesné pořadí jejich vyhodnocení zleva doprava. Nabývá hodnoty nejpravějšího podvýrazu.
Použití operátoru má tvar:
*výraz_1, výraz_2*

Používá se například v příkazu **for**.

*Příklad:* Použití operátoru čárka.
```
for (int i = 0, j = 10; i != j; i++, j--) // i++, j-- - použití operátoru čárka

int vysledek = (a + b, c + d); // výsledkem je c + d, a + b se "zahodí"
vysledek = a + b, c + d; // výsledkem je  a + b, c + d je mrtvý kód
```

*Zásada:*
Raději nepoužívat, smysluplně se dá použít jen v minimálním počtu případů.

### 3.6.12 Priorita operátorů

Priorita operátorů [HePa13, str. 269] udává, v jakém pořadí se budou vyhodnocovat jednotlivé podvýrazy.
Vhodně zavedená hierarchie priorit operátorů dovoluje značně redukovat počet závorek ve výrazech. Pomocí závorek lze měnit pořadí vyhodnocení výrazu podle potřeby.
Ve všech jazycích je přesně specifikována priorita všech operátorů. (v některých jsou však priority navrženy neprakticky – viz Pascal). Platí však pravidlo:

"Nejsi-li si jistý prioritou operátorů, závorkuj."

*Příklad:*

```
X = 3+2*10; // výsledek 23
x = (3+2) * 10; // výsledek 50
```

**Tabulka operátorů a jejich asociativita** [HePa13, str. 270]

| Priorita | **Operátory** | **Asociativita** | **skupina op.** |
|---|---|---|---|
| 1. | () [] -> . | zleva doprava | primární |
| 2. | ! ~ ++ -- + - (typ) * & **sizeof** | zprava doleva | unární |
| 3. | * / % | zleva doprava | multiplikativní |
| 4. | + - | zleva doprava | aditivní |
| 5. | << >> | zleva doprava | posuny |
| 6. | < <= > >= | zleva doprava | relační |
| 7. | == != | zleva doprava | relační |
| 8. | & | zleva doprava | bitový součin |
| 9. | ^ | zleva doprava | exkl. bit. souč. |
| 10. | \| | zleva doprava | bitový součet |
| 11. | && | zleva doprava | logický součin |
| 12. | \|\| | zleva doprava | logický součet |
| 13. | ?: | zprava doleva | ternární podm. |
| 14. | = += -= *- /= %= >>= <<+ &= = ^= | zprava doleva | přiřazení |
| 15. | , | zleva doprava | op. čárka |

## *3.7 Shrnutí*

V této kapitole jsme se seznámili s úlohou správné organizace dat při programování. Čtenář nyní umí popsat vlastnosti datových struktur. Ví, jakými vlastnostmi je jednoznačně určen každý datový typ a umí vyjmenovat a popsat základní datové typy, s nimiž lze na počítači pracovat.
U programovacího jazyka umí rozlišit, jaký typový systém tento jazyk používá.

Čtenář je dále schopen identifikovat elementy programovacího jazyka. V této kapitole je jejich syntaxe v jazyce C popsána pomocí EBNF a čtenář je schopen se v ní orientovat a používat ji (EBNF je popsána v předchozí kapitole 1.).

Čtenář je nyní již také schopen vyvářet proměnné v jazyce C a chápe rozdíl mezi jejich deklarací, definicí a inicializací. Z proměnných je schopen pomocí operátorů vytvářet výrazy. Dokáže také vyjmenovat druhy používaných operátorů a má základní představu o vzájemných prioritách nejpoužívanějších operátorů během vyhodnocování výrazů.

## 3.8 Úlohy k procvičení

Zapište v jazyce C:

podmínky:

$x \in \langle 0,1)$,
$x \notin (5,10\rangle$,
$x \in \{2,5,8\}$,
$x \notin \{1,5\}$,
$x \neq \pm 1$,
c je dělitelné 5,
b je sudé,
$x \geq 1 \wedge$ x je liché,
c je dělitelné 2 nebo 3.

negaci výrazů (bez použití operátoru (!)):

$A>0$,
$A+B \geq C$,
$X*Z=Z*A$,
$(A>0) \wedge (B>0)$,
$(A=0) \vee (B=0)$,
$\neg(A>B)$,

## 3.9 Kontrolní otázky

1. Čím je určen datový typ?
2. Uveďte rozdělení datových typů.
3. Vysvětlete použití specifikátoru typu **signed** a **unsigned**.
4. Jak překladač rozliší typ konstanty (literálu)?
5. Uveďte druhy operátorů.
6. K čemu slouží operátor **sizeof**?
7. Jak probíhá vyhodnocení výrazů?