Data, výrazy, příkazy

Karel Richta a kol.

katedra počítačů FEL ČVUT v Praze

Přednášky byly připraveny s pomocí materiálů, které vyrobili Ladislav Vágner, Pavel Strnad, Martin Hořeňovský, Aleš Hrabalík

Programování v C++, A7B36PJC
09/2015, Lekce 2

https://cw.fel.cvut.cz/wiki/courses/a7b36pjc/start

Reprezentace dat

- Programy pracují s informací.
- Ta musí být programu přístupná – musí být uložena v paměti počítače.
- V imperativních jazycích k tomu máme konstanty a proměnné.
- Způsob uložení informace závisí na počítači, operačním systému a programovacím jazyce.
- Způsob uložení triviálních informací není v C a C++ určen striktně, jen jsou stanovena určitá pravidla. Každá implementace ale musí způsob reprezentace zvolit.
- Zápis triviálních konstantních informací nazýváme **literály**.
- Ten je (na rozdíl od reprezentace) určen striktně.

Lexikální elementy

- **identifikátory**

  písmena (rozlišuje se velikost), číslice, '_'

  délka není omezena, rozlišují se podle úvodních znaků v závislosti na implementaci (ANSI doporučuje nejméně 31 znaků)

  některé jsou rezervovány jako klíčová slova
- **klíčová slova**

  if while … malými písmeny!
- **omezovače** a **operátory**

  { } ; + - & && <<=
- **číselné literály (zápisy čísel)**

  125 125.45e-7 0.12E3

  oktalové: 037 hexadecimální: 0x1F

Lexikální elementy (pokračování)

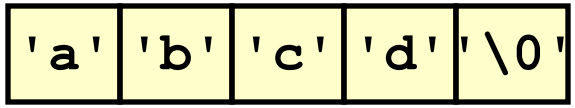
- **znakové literály (zápisy znaků)**

  'x' '\n' '\t' '\0' '\\' '\'' '\014'

- **řetězce**

  "abcd" "" "xyz\nabc"

vnitřní reprezentace: posloupnost znaků zakončená binární nulou `'\0'`



- **komentáře**

  /* toto je komentář */

  // toto je komentář v novější normě C a v C++

- **logické hodnoty (jen v C++)**

  true, false

Typy dat

Jednoduché typy

- **celočíselné (pevná řádová čárka)**
- **racionální (pohyblivá řádová čárka)**
- **logické (jen v C++)**
- **void**: typ s prázdnou množinou hodnot

Odvozené (strukturované) typy

- **pole**
- **struktura** (záznam)
- **třída**
- **union** (sjednocení)

Funkce

Ukazatelé

Základní typy dat

- ***pevná řádová čárka:***

  `char` (nejmenší adresovatelná buňka):

  `char, unsigned char, signed char`

  `int` (dle šířky procesoru)

  `short = short int = signed short = signed short int`
  `int = signed = signed int`
  `unsigned = unsigned int`
  `long = long int = signed long = signed long int`
  `unsigned long = unsigned long int`

  `short` $\subseteq$ `int` $\subseteq$ `long`

  výčtové:

  množina symbolických celočíselných konstant

- ***pohyblivá řádová čárka:***

  `float, double, long double`

- ***logické hodnoty (jen C++):***

  `true, false`

- `void`: typ s prázdnou množinou hodnot

Odvozené (strukturované) typy

- **Pole**
  - jednorozměrná pole prvků libovolného typu (kromě funkce)
  - indexováno vždy od 0
- **Struktura** (záznam)
  - obsahuje pojmenované položky různých typu uložené za sebou, implicitně přístupné
- **Třída**
  - obsahuje pojmenované položky různých typu uložené za sebou, implicitně nepřístupné zvenku
- **Union** (sjednocení)
  - pojmenované položky různých typů uložené "přes" sebe

Funkce a ukazatelé

- **Funkce**

  specifikuje funkci typem návratové hodnoty a typy parametrů

- **Ukazatel**

  adresa datového objektu (proměnné nebo konstanty) nebo funkce specifikovaného typu

- **Klasifikace typů**:
  - aritmetické typy: celočíselné + racionální
  - skalární typy: aritmetické + ukazatelé

Strukturované typy a funkce budeme probírat později.

Typy dat (příklad reprezentace)

Vnitřní reprezentace skalárních typů na PC

- celočíselné typy (reprezentace v pevné řádové čárce) v doplňkovém kódu se znaménkem
- racionální typy (reprezentace v pohyblivé řádové čárce) podle normy IEC 60559:1989 (ANSI/IEEE)

Velikost vnitřní reprezentace:	**BC3.1**	**VC++**
char, signed char, unsigned char	1B	1B
short, unsigned short	2B	2B
int, unsigned int	2B	4B
long, unsigned long	4B	4B
float	4B	4B
double	8B	8B
long double	10B	8B

Cvičení: Jak to zjistit?

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  printf("Reprezentace zakladnich typu dat:\n");
  printf("Delka reprezentace char = %d\n",sizeof(char));
  printf("Delka reprezentace int = %d\n",sizeof(int));
  printf("Delka reprezentace float = %d\n",sizeof(float));
  …

  printf("Delka reprezentace ukazatelu = %d\n",
          sizeof(void *));
  return;
}
```

Řetězce

- V C neexistuje samostatný typ pro reprezentaci textových řetězců.
- Řetězce jsou reprezentovány pomocí pole znaků (typu **`char`**), ukončeného znakem `'\0'`. Samotný řetězec je pak reprezentován ukazatelem na prvý znak (typu **`char*`**).
- V C++ existuje typ **`string`**.

Popis jazyka

- Popis syntaxe (**jak** se to píše)
- Popis sémantiky (**co** to znamená)
- Standardní knihovny (standardní služby, které musí každá implementace jazyka poskytovat - základ přenositelnosti programů)

Popis syntaxe a sémantiky je uspořádán takto:

- popis použité paměti (deklarace)
- popis výpočtů prováděných nad pamětí (výrazy)
- popis posloupnosti provádění výpočtů (příkazy)

Deklarace

paměťová třída kvalifikátor specifikace typu seznam deklarátorů **;**

Příklady:

```
int n;

const int x;

extern pole[10];

auto long double z;
```

Paměťová třída

paměťová třída ***kvalifikátor*** ***specifikace typu*** ***seznam deklarátorů*** **;**

Paměťová třída popisuje doporučený způsob přidělování paměti:

auto	paměť přidělena dynamicky na zásobníku
static	paměť přidělena staticky během překladu
register	jako paměť použít registr procesoru
extern	nepřidělovat paměť

- Implicitně je paměťová třída:

  auto (ve funkcích, v bloku) nebo
  static (v modulu, globální proměnné).

Cvičení: Funkce, která si počítá volání

```
int count()
{
      ?
}
```

Kvalifikátor

paměťová třída kvalifikátor specifikace typu seznam deklarátorů ;

Kvalifikátor popisuje další požadované vlastnosti:

const	konstantní objekt
volatile	objekt, který je sdílen s jiným nezávislým procesem

```
extern const int Max;
…
Max = 10;      /* chyba */
```

Specifikace typu

paměťová třída ***kvalifikátor*** ***specifikace typu*** ***seznam deklarátorů*** **;**

Specifikace typu je:

- **`void`** zastupuje "žádný" typ (prázdná množina hodnot)
- "*nic*" implicitně se doplní **`int`**
- *identifikátor typu* (základního nebo uživatelem definovaného)
- popis struktury (**`struct`**), popis třídy (**`class`**), popis sjednocení (**`union`**) nebo výčtového typu (**`enum`**)

Seznam deklarátorů

paměťová třída kvalifikátor specifikace typu seznam deklarátorů ;

Seznam deklarátorů má syntaxi:

identifikátor

*deklarátor

deklarátor [konstantní výraz]

deklarátor (parametry)

(deklarátor)

Pozn.: za * může být kvalifikátor.

Seznam deklarátorů

paměťová třída kvalifikátor specifikace typu seznam deklarátorů **;**

Příklady:

`int i;`	proměnná typu **`int`**
`int *ai;`	proměnná typu ukazatel na **`int`**
`int ia[10];`	proměnná typu pole 10-ti prvků typu **`int`**
`int fi(double);`	proměnná typu funkce z **`double`** do **`int`**
`(int i);`	proměnná typu **`int`**

Čtení složitějších deklarací

Deklarátory je třeba číst **”zevnitř ven"** postupně podle následujících pravidel:

- najdi deklarovaný identifikátor a zjisti, zda se **vpravo** od něj nachází [nebo (
- interpretuj tyto závorky a pak zjisti, zda se **vlevo** od identifikátoru nachází *
- kdykoliv narazíš na), vrať se zpět a aplikuj předchozí pravidla uvnitř závorek (a)
- nakonec použij specifikaci typu

Příklad čtení deklarace

```
char *( *(*x)() )[10];
```

1.identifikátor x je deklarován jako

2.ukazatel na

3.funkci vracející

4.ukazatel na

5.pole 10-ti elementů, kterými jsou

6.ukazatelé na

7.hodnoty typu **`char`**

Definice vlastních typů

Syntaxe:

`typedef` *deklarace*

Příklady:

```
typedef short int Bool;
typedef unsigned char Byte;
typedef unsigned short int Word;
typedef int Matice[10][10]; /* matice 10x10 */
typedef char *(*ppc)[4]; /* uk.na pole uk.na char */
```

Možné deklarace:

```
static Bool b;
extern Word *ptr;
extern Matice m;
const ppc ptab;                    *ptab[2]= “alfa“;
```

Abstraktní deklarátor

v popisu formálních parametrů v deklaraci funkce, při přetypování, nebo jako parametr operátoru **`sizeof`** lze vypustit identifikátor, tj. použít **abstraktní deklarátor**

`int *`	ukazatel na **`int`**
`char (*)(int)`	ukazatel na funkci z **`int`** do **`char`**
`unsigned *[4]`	pole 4 ukazatelů na **`unsigned`**
`int (* (*)() )()`	ukazatel na funkci, která vrací ukazatel na funkci s libovolným počtem parametrů vracející **`int`**

Nepovolené typy

- `void` mimo následující povolené možnosti:

  void f();

  … f(void);

  void *

- pole funkcí
- funkce vracející pole
- funkce vracející funkci

Popis výčtového typu

- Syntaxe:

  enum ***značka*** **{** ***seznam literálů*** **}**

- Příklad:

  enum Color {Red, Blue, Green};

- Literály jsou synonyma celočíselných hodnot

  /* Red = 0, Blue = 1, Green = 2 */

Popis výčtového typu (pokračování)

- Literálu lze explicitně přiřadit hodnotu

```
enum Masky { Nula, Jedna, Dva, Ctyri=4, Osm=8 };
/* Dva = 2, Ctyri = 4, Osm = 8 */

enum Sign { Minus = -1, Zero, Plus};
/* Minus = -1, Zero = 0, Plus = 1 */

enum { E1, E2, E3 = 5, E4, E5 = E4 + 10, E6};
/* E4 = 6, E5 = 16, E6 = 17 */
```

- Výčtový typ je kompatibilní s `int` i co do délky vnitřní reprezentace

```
enum Sign s;          /* totéž co int s; */
```

Inicializace

- Datové objekty (proměnné, konstanty) se inicializují v době vzniku
  - statické na začátku programu
  - ostatní na začátku funkce (bloku)
- Implicitní inicializace
  - statické objekty jsou vynulovány
  - ostatní nemají definovanou hodnotu
- Explicitní inicializace
  - *deklarátor* = *výraz* nebo
  - *deklarátor* = { *seznam výrazů* }
- Statické objekty lze inicializovat pouze konstantními výrazy
- Příklady:

```
float A = 10;
register int B = A*A;
```

Třídy identifikátorů v C a C++

- jména maker - zpracuje preprocesor (nezávislá na ostatních jménech)
- návěští - pozná se `nav:` (lokální ve funkci - musí být jednoznačné ve funkci)
- jména struktur, tříd, sjednocení a výčtů (musí být jednoznačná v rámci jednotlivých kategorií)
- jména položek (musí být jednoznačná v rámci struktury, třídy nebo sjednocení)
- ostatní jména (proměnné, funkce, typy, konstanty výčtu) musí být v rozsahu platnosti jednoznačná
- v C++ se mohou jména funkcí (metod) opakovat, pokud je lze rozlišit podle skutečných parametrů

Rozsah platnosti identifikátorů

- globální objekty - od místa deklarace do konce zdrojového souboru
- formální parametry - lokální objekty v těle funkce
- formální parametry v deklaraci funkce - do konce deklarace
- lokální objekty - od místa deklarace do konce bloku
- návěští - lokální v těle funkce
- makro - od direktivy **`#define`** do konce zdrojového souboru nebo direktivy **`#undef`**

Preprocesor

Příklad: *Definice symbolické konstanty*

```
#define MAX 100
printf("MAX=%d\n",MAX);
```

```
MAX=100
```

Příklad: *Definice makra*

```
#define SUM(X,Y) ((X)+(Y))
printf("SUM(3,7)=%d\n",SUM(3,7));
```

```
SUM(3,7)=10
```

Ukázka programu v C++ se symb. konst.

```
/* převodní tabulka Fahrenheit - Celsius: C = (5/9)(F-32) */
#include <iostream>
#define POCATEK 0
#define KONEC 300
#define KROK 20
int main(void)
{
    int F; float C;

    F = POCATEK;
    while (F <= KONEC) {
       C = (5.0 / 9.0) * (F - 32);  /* spočte stupně v C */
       std::cout << F << " - " << C << "\n"; /* tiskne řádek tabulky */
       F = F + KROK;              /* posune na další F */
    }
    return 0;
}
```

definice symbolických konstant (makra bez parametrů)

Ukázka programu v C++ s konstantami

```
/* převodní tabulka Fahrenheit - Celsius: C = (5/9)(F-32) */
#include <iostream>
const int pocatek = 0;
const int konec = 300;
const int krok = 20;
int main(void)
{
    int F; float C;

    F = pocatek;
    while (F <= konec) {
       C = (5.0 / 9.0) * (F - 32);  /* spočte stupně v C */
       std::cout << F << " - " << C << "\n"; /* tiskne řádek tabulky */
       F = F + krok;              /* posune na další F */
    }
    return 0;
}
```

definice konstantních buněk

Preprocesor (doplňky)

Konstanty zavedené pomocí **`const`** a **`#define`**:

- Poskytují podobnou funkcionalitu.
- **`#define`** je v C i v C++, **`const`** pouze v C++.
- Debugger zná konstanty zavedené **`const`**, ale nezná konstanty nahrazené preprocesorem.
- Konstanty zavedené **`#define`** nerespektují jmenné prostory a pravidla zastiňování.
- **`#define`** je lepší použít, pokud píšeme rozhraní dynamicky linkované knihovny:
  - nešikovné řešení s **`const`** zavádí zbytečné relokace,
  - u dynamicky linkovaných knihoven není předem jasné, kde všude budou použity (zda vždy pouze v C++).

Preprocesor (doplňky)

Podmíněný překlad:

```
#if konstantní výraz
#ifdef jméno
#ifndef jméno
#elif konstantní výraz
#else
#endif
```

Příklad:

```
#if MAX > 10
        Zde je kód pro MAX > 10
#else
        Zde je kód pro MAX <= 10
#endif
```

Preprocesor (doplňky)

Příklad: *vícenásobné větvení*

```
#if MAX > 10
        Zde je kód pro MAX > 10
#elif MAX > 5
        Zde je kód pro MAX <= 10 a MAX > 5
#else
        Zde je kód pro MAX <= 5
#endif
```

Preprocesor (doplňky)

Příklad: *korektní předefinování makra*

```
#ifdef MAX
#undef MAX
#endif
#define MAX 100
```

Preprocesor (doplňky)

Příklad: stráže hlaviček knihoven (umožňují libovolný počet vložení hlavičky)

```
#ifndef __STDIO_H
#define __STDIO_H
  …
     tělo knihovny
  …
#endif
```

Alternativa:

```
#pragma once
```

Preprocesor (doplňky)

- Preprocesor nabízí některá předdefinovaná makra:
  - **__FILE__** jméno zdrojového souboru,
  - **__LINE__** číslo řádky v aktuálním souboru,
  - **__func__** jméno aktuálně překládané funkce,
  - **__cplusplus** příznak, zda je zdrojový kód kompilován překladačem C nebo C++.
- Kompilátory pak nabízejí předdefinovaná makra specifická pro výrobce/platformu:
  - **WIN32** příznak kompilace pro Windows,
  - **BORLANDC** Borland kompilátory,
  - **DEBUG** příznak kompilace pro ladění,
  - **RELEASE** příznak kompilace pro ostré nasazení.

Preprocesor (doplňky)

Předdefinovaná makra mohou usnadnit ladění:

```
int divide ( int num, int denom ) {
   if ( denom == 0 )
    {
       cout << "Deleni nulou" << endl <<
            " funkce: " << __func__ <<
            " soubor: " << __FILE__ <<
            " radek: "  << __LINE__ << endl;
      return ( 0 );
    }
   return ( num / denom );
 }
```

Preprocesor - assert

- Pomoc při ladění – makro **`assert`** v hlavičkovém souboru **`<assert.h>`** (**`<cassert>`**).
- Makro má jako parametr podmínku:
  - je-li podmínka platná, nic se nedělá,
  - není-li podmínka platná, zobrazí pozici a podmínku, která vedla k jeho vyvolání. Volitelně umožní ukončit program.
- Volání **`assert`** se typicky nechají v kódu:
  - pro ladění se použijí,
  - ve finální verzi se hromadně odpojí definicí:

```
#define NDEBUG
```

Preprocesor – assert (příklad)

```
#include <cassert>
#include <stdio.h>      /* printf */

void print_number(int* myInt) {
         assert(myInt != NULL);
         printf("%d\n", *myInt);
}

int main()
{
         int a = 10;
         int *b = NULL;
         int *c = NULL;

         b = &a;

         print_number(b);
         print_number(c);

         return 0;
}
```

```
Assertion failed: myInt != NULL, file
d:\výuka\fel\a7b36pjc\prednasky\2015\priklady\assert\assert\assert.cpp, line 12
abnormal program termination
```

Výrazy

- Bohatý repertoár **operací**: aritmetické, bitové, logické, relační, operace s ukazateli
- Výraz může označovat **modifikovatelný objekt** (lvalue) nebo **hodnotu** (rvalue)
- Pořadí vyhodnocení operandů není (až na výjimky) definováno!
- Některé operace mají vedlejší efekt (inkrementace, dekrementace, přiřazení) => v jednom výrazu nad jedním objektem max. jeden!

Výrazy (pokračování)

- Operace se provádějí v těchto aritmetikách:

  `int, unsigned, long, unsigned long,`

  `float, double, long double`

- Před provedením operace může dojít k ***implicitní konverzi operandů***:
  - **`char`**, **`short`** (**`signed`**, **`unsigned`**) a **`enum`** jsou před provedením operace konvertovány na **`int`** (nebo **`unsigned int`**) - tzv. ***roztažení*** (***integral promotion***)
  - jméno pole prvků typu *T* je konvertováno na typ ukazatel na *T*
  - jméno funkce je konvertováno na typ ukazatel na funkci

Výrazy (pokračování)

U většiny operací s aritmetickými operandy se provádějí tzv. ***běžné aritmetické konverze*** (usual arithmetic conversion):

- operandy se konvertují pomocí roztažení, a potom:
- je-li typ jednoho operandu: druhý se převede na:

`long double`	`long double`
`double`	`double`
`float`	`float`
`unsigned long`	`unsigned long`
`long`	`long`
`unsigned`	`unsigned`
`int`	`int`

Typy číselných literálů

- Typy číselných literálů:

  12 int 12L long

  3.4 double 3.4F float 3.4L long double

- Příklady:

  'a' + 'b' konverze na int, výsledek int

  1 / 2 celočíselné dělení, výsledek 0

  1.0 / 2.0 reálné dělení, výsledek double

  1 / 2.0L konverze na long double

Aritmetické operace

- Operandy aritmetického typu, provádějí se běžné aritm. konverze
- Unární: + –
- Binární: + – * / %
- Dělení je podle typu operandů buď celočíselné nebo v pohyblivé řádové čárce
- Operace % je zbytek po dělení, musí mít operandy celočíselné, a je definována takto:

  x == (x / y) * y + x % y

- Operace + a - jsou definovány též pro ukazatele (viz dále)

Relační operace

- Operandy aritmetického typu, provádějí se běžné aritm. konverze
- Výsledek je v C vždy typu `int`:
  - **0** … relace neplatí
  - **1** … relace platí
- Výsledek v C++ je typu `bool`:
  - **false** … relace neplatí
  - **true** … relace platí
- Binární:

  `<  >  <=  >=  ==  !=`
- Relační operátory jsou definovány též pro ukazatele

Logické operace

- Operandy skalárního typu
- Výsledek je v C typu `int` (0 nebo 1)
- Výsledek v C++ je typu `bool` (false nebo true)
- Operandy se vyhodnocují **zleva**, druhý operand se nevyhodnocuje, je-li výsledek dán prvním operandem

!	**logická negace** - výsledek 1, má-li operand hodnotu 0, jinak je výsledek 0
&&	**logický součin** - výsledek 1, jsou-li oba operandy nenulové, jinak 0
||	**logický součet** - výsledek 1, je-li alespoň jeden operand nenulový, jinak 0

Příklad: načtení řádky znaků

```
#define MAXDELKA 80

char radek[MAXDELKA+1];
int i = 0;
int c;
int konec = 0;

while ( !konec ) {
  if ((i <= MAXDELKA) &&
      ((c = getchar()) != '\n') &&
      (c != EOF))
  {
    radek[i] = c;
    i = i + 1;
  }
  else konec = 1;
}
```

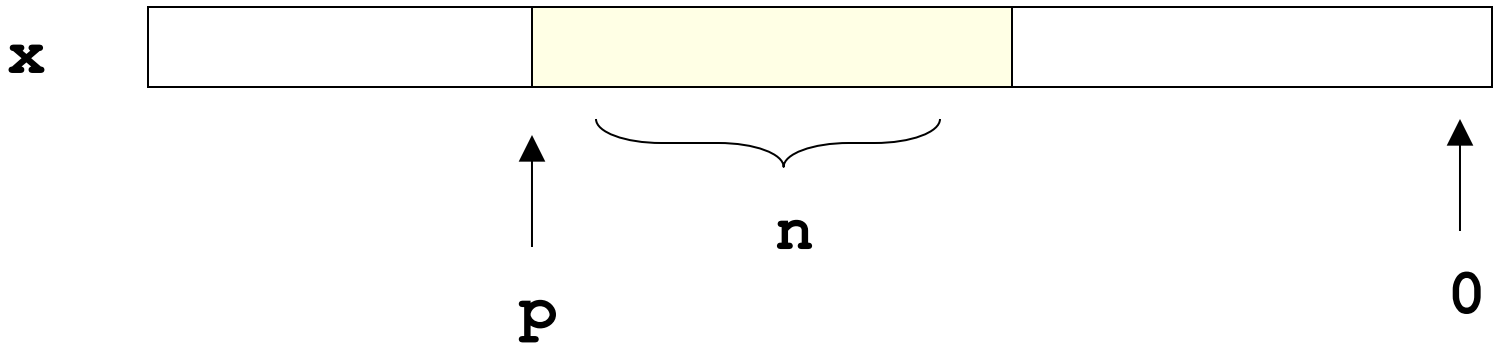
když není místo, vůbec se nečte

Bitové operace

- Operandy jen celočíselné, provádějí se běžné aritmetické konverze
- Unární:
  - ~ negace všech bitů
- Binární:
  - & logický součin všech bitů
  - | logické součet všech bitů
  - ^ logické *xor* všech bitů
  - << posun bitové reprezentace levého operandu vlevo
  - >> posun bitové reprezentace levého operandu vpravo
- U operací posunu pravý operand udává délku posunu v bitech
- Pozor, plete se & a &&, | a || :

```
x = 1; y = 2;
x & y == 0
x && y == 1
```

Příklad: vyříznutí n bitů z x od pozice p




```
unsigned getbits(unsigned x, unsigned p, unsigned n)
{
  return (x >> (p-n+1)) &
         (~((~0)<<n));
}
```

Inkrementace a dekrementace

- Unární operace, operand skalárního typu, modifikovatelný, označení místa v paměti
- Lze zapsat prefixově nebo postfixově
- Má vedlejší efekt: změní hodnotu operandu
- Inkrementace: ++ Dekrementace: --
- Prefix: ***++X*** ***--X***

  inkrementuje (dekrementuje) *X*, hodnotou je změněné *X*
- Postfix: ***X++*** ***X--***

  inkrementuje (dekrementuje) *X*, hodnotou je *X* před změnou

Příklady:

před	operace	po
x == 1	y = ++x	x == 2, y == 2
x == 1	y = x++	x == 2, y == 1
x == 1	y = --x	x == 0, y == 0
x == 1	y = x--	x == 0, y == 1

Přiřazení

- Syntaxe: ***X*** **=** ***Y***
- Levý operand musí být modifikovatelný, označení místa v paměti
- Přiřazení má hodnotu a vedlejší efekt
- Hodnotou výrazu je přiřazovaná hodnota, typem je typ levé strany
- Pozor: plete se = a == !
- Přípustné typy operandů:
  - levá a pravá strana jsou téhož skalárního typu nebo téhož typu **`struct`** nebo **`union`** (pro pole není přiřazení dovoleno)
  - jsou-li typy levé a pravé strany skalární, avšak různé, musí být hodnota pravé strany konvertibilní na typ levé strany
- Složené přiřazení:

  `+= -= *= /= %= &= |= ^= <<= >>=`
- Význam:
  - *X op= Y* je zkratkou za *X = X op Y*

Konverze při přiřazení

Následující tabulka udává možné konverze při přiřazení:

typ pravé strany	typ levé strany	poznámka ke konverzi
racionální	kratší racionální	zaokrouhlení mantisy
racionální	delší racionální	doplnění mantisy nulami
racionální	celočíselný	odseknutí necelé části
celočíselný	racionální	možná ztráta přesnosti
celočíselný	kratší celočíselný	odseknutí vyšších bitů
celočíselný unsgn.	delší celočíselný	doplnění nulových bitů
celočíselný sgn.	delší celočíselný	rozšíření znaménka
0	T *	
T * nebo 0	void *	

Reference a dereference

- Reference (adresa):

  & *X* kde *X* je označení datového objektu nebo funkce

- Je-li *X* typu *T*, pak &*X* je typu *T**, tzn. ukazatel na *T*
- Dereference (zpřístupnění objektu, na který ukazuje ukazatel):

  ***** ***X*** kde *X* je výraz typu ukazatel

- Je-li *X* typu *T**, pak **X* je typu *T*
- Příklady:

```
int  i, *pi = &i;          /* pi obsahuje adresu i */
char c, *pc = &c;          /* pc obsahuje adresu c */
*pi = 25;                  /* do i se uloží 25 */
*pc = 'a';                 /* do c se uloží 'a' */
pi = pc;                   /* chyba */
```

Reference a dereference (pokračování)

- Příklady:

```
char str[] = "Hello, word";
char *ptr  = &str[2];
/* ptr pbsahuje adresu 3. prvku řetězce */
*ptr = '1';
/* změní se 3. prvek řetězce */
```

Aritmetika s ukazateli

- Ukazatelé mohou být operandy sčítání a odčítání
- Dovolené kombinace a typ výsledku:

  T* **+ int -> *T****

  T* **- int -> *T****

  T* **- *T**** **-> int**
- Přičtení *n* k ukazateli typu *T** znamená jeho změnu o *n*-násobek délky typu *T,* podobně odečtení a rozdíl ukazatelů.
- Příklady:

```
int a[10], *p = &a[0];
*(p + 3) = 10;              /* do a[3] se uloží 10 */
/* vynulování pole a */
for (p = &a[0]; p <= &a[9]; p++) *p = 0;
/* nebo */
for (p = &a[0]; p <= &a[9]; *p++ = 0);
/* nebo */
for (p = a; p <= a + 9; *p++ = 0);
```

Indexace

- Syntaxe: ***X* [*Y*]** , kde jeden výraz je typu ukazatel na *T* a druhý typu *int*, výsledek je typu *T*
- Jméno pole prvků typu *T* je konstantní ukazatel na *T*
- Příklady:

```
int a[10], i, *p;
a[1] = 5;
p = a;
*(p + 2) = 0;
p[2] = 0;
```

- Výraz *X* [*Y*] je ekvivalentní s *(*X* + (*Y*))
- Žádná kontrola indexů se neprovádí
- Vícerozměrná pole jsou pole polí, indexace v každém rozměru zvlášť

```
int mat[2][3];
mat[1][2] = 1;
```

- Pozor:

  mat[1,2] je dovoleno, neznamená však dvojitou indexaci!!

Selekce

- Zpřístupnění položky struktury, třídy nebo unionu
- Přímá selekce:

  X . položka kde *X* je výraz typu **`struct`** nebo **`union`**
- Nepřímá selekce (přes ukazatel):

  X -> položka kde *X* je výraz typu ukazatel na **`struct`** nebo **`union`**
- *X -> položka* je zkratkou za (**X*) *. položka*
- Příklad:

```
struct { int a; char b; } x, *px = &x;
x.a = 1;
px -> b = 'a';
(*px).b = 'a';          /* totéž */
```

Operátor délky typu

- Zjištění velikosti vnitřní reprezentace objektu nebo typu
- Syntaxe:

  `sizeof` *výraz* nebo

  `sizeof` **`(`** *označení typu* **`)`**
- Výsledek v bytech (přesněji nejmenších adresovatených jednotkách), typu **`size_t`** (většinou **`unsigned`**)
- Výraz se nevyhodnocuje, jména polí se nekonvertují na ukazatele!
- Příklady:

```
int pole[] = {1, 2, 3, 4};
sizeof(pole) / sizeof(int) /* počet prvků pole */
sizeof(void *) /* délka vnitřní reprezentace ukazatelů */
```

Přetypování (casting)

- Explicitní změna typu (většinou u ukazatelů), může znamenat změnu vnitřní reprezentace
- Syntaxe:

  (*T*) *X*

  kde *T* je označení skalárního typu nebo **`void`** a *X* je výraz skalárního typu

- Příklady:

```
int a, b; float r;
r = (float)a / b;

int *p;
char *q;
q = p;                  /* chyba nebo varovné hlášení */
q = (char*)p;           /* O.K. */
```

Podmíněný výraz

- Výsledek výrazu je závislý na hodnotě podmínky
- Syntaxe:

  Podmínka* ? *Výraz1* : *Výraz2

- kde *Výraz1* a *Výraz2* jsou kompatibilních typů
- Je-li *Podmínka* splněna, je výsledkem hodnota *Výrazu1*, jinak je výsledkem hodnota *Výrazu2*
- Vyhodnocuje se podmínka a pak jen příslušný výraz
- Příklad:

```
max = a > b ? a : b;
p = r * (x < 0 ? -1 : x == 0 ? 0 : 1);
```

Operátor „čárka“ (comma)

- Umožňuje několik výrazu v místě, kde se očekává jeden výraz (nejčastěji v příkazu **`for`**, viz dále)
- Syntaxe:

  Výraz1 , Výraz2 , … , Výrazn

- Výrazy se vyhodnotí v pořadí zleva doprava, výsledkem je hodnota posledního výrazu (hodnoty ostatních výrazů se "zahodí")
- Pozor: může škodit (viz indexace)
- Nejčastější použití v příkazu **`for`**:

```
for (v = 1, i = n; i > 1; v *= i, i--);
```

Priorita a asociativita operátorů

`[ ] ( ) . -> postfix ++ and postfix --`	Zleva-doprava
`prefix ++ and prefix -- sizeof & * + - ~ !`	Zprava-doleva
`() - přetypování`	Zprava-doleva
`* / %`	Zleva-doprava
`+ -`	Zleva-doprava
`<< >>`	Zleva-doprava
`< > <= >=`	Zleva-doprava
`== !=`	Zleva-doprava
`&`	Zleva-doprava
`^`	Zleva-doprava
`|`	Zleva-doprava
`&&`	Zleva-doprava
`||`	Zleva-doprava
`? :`	Zprava-doleva
`= *= /= %= += -= <<= >>= &= ^= |=`	Zprava-doleva
`,`	Zleva-doprava

Konstantní výrazy

- Konstantní výraz je výraz vyhodnotitelný v době překladu
- Nesmí obsahovat vedlejší efekty
- Použití:
  - počet prvků pole
  - inicializace statických objektů
  - návěští v přepínači atd.

Příkazy

- Obecně:
  - příkazy se dělí na jednoduché a strukturované
  - všechny jednoduché jsou zakončeny středníkem
  - podmínka v cyklech a podmíněném příkazu je dána výrazem skalárního typu; je splněna, je-li hodnota výrazu různá od nuly
- Výrazový příkaz:

  ***výraz* ;**
  - smysl mají jen výrazy s vedlejším efektem

```
x = y + 2; y++; f(a, 2); g();
g;  /* příkaz bez vedlejšího efektu */
```

- Prázdný příkaz:

```
;
```

  - použití např. jako prázdné tělo cyklu

```
for (i = 0; i < 10; a[i++] = 0);
```

Příkaz bloku

- Příkaz bloku:

  { ***posloupnost deklarací posloupnost příkazů*** **}**

  – deklarace jsou v bloku lokální,
  – v bloku platí též deklarace z nadřazeného kontextu,
  – lokální deklarace zastíní deklaraci stejného identifikátoru z nadřazeného kontextu,
  – deklarace mohou chybět (složený příkaz).

```
{    int a, b;
     a = 10;
     { int c = a;          /* c = 10 */
       int a = 20;
       c = a;              /* c = 20 */
       a = 0;
     }
     printf("%d", a);  /* vypíše se 10 */
}
```

Podmíněný příkaz

- Podmíněný příkaz:
  - **`if`** (*podmínka*) *příkaz*
  - **`if`** (*podmínka*) *příkaz1* **`else`** *příkaz2*
    - **`else`** se vztahuje k nejbližšímu předcházejícímu **`if`** ve stejném bloku (závorkuje se zprava)

```
if (a > b) m = a; else m = b;
```

```
if (a > b)
  if (a > c) m = a; else m = c;
```

```
if (a > b) {
  if (a > c) m = a;
} else m = b;
```

Příkazy cyklu

- Příkaz **`while`**:

  `while` **(** ***podmínka*** **)** ***příkaz***

  - tělo cyklu se provádí, dokud je splněna podmínka
  - podmínka se vyhodnocuje na začátku
- Příkaz **`do`**:

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

  - tělo cyklu se provádí, dokud je splněna podmínka
  - podmínka se vyhodnocuje na konci těla
  - jediný strukturovaný příkaz zakončený středníkem
- Příkaz **`for`**:

  `for` **(** ***výraz1*** **;** ***výraz2*** **;** ***výraz3*** **)** ***příkaz***

  - příkaz má stejný efekt, jako příkazy:

```
výraz1 ;
while ( výraz2 ) { příkaz ; výraz3 ; }
```

- V nové verzi C11 a C++11 existují další tvary cyklů pro kontejnery.

Strukturované skoky

- Příkaz *break*:

  ```
  break ;
  ```

  - ukončí bezprostředně nadřazený cyklus nebo přepínač
- Příkaz *continue*:

  ```
  continue ;
  ```

  - přejde na nové vyhodnocení podmínky cyklu

```
while ( podm1 ) {
    …
  if ( podm2 ) break;
  …
  if ( podm3 ) continue;
  …
}
```

Příkaz větvení (přepínač)

- Příkaz **`switch`**:

  `switch` (*výraz*) *příkaz*

  - slouží k rozvětvení výpočtu do několika větví podle hodnoty celočíselného výrazu
  - příkazem je téměř vždy složený příkaz
  - některé z příkazů ve složeném příkazu jsou označeny návěštími ve tvaru:

  `case` *konst. výraz*`:` nebo **`default:`**

  - po vyhodnocení výrazu se pokračuje tou variantou, která je označena návěštím se stejnou hodnotou (návěští musí být různá)
  - návěští **`default`** pokrývá ostatní hodnoty
  - každá větev obyčejně končí příkazem **`break`**

```
switch ( n ) {
  case 1: printf("*"); break;
  case 2: printf("**"); break;
  case 3: printf("***"); break;
  default: printf("-")
}
```

Příklad: použití přepínače

```
int zn = 2;

switch ( zn ) {
  case 1: printf("výborně");
  case 2: printf("velmi ");
  case 3: printf("dobře");
  default: printf("nedobře");
}
```

Chybně

```
int zn = 2;

switch ( zn ) {
  case 1: printf("výborně");
          break;
  case 2: printf("velmi ");
  case 3: printf("dobře");
          break;
  default: printf("nedobře");
}
```

Správně

Příkaz skoku a návratu

- Příkaz skoku:

  `goto` *návěští* **`;`**
  - návěští je identifikátor
  - použití: výskok z více úrovní vnoření cyklu apod.
  - lze skákat dovnitř strukturovaných příkazů - nedoporučuje se
- Příkaz s návěštím:

  návěští : *příkaz*
  - návěští jsou lokální ve funkci

```
for (i = 1; j < 10; i++)
  for (j = 1; j < 20; j++) {
    …  if (podm) goto konec; …
  }
konec: ;
```

- Příkaz **`return`**:

  `return` *výraz* **`;`** nebo **`return;`**
  - ukončí funkci a definuje návratovou hodnotu

The End