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 a Martin Mazanec

Programování v C++, A7B36PJC

09/2015, Lekce 2

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

Reprezentace dat

- Programy pracují s daty.
- Ta musí být programu přístupná – musí být uložena v paměti počítače.
- V imperativních jazycích k tomu máme konstanty a proměnné.
- Způsob uložení dat závisí na počítači, operačním systému a programovacím jazyce.
- Přesná reprezentace dat není v C a C++ určena striktně, ale jsou stanovena pravidla, která musí každá implementace dodržet.
- Zápis konstant základních datových typů nazýváme **literály**.
- Ten je (na rozdíl od reprezentace) určen striktně.

Lexikální elementy

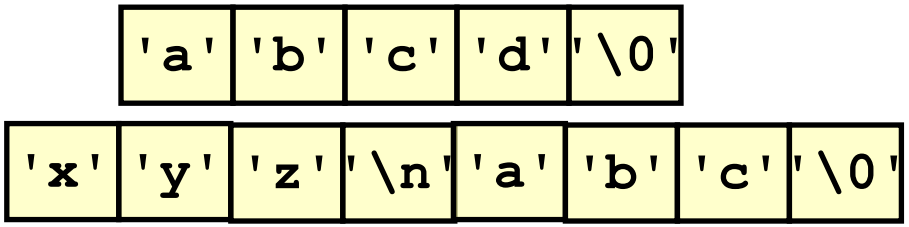
- **identifikátory**
  - písmena (rozlišuje se velikost), číslice, `'_'`
  - délka není omezena, rozlišují se podle úvodních znaků v závislosti na implementaci (ANSI doporučuje nejméně 31 znaků)
  - některé jsou rezervovány jako klíčová slova
- **klíčová slova**
  - `if while` … malými písmeny!
- **oddělovače** a **operátory**
  - `{ } ; + - & && <<=`
- **číselné literály (zápisy čísel)**
  - `125` `1'234'567'893'234` `125.45e-7` `0.12E3`
  - oktalové: `037` hexadecimální: `0x1F` binární: `0b101011`

Lexikální elementy (pokračování)

- **řetězce**

  `"abcd"  ""  "xyz\nabc"`

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



- **znakové literály (zápisy znaků)**

```
'x'  '\n'  '\t'  '\0'  '\\'  '\''  '\014' '\u010C',
wchar_t c = L'\u79c1'; /* Unicode character */
const char* str = "\xEC\xA4\x91"; //UTF-8 character string
```

- **komentáře**

```
/* toto je komentář */
// toto je komentář v C++ a v novějších normách C (C99+)
```

Typy dat

Jednoduché typy

- **celočíselné** (pevná řádová čárka)
- **reálné** (pohyblivá řádová čárka)
- **logické** (chybí v C89)
- **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` (šířka je závislá na procesoru a systému)

  ```
  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
  long long = long long int
  unsigned long long = unsigned long long int
  short <= int <= long
  ```

- ***pohyblivá řádová čárka:***

  `float, double, long double`

- ***logické hodnoty (chybí v C89):***

  `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 typů uložené za sebou, implicitně přístupné
- **Třída**
  - obsahuje pojmenované položky různých typů uložené za sebou, implicitně nepřístupné zvenku
- **Union** (sjednocení)
  - pojmenované položky různých typů uložené „přes“ sebe

Funkce a ukazatele

- **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 + reálné
  - skalární typy: aritmetické + ukazatele

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
- reálné typy (reprezentace v pohyblivé řádové čárce) podle normy IEC 60559:1989 (ANSI/IEEE)

Velikost vnitřní reprezentace:	**Clang++3.3, 64bit Linux**	**VC++ 12 64bit**
char, signed char, unsigned char	1B	1B
short, unsigned short	2B	2B
int, unsigned int	4B	4B
long, unsigned long	8B	4B
long long, unsigned long long	8B	8B
float	4B	4B
double	8B	8B
long double	16B	8B
Platforma	*I32LP64*	*IL32P64*

I32LP64: Integer má 32 bitů, Long a Pointer mají 64 bitů

Jak to zjistit?

```
#include <iostream>

int main() {
    std::cout << "char: " << sizeof(char) << '\n'
          << "short: " << sizeof(short) << '\n'
          << "int: " << sizeof(int) << '\n'
          << "long: " << sizeof(long) << '\n'
          << "long long: " << sizeof(long long) << '\n'
          << "float: " << sizeof(float) << '\n'
          << "double: " << sizeof(double) << '\n'
          << "long double: " << sizeof(long double) << '\n';
}
```

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 první znak (typu `char*`).
- V C++ existuje typ `std::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 = 2;
extern pole[10];
static long double z;
```

Paměťová třída

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

Paměťová třída popisuje způsob přidělování paměti:

nic znamená, že paměť je přidělena automaticky
static paměť přidělena staticky během překladu
extern nepřidělovat paměť

- Implicitně je paměťová třída:

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

- Automatické proměnné jsou alokovány na zásobníku, statické zpravidla v datovém segmentu programu. Statické existují po celou dobu výpočtu, automatické jen po dobu zpracování bloku, ve kterém jsou lokální.

Cvičení: Funkce, která si počítá volání

```
// vypíše: Toto je volání číslo N.
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
mutable	vzácné, třídní proměnné které mohou být modifikovány na const objektu

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

Specifikace typu

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

Specifikace typu je:

- **`void`** zastupuje „žádný“ typ (prázdná množina hodnot)
- *identifikátor typu* (základního nebo uživatelem definovaného)
- popis struktury (**`struct`**), popis třídy (**`class`**), popis sjednocení (**`union`**), nebo výčtového typu (**`enum`**)
- **`auto`** typ je doplněn kompilátorem jako typ na pravé straně výrazu

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 s 10 prvky 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` *typ alias*
`using` *alias* = *typ*;

Příklady:

```
typedef unsigned char Byte;
using Byte = unsigned char;

typedef unsigned short int Word;
using Word = short int;

typedef int Matice[10][10];
using Matice = int[10][10];
```

Možné deklarace:

```
static Word *ptr;
const Matice m = {{ /**/ }};
```

```
const Byte b = 0xBE;
extern Word w;
```

Inicializace

- Datové objekty (proměnné, konstanty) se inicializují v době vzniku
  - statické a globální na začátku programu
  - ostatní v době deklarace (na začátku funkce, bloku), pokud jsou statické, tak jen poprvé
- 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 = 10.2F;
int B = 123;
```

Inicializace v C++

- Lze použít stejné tvary jako v C.
- Deklarace s explicitní inicializací může mít tvar:

  typ deklarátor (*výraz*); nebo
  typ deklarátor {*výraz*}; (C++11)

```
int a = 5;              // počáteční hodnota: 5
int b(3);               // počáteční hodnota: 3
int c{ 2 };             // počáteční hodnota: 2
int result;             // počáteční hodnota není určena
```

- V deklaraci s explicitní inicializací můžeme použít automatické odvození typu:

```
int a = 0;
auto b = a;  // totéž jako: int b = a;
decltype(a) c;  // totéž jako: int c;
```

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 vstupních 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
- makro – od direktivy `#define` do konce zdrojového souboru nebo direktivy `#undef`

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) specifikováno normou, ale implementací !
- Některé operace mají vedlejší efekt (inkrementace, dekrementace, přiřazení) => v jednom výrazu nad jedním objektem max. jeden!

```
i = i++; // Chyba
a = ++i + i++; // Chyba
```

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***)
  - pole prvků typu *T* je konvertováno na 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 následně se převedou na stejný typ, dle tabulky

operand typu … zůstane	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
12U     unsigned int
12L     long
12ULL   unsigned long 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`	`1` je konvertována 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
  - Celočíselné dělení zaokrouhluje k 0
- Operace `%` je zbytek po dělení, musí mít operandy celočíselné, a je definována takto:
  - `x == (x / y) * y + x % y`
  - Pozor, výsledek může být záporný
- 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

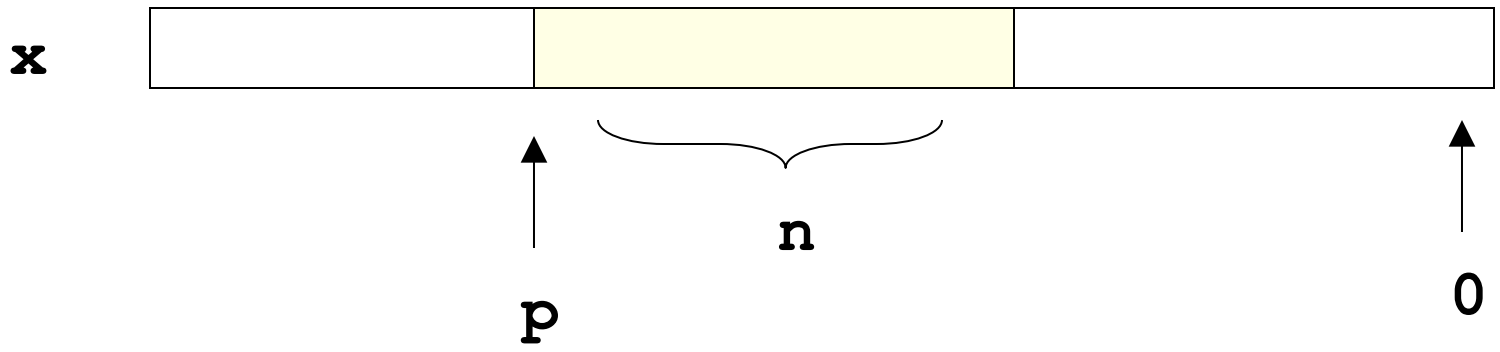
!	**logická negace** - výsledek 1, má-li operand hodnotu 0, jinak je výsledek 0
&&	**logický součin** - výsledek 1, jsou-li oba operandy nenulové, jinak 0
||	**logický součet** - výsledek 1, je-li alespoň jeden operand nenulový, jinak 0

Bitové operace

- Operandy jen celočíselné, provádějí se běžné aritmetické konverze
- Unární:
  - `~` negace jednotlivých bitů
- Binární:
  - `&` logický součin jednotlivých bitů
  - `|` logický součet jednotlivých bitů
  - `^` logický *xor* jednotlivých 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




```
#include <cstdint> // Pro typy s fixní velikostí (zde 32 bitů)

uint32_t getbits(uint32_t x, uint32_t p, uint32_t n)
{
        return (x >> (p - n + 1)) & (~((~0) << n));
}
```

Inkrementace a dekrementace

- Unární operace s vedlejším efektem
- Lze zapsat prefixově nebo postfixově
- Mění 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
- Přiřazení má hodnotu a vedlejší efekt
- Hodnotou výrazu je přiřazená hodnota (***X***)
- Pozor: plete se = a == !
- Přípustné typy operandů:
  - levá a pravá strana jsou téhož skalárního typu nebo téhož typu `struct, class` 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
reálný	kratší reálný	zaokrouhlení mantisy
reálný	delší reálný	doplnění mantisy nulami
reálný	celočíselný	odseknutí necelé části
celočíselný	reálný	možná ztráta přesnosti
celočíselný	kratší celočíselný	odseknutí vyšších bitů
celočíselný unsgn.	delší celočíselný	doplnění nulových bitů
celočíselný sgn.	delší celočíselný	rozšíření znaménka
nullptr	T *	
T * nebo nullptr	void *	opačné přiřazení musí být exlicitní

Selekce

- Zpřístupnění položky struktury, třídy nebo unionu

  X . položka kde *X* je výraz typu **`struct`**, **`class`** nebo **`union`**

- Příklad:

```
struct C { int a; char b; };
C x;
x.a = 1;
x.b = 'T';
```

Ternární podmíněný operátor

- 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 jeden příslušný výraz (podle hodnoty podmínky)
- Příklad:

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

Operátor „čárka“ (comma)

- Umožňuje několik výrazů 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--);
```

- Přetypování může být složitější, pokud se jedná o objekty, jejichž typ se může měnit (např. ukazatel na nějakou hierarchii tříd). Pak je nutno rozlišovat, zda má být přetypování statické, či dynamické a je nutno používat operátory static_cast<typ>, příp. dynamic_cast<typ>:

```
q = static_cast<char*>(p);
```

Priorita a asociativita operátorů

[] () . -> postfix++ postfix--	Zleva-doprava
prefix++ prefix-- sizeof & * + - ~ !	Zprava-doleva
(typ) - 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.

```
int pole[10];      // 10 je konstanta
int pole[5 + 7];   // soucet konstant je konstanta
const int velikost = 20;
int pole[velikost]; // v poradku, velikost je const
int velikost = 20;
int pole[velikost]; // chyba, velikost neni const
```

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 příkazů a deklarací*** }

  - 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;
        }
        cout << 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 C++11 existují další tvary cyklů pro kontejnery:

  `for` (***deklarace : rozsah***) ***příkaz***

Příklad

```
// Cyklus for řízený rozsahem (range-based)
#include <iostream>
#include <string>

int main()
{
	std::string str{ "Hello!" };
	for (char c : str)
	{
		std::cout << "[" << c << "]";
	}
	std::cout << '\n';
}

// typ proměnné v deklaraci lze nechat odvodit
	for (auto c : str)
```

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("nedostatečně");
}
```

Chybně

```
int zn = 2;

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

Správně

The End