1. Základní prvky jazyka C/C++

Ing. Peter Janků, Ph.D.
Ing. Michal Bližňák, Ph.D.

Ústav informatiky a umělé inteligence
Fakulta aplikované informatiky
UTB Zlín

Programování v jazyce C++, Zlín, 23. září 2021

Základní vlastnosti programovacího jazyka C++

Základní vlastnosti programovacího jazyka C++

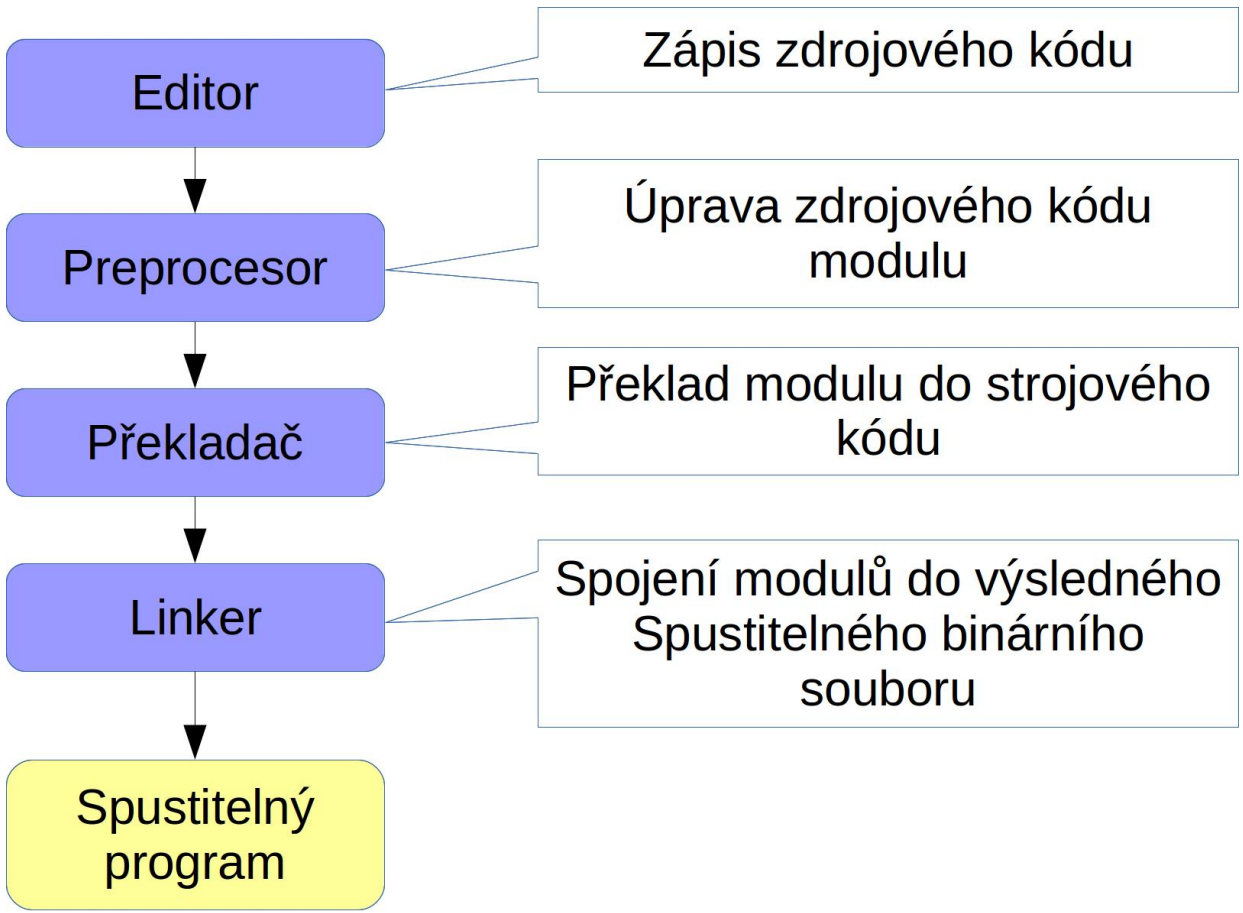
- C++ je objektová nadstavba jazyka C (++ jsou zástupné symboly pro sousloví "with classes")
- Objekty lépe modelují reálný svět a vazby v něm
- C++ zpřehledňuje zápis zdrojového kódu
- Umožňuje využití moderních přístupů při vývoji SW

Obrázek: Bjarne Stroustrup, tvůrce jazyka C++

Standardizace programovacího jazyka C

- Velké množství dialektů
- Snaha o standardizaci započala v roce 1989 (ANSI C 89)
- Proces standardizace pokračuje dodnes (ANSI C 99, C++11, C++14, c++17,c++20,??c++23?? ...)
- Implementace překladačů vždy několik (i mnoho) let za standardem
  - GCC
  - MSVC
  - Clang
  - a další...
- Kvalita implementace překladačů se značně liší dle dodavatele...

Pripcip práce překladačů jazyka C/C++



Obrázek: Proces překladu zdrojových kódů

Rozšíření C++ oproti C o neobjektové vlastnosti

Datové typy

Datový typ	Klíčové slova jazyka C++
celá čísla	`short`, `int`, `long`, *enumerace*
reálná čísla	`float`, `double`
znak	`char`[1]
logická hodnota	`boolean`[2]
prázdná hodnota	`void`
automatická hodnota	`auto`[3]
textový řetězec	`string`[4]

Tabulka: Základní datové typy

C++ umožňuje vytvářet uživatelem definované datové typy.

[1] Defakto také celočíselný typ; ordinální hodnota znaku v ASCII tabulce
[2] V jazyce C deklarováno hlavičkou stdbool.h
[3] Od normy C++11
[4] Pouze v C++. Reprezentováno třídou.

Přetěžování funkcí

Jazyk C++ umožňuje přetížit funkce a operátory.

- Přetížením operátoru je možné změnit jeho výchozí chování v rámci třídy.
- Přetížení funkce znamená možnost definovat více funkcí se stejným jménem.
  - může se měnit typ a počet parametrů
  - může se měnit návratový typ funkce
  - funkce musí být jednoznačně identifikovatelné
    - shoda v typech
    - shoda v rozšíření typů
    - shoda v typech po implicitní konverzi

Přetěžování funkcí

Listing 1: Příklady přetížení funkce

```
1  int vypocetObsahu(int a, int b){
2      return a*b;
3  }
4  double vypocetObsahu(double a, double b){
5      return a*b;
6  }
7  int main()
8  {
9      cout << "Obsah obrazce je: " << vypocetObsahu(10,10);
10     cout << "Obsah obrazce je: " << vypocetObsahu(9.9,9.9);
11     return 0;
12 }
```

Výchozí parametry

- C++ umožňuje u parametrů definovat jejich výchozí hodnotu
  - Tato hodnota se použije, pokud není jiná hodnota přiřazena při volání funkce.
  - Výchozí hodnoty je možné udávat u parametrů od konce.
  - Výchozí hodnoty nelze v rámci pořadí parametrů kombinovat s parametry bez výchozí hodnoty.
  - Použití hodnoty při volání funkce u parametrů s výchozí hodnotou je dobrovolné.

Listing 2: Příklad použití výchozích parametrů

```
1  int nasobek(int cislo, int n = 2){
2      return cislo*n;
3  }
4  int main()
5  {
6      cout << "Nasobek je: " << nasobek(10,2);
7      cout << "Nasobek je: " << nasobek(10);
8      return 0;
9  }
```

Alokace a dealokace paměti

- Klíčové slovo *new*
  - lze použití pro dynamickou alokaci paměti jako např. malloc (vrací ukazatel na přidělenou paměť)
  - alokace probíhá pro zmíněný objekt/datový typ (není nutné udávat velikost)
  - lze použít pro alokaci polí
  - v případě alokace paměti pro třídu korektně volá konstruktor
  - v případě nedostatku paměti nebo chyby tvorby projektu vrátí prázdný ukazatel
- Klíčové slovo *delete*
  - maže paměť přidělenou pomocí *new*
  - lze využít i pro mazání paměti přidělené poli
  - při mazání paměti třídy zavolá destruktor

Listing 3: Příklad práce s pamětí pomocí new/delete

```
1  int * test = new int;
2  *test = 10;
3  delete test;
4
5  double * pole = new double [10];
6  pole[0] = 10;
7  delete [] pole;
```

Reference - odkazy

- odkazy používají symbol & před jménem proměnné
- odkaz vytvoří něco jako alternativní název proměnné
- lze jej použít pro předávání proměnných do funkce tak aby bylo možné je změnit
- lze vytvářet konstantní odkazy
- při předání do fce musí odkazovat na validní místo v paměti, ne na runtime hodnotu
  - nelze změnit jejich hodnotu
- odkaz je na první pohled podobný ukazatelům ALE
  - odkaz je možné přiřadit pouze jednou - nelze jej změnit tak aby odkazoval na jinou proměnnou
  - na odkaz nelze aplikovat jednoduchou ukazatelovou aritmetiku

Listing 4: Příklad použití odkazu

```
1  #include <iostream>
2
3
4  int main()
5  {
6      int a = 10;
7      int &test = a;
8      test = 20; //provede se zmena promenne a
9      int b = 20;
10     test = b; //NEZMENI ODKAZ
11     //znamena ve skutecnosti a = b;
12     &test = b; // hlasi chybu kompilace
13     return 0;
14 }
```

Listing 5: Příklad použití odkazu jako parametru funkce

```
1  void swap (int & a, int & b){
2      int temp = a;
3      a = b;
4      b = temp;
5  }
6  int promenaA = 10;
7  int promenaB = 20;
8  swap(promenaA,promenaB);
9  std::cout << "Promena A: " << promenaA << " Promena B: " <<
       promenaB << std::endl;
10 swap(10,20); // spatne, nelze pouzit konstantni hodnoty
```

Jmenné prostory (obory názvů)

- funkce a třídy se určují názvy - nelze mít v jednom projektu definovány dvě třídy/funkce stejného názvu (vyjma přetížení)
- ke kolizi může docházet i při použití knihoven 3. stran
- C++ podporuje vytvoření jmenného prostoru
  - Jmenný prostor je definovaný svým specifickým jménem
  - Toto jméno slouží jako jakýsi prefix pro obsažené proměnné/funkce/struktury/...
  - Jmenný prostor může být obsažen ve více souborech (např.std)
  - Členy jmenného prostoru lze deklarovat pouze uvnitř jeho těla
  - Členy jmenného prostoru lze definovat i mimo jeho tělo
  - Jmenné prostory lze vzájemně vnořovat

Jmenné prostory (obory názvů)

- klíčové slovo *using*
  - je možné jej použít pro globální použití jmenného prostoru **v rámci daného kontextu**
  - je možné jej použít pro zviditelnění konkrétního člena kontextu **v rámci daného kontextu**

Listing 6: Příklad použití jmenného prostoru

```
1  namespace MojeFunkce {
2      int globalniPromenaContextu = 10;
3      int sum(int a, int b){return a+b; } // funkce definovana
           unitr kontextu
4      int div(int a, int b);
5  }
6  int MojeFunkce::div(int a, int b){
7      return a/b;
8  }
9  int vysledek = MojeFunkce::sum(10,20);
10 int podil = MojeFunkce::div(10,2);
11 MojeFunkce::globalniPromenaContextu = 20;
12
13 using MojeFunkce::globalniPromenaContextu;
14 globalniPromenaContextu = 30;
15 using namespace MojeFunkce;
16 int vysledek2 = sum(10,20);
```

Direktiva `#include`

- `#include`[5]
  - Vloží specifikovaný soubor do místa volání direktivy
  - `#include <filename>`
    - Hledá **standardní** knihovnu
  - `#include "filename"`
    - Hledá **uživatelský soubor** nejprve v aktuálním adresáři, poté standardní knihovnu
    - Lze specifikovat **místa hledání** (adresáře)
    - Absolutní i relativní cesty
  - Použití pro aplikace s více zdrojovými kódy
  - Obecné deklarace a rozhraní knihoven
  - Jedna nebo více direktiv v každém souboru

[5] Od standardu c++20 zavedeny moduly a *import*, *export* příkazy.

.

Příklady použití direktivy #include

Listing 7: Příklady použití

```
1 #include <stdio.h>
2 #include <vector>
3 #include "main.h"
4 #include "impl.cpp"
5 #include "/home/user/project/lib.h"
```

Direktiva `#define` - symbolická konstanta

- `#define`
  - **Symbolická konstanta**
    - Všechny její výskyty budou nahrazeny specifikovaným obsahem před překladem
  - Syntaxe:

```
#define <constant> <replacement>
```

    - Vše napravo od názvu je použito při nahrazení
  - Existující konstantu **nelze předefinovat**
  - Existující konstantu **lze zrušit**

Příklady použití direktivy #define - symbolická konstanta

Listing 8: Příklady použití

```
1 /* "PI" replaced with "3.14159267" */
2 #define PI 3.14159267
3
4 /* "PI" replaced with "= 3.14159267" */
5 #define PI = 3.14159267
```

Direktiva `#define` - makro

- `#define`
  - **Makro**
    - Operace s parametry specifikovanými v době použití
    - Specifikované argumenty jsou nahrazeny v době použití
    - Opět se jedná o prosté nahrazení textu
    - Šetříme režie spojené s voláním funkce → zvýšení rychlosti provedení operace
    - Roste zdrojový i binární kód
  - Syntaxe:

```
#define <macro> (<args>) <replacement-with-args>
```

    - Vše napravo od názvu je použito při nahrazení
  - Existující makro nelze předefinovat
  - Existující makro lze zrušit

Příklady použití direktivy #define - makro

Listing 9: Příklady použití

```
1 /* Define macro */
2 #define CIRCLE_AREA( x ) ( PI * ( x ) * ( x ) )
3
4 /* Usage */
5 auto area = CIRCLE_AREA(4);
6
7 /* Expanded form */
8 auto area = ( 3.14159267 * ( 4 ) * ( 4 ) );
```

Specifika použití maker

- **Pozor na závorky !!!**
- Chybějící závorky **mohou** změnit prioritu zpracování operandů

Listing 10: Příklady chybného použití

```
1  /* Define macro */
2  #define CIRCLE_AREA( x ) ( PI *  x  *  x  )
3
4  int a = 1;
5
6  /* Usage */
7  auto area = CIRCLE_AREA(a + 3);
8
9  /* Expanded form with different meaning... */
10 auto area = ( 3.14159267 * a + 3 * a + 3 );
```

Vícenásobné argumenty makra

- Makro zle definovat s více argumenty
- Vícenásobné argumenty oddělujeme čárkami

Listing 11: Příklady použití vícenásobných argumentů

```
1  /* Define macro */
2  #define RECTANGLE_AREA( x, y )  ( ( x ) * ( y ) )
3
4  int a = 1;
5
6  /* Usage */
7  auto area = RECTANGLE_AREA( a + 4, a + 7 );
8
9  /* Expanded form with different meaning... */
10 auto area = ( ( a + 4 ) * ( a + 7 ) );
```

Direktiva #undef

- `#undef`
  - Ruší existující symbolickou konstantu nebo makro
  - Zrušenou symbolickou konstantu nebo makro lze později předefinovat

Speciální direktivy a operátory

Speciální direktivy a operátory

- Direktiva

```
#error <message>
```

  - Vypíše chybovou hlášku v době překladu
  - Ukončí překlad s chybou

- Direktiva

```
#pragma <directive> <arguments>
```

  - Specifické příkazy preprocesoru překladače
  - Mohou být podporovány pouze vybranými překladači
  - Nepodporované direktivy jsou ignorovány

Operátory # a

- Operátor #
  - Nahradí argument makra řetězcem v uvozovkách
- Operátor ##
  - Spojí dva argumenty do jednoho řetězce

Předdefinované symbolické konstanty

- Předdefinované symbolické konstanty nahrazené příslušnou hodnotou
- Nelze použít jako operand direktiv `#define` nebo `#undef`

Konstanta	Význam
__LINE__	Aktuální číslo řádku
__FILE__	Aktuální jméno překládaného souboru
__DATE__	Aktuální datum
__TIME__	Aktuální čas

Streamový vstup a výstup v C++

Streamový vstup a výstup v C++

- Jazyk C++ nabízí **alternativu** ke standardnímu formátovanému vstupu a výstupu (`scanf` a `printf`)
- Implementace pomocí **šablonových tříd** a **přetížených operátorů**
- Knihovna `<iostream>` a jmenný prostor `std`
- Možnost standardního přesměrování
- **Výstup**
  - Standardní výstup `std::cout`
  - Chybový výstup `std::cerr`
  - Nový řádek `std::endl`
- **Vstup**
  - Standardní vstup `std::cin`
  - Automatické ošetření neplatných konverzí

Příklad použití I/O streamů jazyka C++

Listing 12: I/O streamy jazyka C++

```
1  #include <iostream>
2
3  int main(int argc, char** argv)
4  {
5      int a = 0;
6
7      std::cout << "Enter integer value: " << std::endl;
8      std::cin >> a;
9      std::cout << "Entered value is " << a << std::endl;
10
11     return 0;
12 }
```

Příklad použití I/O streamů jazyka C++

Listing 13: Výstup z programu

```
1 Enter integer value:
2 100
3 Entered value is 100
```

Další rozšíření jazyka C++

Tyto rozšíření budou diskutovány na dalších přednáškách

- Tvorba tříd a objektů
- Dědičnosti tříd a objektů
- Využití vyjímek pro ošetření chyb
- Využití lambda funkcí
- a další

Děkuji za pozornost

A to je pro dnešek vše. Nastává čas pro vaše dotazy...