Funkce v C++

Petr Šaloun

3. listopadu 2003

Funkce
– základní stavební kámen procedurálního programu,
– nemá být rozsáhlá,
– řeší ucelený problém,
– je-li problém příliš složitý, volá na pomoc další funkce,

– může vracet návratovou hodnotu,
– může mít argumenty.

Každý C++ program:
[void , int] main(). . .

obecně:

```
typ jmeno(formalni argumenty) {
    telo funkce
}
```

definice funkce – identifikátor, typ návratové hodnoty, typ a názvy argumentů (mezi (a)) a tělo funkce (mezi { a }) – kód, který bude při každém volání funkce proveden ({ a }) vs. *deklarace funkce*.

() vs. void

formální argumenty vs. *skutečné argumenty*

volání funkce

```
jmeno(skutecne argumenty)
```

skutečný argument – identifikátor proměnné, konstanty, výraz, přímo uvedená hodnota konstanty.

Počet *skutečných argumentů* funkce je dán potřebou programátora:

() – nula skutečných argumentů,
(. . .) – proměnným počet argumentů,
bez závorek – adresa funkce (vstupní bod funkce).

návratová hodnota funkce

```
return vyraz_vhodneho_typu;
```

příklad:

```
int isqr(int i) {
 return i * i;
}
...
 int vysledek;

 vysledek = isqr(4);
```

Argumenty funkcí a způsob jejich předávání

- hodnotou – kopie skutečných arg. na zásobník, změna formálních argumentů jen ve funkci,
- adresou – ukazatel realizuje vazbu mezi formálním a skutečným a.,
- odkazem – vazbu mezi s.a. a f.a. realizuje překladač.

adresový operátor & a dereference $*$

```
pocatecni stav
skutecny1 = 123, skutecny2 = -456

po volani fce 'zamenit_hodnotou_nejde()'
skutecny1 = 123, skutecny2 = -456

po volani fce 'zamen_ukazatelem()'
skutecny1 = -456, skutecny2 = 123

po volani fce 'zamen_odkazem()'
skutecny1 = 123, skutecny2 = -456
```

```
// fn-argumenty.cpp

#include <iostream>

using namespace std;

void zamenit_hodnotou_nejde(int fa, int fb) {
 int pomocna = fa; // hodnota
 fa = fb;
 fb = pomocna;
}

void zamen_ukazatelem(int *fa, int *fb) {
 int pomocna = *fa; // dereference
 *fa = *fb;
 *fb = pomocna;
}

void zamen_odkazem(int& fa, int& fb) {
 int pomocna = fa; // odkaz
 fa = fb;
 fb = pomocna;
}
```

```
void main() {
 int sprvni= 123, sdruhy = -456;
 cout << "pocatecni stav" << endl;
 cout << "sprvni= " << sprvni<< ", sdruhy = "
      << sdruhy << endl << endl;

 zamenit_hodnotou_nejde(sprvni, sdruhy);
 cout << "po 'zamenit_hodnotou_nejde()'"
      << endl;
 cout << "sprvni= " << sprvni<< ", sdruhy = "
      << sdruhy << endl << endl;

 zamen_ukazatelem(&sprvni, &sdruhy);
 cout << "po 'zamen_ukazatelem()'" << endl;
 cout << "sprvni= " << sprvni<< ", sdruhy = "
      << sdruhy << endl << endl;

 zamen_odkazem(sprvni, sdruhy);
 cout << "po 'zamen_odkazem()'" << endl;
 cout << "sprvni= " << sprvni<< ", sdruhy = "
      << sdruhy << endl;
} // void main()
```

paměť. třída	výklad
auto	umístění na zásobník, neinicializované;
extern	nevytvářet, bude připojen z jiného modulu;
register	přání umístit do registru procesoru, neinicializované;
static	umístění do datového segmentu, inicializované nulou;

modifikátor	význam
const	vyjadřuje neměnitelnost;
volatile	vyjadřuje neustálou proměnnost – nekešovat.

objekt	paměťová třída, výklad, umístění
glob. prom.	static a extern, inicializovány nulou, DS;
lokální prom.	auto, neinicializovány, zásobník;
formální arg.	auto, neinicializovány, zásobník;
def. fce	extern, definice dle kódu, CS

Rekurze

Formální argumenty funkce i její lokální proměnné se umisťují na **zásobník** – LIFO *Last In First Out – poslední dovnitř, první ven*.

$n! = n \times (n-1)!$
$0! = 1$.

```
zadej prirozene cislo n:5
5! = 120

zadej prirozene cislo n:69
69! = 1.71122e+098
```

```
/*********************
 * soubor fact-r.cpp
 * faktorial rekurzi
 *********************/

#include <iostream>

using namespace std;

double fact(long n) {
 if (n == 0L)
   return 1.0L;
 else
   return n * fact(n-1);
} // double fact(long n)

void main() {
 static long n;
 cout <<"zadej prirozene cislo n:";
 cin >> n;
 cout << n << "! = " << fact(n) << endl;
} // void main()
```

```
// soubor fact-rst.cpp
#include <iostream>
using namespace std;
double fact(long n) {
 static int h;
 double navrat;
 cout << "hloubka= " << ++h << "\tn= "
      <<n << endl;
 if (n == 0L)
   navrat = 1.0L;
 else
   navrat = n * fact(n-1);
 cout << "hloubka= " << h-- <<"\tn= "
      <<n << "\tnavrat= " << navrat << endl;
 return navrat;
}
void main() {
 long n;
 cout <<"zadej prirozene cislo n:";
 cin >> n;
 cout << n << "! = " << fact(n) << endl;
}
```

```
zadej prirozene cislo n:5
hloubka= 1      n= 5
hloubka= 2      n= 4
hloubka= 3      n= 3
hloubka= 4      n= 2
hloubka= 5      n= 1
hloubka= 6      n= 0
hloubka= 6      n= 0    navrat= 1
hloubka= 5      n= 1    navrat= 1
hloubka= 4      n= 2    navrat= 2
hloubka= 3      n= 3    navrat= 6
hloubka= 2      n= 4    navrat= 24
hloubka= 1      n= 5    navrat= 120
5! = 120
```

Hlavičky v C++ původních knihoven jazyka C.

<cassert> <ciso646> <csetjmp> <cstdio> <ctime>
<cctype> <climits> <csignal> <cstdlib> <cwchar>
<cerrno> <clocale> <cstdarg> <cstring> <cwctype>
<cfloat> <cmath> <cstddef>

Deklarace funkce – *prototyp*:

<algorithm> <iomanip> <list> <ostream> <streambuf>
<bitset> <ios> <locale> <queue> <string>
<complex> <iosfwd> <map> <set> <typeinfo>
<deque> <iostream> <memory> <sstream> < utility >
<exception> <istream> <new> <stack> <valarray>
<fstream> <iterator> <numeric> <stdexcept> <vector>
<functional> <limits>

deklarace funkce – informace pro překladač,

definice funkce má *paměťové nároky*.

identifikátory form. arg. stačí uvést až při definici funkce.

ISO C++ vyžaduje prototyp každé funkce, kterou chceme použít.

Přetěžování identifikátoru funkce:
více funkcí se stejným identifikátorem – při volání rozhoduje počet argumentů, nebo typ argumentů.

Příklady: „Rozsviť!“, tiskni ().

Přetížené operátorové funkce

Přetížit můžeme všechny operátory jazyka C++, kromě:

. .* :: ?:

C++ chápe operátor jako funkci s jedním či dvěma argumenty (pro unární resp. binární operátory).

Alespoň jeden z operandů musí být objektového typu (instancí třídy) definovaný pomocí class (nebo struct).

Přetížit můžeme unární i binární operátory.

Přetížení zachovává původní prioritu i asociativitu operátoru. U unárních operátorů ++ a −− lze rozlišovat jejich prefixový či postfixový zápis.

Přetížení **new a delete** umožňuje variantní alokaci či dealokaci volného paměťového prostoru (haldy). Přetížený operátor new musí

- vracet ukazatel typu void $*$,
- mít první argument typu size_t.

Přetížený operátor delete

- nesmí mít žádnou návratovou hodnotu (ani její určení jako void),
- musí mít ukazatel typu void $*$ jako svůj první argument,
- může mít druhý argument typu size_t (jako nepovinný).

možnosti přetížení operátorů:

- operátor funkčního volání `()`:
  zápis `x(arg1, arg2);`
  odpovídá `x.operator() (arg1, arg2);`

- operátor přístupu do pole `[]`:
  zápis `x[y];`
  odpovídá `x.operator[] (y);`

- operátor členského přístupu $->$:
  zápis `x->m;`
  odpovídá `(x.operator->())->m;`

- operátor přiřazení `=` musí být deklarován jako členská metoda :
  `X& operator= (X& x);` s možnými modifikátory const a volatile
  Přiřazení úzce souvisí s kopírovacím konstruktorem!

Přetížení unárních operátorů

```
// soubor opf-un.cpp

#include <iostream>

using namespace std;

const int mez = 12345;

class Citac {
  unsigned int hodnota;
 public:
  Citac(void) {hodnota = 0;}
  void operator++(int) {          // postfix
    if (hodnota < mez) hodnota++;
  }
  void operator--() { // prefix
    if (hodnota > 0) hodnota--;
  }
  unsigned int operator()() {
    return hodnota;
  }
}; // class Citac
```

```
void main() {
  Citac citac;
  for (int i=0; i < 5; i++)
    citac++; // Citac.operator++()
  citac--; // varovani
  cout << "hodnota je " << citac() << endl;
  // Citac.operator() ()
}
```

Implicitní hodnoty argumentů funkce

můžeme přiřazovat argumentům podle jejich pořadí uvedení v deklaraci/definici funkce pouze zezadu.

```
//deklarace:
  int fn(int i = 1, int j = 2);// O.K.
  int fn(int i = 1, int j);//chyba, neni odzadu

//volani:
  fn();      // jako fn(1, 2)
  fn(5);     // jako fn(5, 2)
  fn(7, 8); // jako fn(7, 8), ne implicitni ho

//definice
  int fn(int i, int j) {...
 // O.K. implicitni hodnoty z deklarace

  int fn(int i, int j = 8) {...
 // nelze, implicitni hodnoty i v deklaraci
```

Inline funkce

optimalizace na rychlost nebo na velikost,
modifikátor inline.

- Opakované „volání“ rozhodně **zvětší výslednou délku programu**.

- Umístěním kódu inline funkce namísto volání se *sníží časová režie systému* – **zrychlení aplikace**, nemusí se:
  – kopírovat skutečné argumenty,
  – předávat řízení chodu programu na jiné místo,
  – po vykonání vrátit na správnou adresu zpět,
  – uklízet již nepotřebné kopie skutečných argumentů.

deklarace funkce

```
typ jmeno(seznam_argumentu);
```

- typ je typ návratové hodnoty funkce
- jmeno je identifikátor, který funkci dáváme,
- () je povinná dvojice závorek, vymezující deklaraci argumentů,
- seznam_argumentu je nepovinný (žádné, jeden nebo více argumentů), implicitní hodnoty argumentů (odzadu), nebo proměnný počet argumentů (. . .).
  Argumenty oddělujeme navzájem čárkami. Každý argument musí mít samostatně určen datový typ.

Deklarace funkce popisuje vstupy a výstupy (*rozhraní funkce*), které funkce poskytuje, ale nedefinuje posloupnost příkazů, které má funkce vykonávat.

Funkce nemá provádět akce s jinými daty, než která jí předáme jako argumenty.

Výstupy z funkce mají probíhat jen jako její návratová hodnota, nebo přes argumenty předávané odkazem či adresou. Jinak „vedlejší účinky“.

Deklarace vlastních funkcí do *hlavičkových souborů* – ne paměťové nároky. Jinak problémy – spojovací program (linker).

```
/*****************************************
 * soubor fn-argn.cpp
 * funkce s ěýpromnnm čpotem ůargument ůýrznch
 *****************************************

#include <iostream>
#include <cstdarg>

using namespace std;

double polynom(short pocet, ...) {
/*****************************************
 * číívysl polynom an*xˆn + an-1*xˆn-1 + ... a1*
 * čpoet = čpoet švech ůkoeficient (n+1), ánsle
 * x typu double, koeficienty typu int
 *****************************************
 double hodnota = 0.0, x;
 va_list ap;

 va_start(ap, pocet);
 x = va_arg(ap, double);
 while (pocet-- != 0)
     hodnota = hodnota * x + va_arg(ap, int);
```

```
  va_end(ap);
  return hodnota;
}

void main(void) {
  double x = 2.0;
  /* pol. st. 2; 2x^2 + 3x + 4, x=2 -> 18 */
  double f = polynom(3, x, 2, 3, 4);
  cout << "p(" << x << ") = " << f << endl;

  x = 3.0;
  /* pol. st. 3; x^3, x=3 -> 27 */
  f = polynom(4, x, 1, 0, 0, 0);
  cout << "p(" << x << ") = " << f << endl;

  x = 5.0;
  /* pol. st. 4; 2x^4 - 10x^3 + 2x^2 + 3x - 1, x
  f = polynom(5, x, 2, -10, 2, 3, -1);
  cout << "p(" << x << ") = " << f << endl;
} // void main(void)
```

```
p(2) = 18
p(3) = 27
p(5) = 64
```