Funkce a parametry

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, Martin Mazanec

© Karel Richta, 2015

Programování v C++, A7B36PJC
09/2015, Lekce 4

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

Deklarace a definice funkce

- Funkce obsahuje kód, který je opakovaně využit při každém volání.
- V jazyce C++ rozlišujeme mezi deklarací a definicí funkce.
- **Deklarace funkce** zahrnuje identifikátor funkce, typ výsledku a typy parametrů.
- **Definice funkce** je deklarace a implementace (tělo) funkce.
- Příklad deklarace:

```
long Power(long x, int y);
```

- Příklad definice:

```
long Power(long x, int y)
{
  long result = 1;
  while (y-- >= 0) result *= x;
  return result;
}
```

tělo funkce

Deklarace funkce

- Funkce **může** být deklarována opakovaně – i v rámci jednoho `.cpp` souboru.

```
int fac(int x);

int fac(int x); // OK

int fac(int x) { // OK
    if (x <= 1) {
        return 1;
    }
    return x * fac(x - 1);
}

int fac(int x); // OK
```

Definice funkce

- Funkce **nesmí** být definována opakovaně – ani v rámci více `.cpp` souborů.

soubor1.cpp

```
int fac(int x) {
    if (x <= 1) {
        return 1;
    }
    return x * fac(x - 1);
}
```

soubor2.cpp

```
int fac(int k) {
    int result = 1;
    for (int i = 2; i <= k; ++i) {
        result *= i;
    }
    return result;
}
```

`link error: redefinition of int fac(int)`

Volání funkce

- Aby funkce mohla být zavolána, musí být
  - alespoň jednou deklarována v tomto `.cpp` souboru
  - právě jednou definována v nějakém `.cpp` souboru

fac.cpp

```
int fac(int x) { // definice
    if (x <= 1) {
        return 1;
    }
    return x * fac(x - 1);
}
```

main.cpp

```
#include <iostream>
int fac(int x);  // deklarace
int main() {
    std::cout << fac(6) << "\n";
}
```

Význam hlavičkových souborů

- Deklaraci funkce často umisťujeme do hlavičkového souboru. Umožníme tím použití funkce v celém programu, aniž bychom její deklaraci všude opisovali.

fac.h

```
int fac(int x);
```

fac.cpp

```
int fac(int x) { // definice
    if (x <= 1) {
        return 1;
    }
    return x * fac(x - 1);
}
```

main.cpp

```
#include <iostream>
#include "fac.h" // deklarace
int main() {
    std::cout << fac(6) << "\n";
}
```

Prototyp funkce

- Funkční prototyp (lépe funkční profil) se skládá z
  - typu výsledku,
  - počtu parametrů,
  - typů jednotlivých parametrů funkce.

```
int F(int k);          // F má prototyp int(int)
void G(long x, int y); // G má prototyp void(long, int)
```

- Názvy parametrů nejsou důležité pro volání, v deklaraci je můžeme vynechat.

```
char H(char, char);    // bez jmen parametrů
```

Příklad – použití deklarací

```
#include <iostream>
void odd(int x);
void even(int x);

int main() {
    int i;
    do {
        std::cout << "Please, enter number (0 to exit): "; std::cin >> i;
        odd(i);
    } while (i != 0);
}

void odd(int x) {
    if ((x % 2) != 0) std::cout << "The number is odd.\n"; else even(x);
}

void even(int x) {
    if ((x % 2) == 0) std::cout << "The number is even.\n"; else odd(x);
}
```

Deklarace funkce bez parametrů

- Pozor na odlišnost mezi C++ a C:
- C++

```
int F(void); // funkce F má 0 parametrů
int G();     // funkce G má 0 parametrů
```

- C

```
int F(void); // funkce F má 0 parametrů
int G();     // funkce G má neurčený počet parametrů (!)
```

- Prázdný seznam parametrů v jazyce C znamená libovolný, blíže neurčený počet parametrů.

Parametry a výsledek funkce

- Parametry funkcí jsou nahrazovány **hodnotou** argumentu, nebo **referencí** na argument.
- **Pořadí** vyhodnocení argumentů není určeno. Různé implementace C++ mohou vyhodnocovat argumenty v různém pořadí.
- **Výsledek** (návratová hodnota) funkce může mít libovolný typ. Může to být hodnota aritmetického typu, ukazatel, reference, struktura, třída, union, enumerátor...

Volání funkce

- Syntaxe:

  F (X1, X2, … Xn)

  kde *F* je identifikátor funkce (nebo ukazatel na funkci) a *X1, … Xn* jsou výrazy tvořící argumenty volání.
- Závorky jsou třeba, i když se jedná o funkci bez parametrů.
- Pro každý argument se provádí
  - roztažení (promotion) z menšího číselného typu na větší, pokud je to třeba, např. `char` na `int`, `float` na `double`,
  - konverze (conversion) z většího číselného typu na menší, pokud je to třeba,
  - kontrola přípustnosti typu argumentu pro daný parametr.
- V C++ může existovat více funkcí stejného jména, musí být ale rozlišitelné pomocí počtu nebo typy parametrů (tzv. přetížení funkcí).
- V C existuje vždy pouze jedna funkce daného jména.

Příklady volání funkcí

```
long Power(long, int);
long l; int a, b; float f;
l = Power(l, 10);
a = Power(a + 3, b);
f = Power(f, 3);

void Swap(int *, int *);
int x, y;
Swap(&x, &y);

void CtiPole(int, int *);
int pole[10];
CtiPole(10, pole);
```

Definice funkce

- Syntaxe:

  ***hlavička* { *tělo* }**

  kde hlavička má tvar deklarace funkce: typ návratové hodnoty, jméno funkce a seznam deklarací parametrů.

```
long Power(long x, int z)
{
    long result = 1;
    while (y-- >= 0) result *= x;
    return result;
}
```

- Každý parametr je deklarován zvlášť, nelze např.

```
double vec_len(double x, y, z) // takto ne
```

- Typ návratové hodnoty `void` = funkce nevrací hodnotu.
  Místo seznamu parametrů `void` = funkce bez parametrů.

```
void Dummy(void) { }
```

Parametry funkce volané hodnotou

- Volání hodnotou znamená, že jako skutečné parametry můžeme použít libovolný výraz převeditelný na daný typ. Výraz se vyhodnotí a jeho hodnota se uloží do formálního parametru, který funguje jako lokální proměnná.
- Funkce s parametry volanými hodnotou vrací nejvýše jeden výsledek – funkční hodnotu (zpravidla přes zásobník). Pokud vrací `void` – nevrací nic (je to procedura).
- Pokud chceme, aby funkce vracela více hodnot, musíme využít globální proměnné, nebo použít jako parametry ukazatele, nebo použít parametry volané referencí (viz dále).
- Přes parametry typu ukazatel můžeme manipulovat s objekty mimo lokální prostor těla funkce. To může vyžadovat určitou disciplinu.

Příklad: funkce, která prohodí 2 buňky

```
void swap(int x, int y)
{
   int z;
   z = x;
   x = y;
   y = z;
}

int A, B;
swap(A, B);
```

Chybně

```
void swap(int *x, int *y)
{
   int z;
   z = *x;
   *x = *y;
   *y = z;
}

int A, B;
swap(&A, &B);
```

Správně

Parametry typu reference (C++)

- Standardně jsou parametry předávány hodnotou, tj. vyhodnotí se skutečný parametr a tato hodnota se uloží do odpovídajícího „formálního“ parametru a poté se funkce spočítá.
- Formální parametry v hlavičce funkce jsou vlastně další lokální proměnné, iniciované při volání funkce.
- Pokud chceme, aby funkce něco měnila v prostředí, kde je volána, může jako parametr předat hodnotou ukazatel na daný objekt. Přes tento ukazatel může funkce modifikovat prostředí (proměnné).
- Jiná možnost je použít parametry typu reference (na objekt), kdy se nepředává hodnota objektu, ale reference na něj – jako by skutečný parametr nahradil parametr formální. Skutečný parametr pak ale musí být proměnná (přesněji – může to být výraz nabývající typu l-hodnota).

Příklad: Parametry typu reference

```
// parametry typu reference
#include <iostream>

void duplicate(int& a, int& b, int& c) {
	a *= 2;
	b *= 2;
	c *= 2;
}

int main() {
	int x = 1, y = 3, z = 7;
	duplicate(x, y, z);
	std::cout << "x=" << x << ", y=" << y << ", z=" << z;
	return 0;
}
```

Příklad: swap pomocí parametrů typu reference

```
void swap(int &x, int &y)
{
   int z;
   z = x;
   x = y;
   y = z;
}

int A, B;
swap(A, B);
```

```
void swap(int *x, int *y)
{
   int z;
   z = *x;
   *x = *y;
   *y = z;
}

int A, B;
swap(&A, &B);
```

Konstantní reference

- Parametry volané referencí poslouží pro modifikaci skutečných parametrů, které se za ně dosadí.
- Můžeme je ale používat i za jiným účelem, zejména k tomu, abychom vynechali kopírování argumentů při volání funkce (viz dále).
- Pokud funkce s parametry volanými referencí tyto parametry nemodifikuje, můžeme to explicitně vyznačit pomocí tzv. konstantních referencí:

```
void copy(int &x, const int &y) {
/* x se mění, y nikoliv - jen se kopíruje do x */
          x = y;
}
```

Reference jako proměnná

- Občas se hodí použít referenci jako proměnnou, například pokud budeme opakovaně přistupovat do pole na stejný index – ke stejnému prvku pole:

```
int pole[20];
int &pole5 = pole[5]; /* definice reference s inicializací –
                        zastupuje pole[5] */
pole5 = 1; // totéž jako pole[5] = 1; ale nepočítá se pole[5]
```

- Důležité je, že reference musí být inicializovaná již v místě definice a není možné později znovu určit nebo změnit, na kterou proměnnou reference odkazuje. To je podstatná změno oproti ukazatelům, která činí používání referencí o něco bezpečnější.

```
int *pole5; // definice bez inicializace - u reference nelze
pole5 = pole + 5; // inicializace
*pole5 = 1; // totéž jako pole[5] = 1;
pole5++; // změna ukazatele, u reference nelze
```

Reference jako návratová hodnota

- Referenci lze použít také jako návratovou hodnotu funkce. V definici funkce vrátíme nějakou l-hodnotu, která zůstane platná i po opuštění těla funkce (ne tedy například lokální proměnnou). Ve volajícím kódu lze návratovou hodnotu použít jako běžnou l-hodnotu, takže výsledek funkce lze použít i na levé straně přiřazení.

```
int data[10];

int& vektor(int index) {
   // Tady můžeme ošetřit meze polí.
   return data[index];
}

int main(void) {
   vektor(5) = 7;
   vektor(3) = vektor(5);
   return 0;
}
```

- Kdybychom se pokusili ve funkci **`vektor`** vrátit například číselnou konstantu, došlo by k chybě při překladu funkce **`vektor`**. Stejně tak by selhal překlad **`main`**, kdybychom návratový typ funkce **`vektor`** deklarovali jako běžný **`int`** a nikoli referenci.

Kopírování parametrů

- Parametry typu reference mohou ušetřit nadbytečné kopírování hodnot parametrů.
- U parametrů volaných referencí může skutečným parametrem být pouze proměnná, nikoliv výraz (přesněji – může to být výraz nabývající typu l-hodnotu). Tato proměnná pak zastoupí v těle funkce formální parametr (ten není samostatnou lokální proměnnou).
- Ušetří se tak přesun hodnoty do formálního parametru a rovněž přesun hodnoty zpět po ukončení funkce.

Kopírování parametrů (pokračování)

- Příklad – spojení řetězců:

```
#include <string>
string concatenate(string a, string b) {
        return a + b;
}
```

- Parametry jsou zde volané hodnotou, tj. skutečný řetězec dosazený za parametr a je zkopírován do pracovní lokální proměnné. Podobně pro parametr b.
- U parametrů volaných referencí se místo toho pracuje se skutečným řetězcem:

```
string concatenate(string& a, string& b) {
        return a + b;
}
```

- K přesunům zde nedochází. Protože funkce s parametry volanými referencí mohou modifikovat své parametry, je bezpečnější použít:

```
string concatenate(const string& a, const string& b) {
        return a + b;
}
```

Příklad: rozlišení funkcí dle skutečných parametrů (C++)

```
#include <iostream>

void test(char x) { std::cout << x << '\n'; return; }
void test(int x) { std::cout << x << '\n'; return; }
void test(float x) { std::cout << x << '\n'; return; }

int main(void) {
      test('a');
      test(17);
      test(2.3F);
      return 0;
}
```

Pořadí vyhodnocení parametrů

- Není normou jazyka definováno, implementace to může udělat tak, jak to vyjde efektivněji (pro procesory Intel zpravidla zprava doleva, pro procesory Sparc zleva doprava).
- Musíme programy psát tak, aby na tom nezáleželo – ve skutečných argumentech nepoužívat operace s vedlejšími efekty.

```
#include <cstdio>

void foo(int x, int y) {}

int main() {
   foo(std::printf("foo"), std::printf("bar"))
   return 0;
}
```

Ve Windows vytiskne:
barfoo
Na Linuxu:
foobar

Cvičení: Pořadí vyhodnocení param.

```
#include <iostream>

void f(int x, int y, int z) {}

int p(int x)
{
   std::cout << "Parametr: " << x << "\n"; return x;
}

int main()
{
   f(p(1), p(2), p(3));
   std::cin.get();
   return 0;
}
```

Implicitní hodnoty parametrů (C++)

```
// implicitní hodnoty parametrů
#include <iostream>

int divide(int a, int b = 2) {
   int r;
   r = a / b;
   return (r);
}

int main() {
   std::cout << divide(12) << '\n';
   std::cout << divide(20, 4) << '\n';
   return 0;
}
```

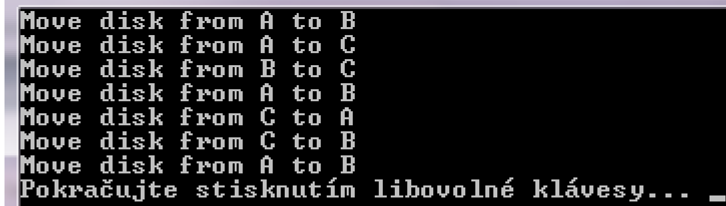
Výsledek:
6
5

Rekurzivní funkce

```
// factorial calculator
#include <iostream>

long factorial(long a) {
	if (a > 1)
		return (a * factorial(a - 1));
	else
		return 1;
}

int main() {
	long number = 9;
	std::cout << number << "! = " << factorial(number);
	return 0;
}
```

Jiný příklad rekurzivní funkce

```
void tower(int disks, char from, char to, char hlp) {
	if (disks <= 1) cout << "Move disk from " << from << " to "
			<< to << endl;
	else {
		tower(disks - 1, from, hlp, to);
		tower(1, from, to, hlp);
		tower(disks - 1, hlp, to, from);
	}
}

int main() {
	tower(3, 'A', 'B', 'C');
	system("pause");
	return 0;
}
```



Inline funkce

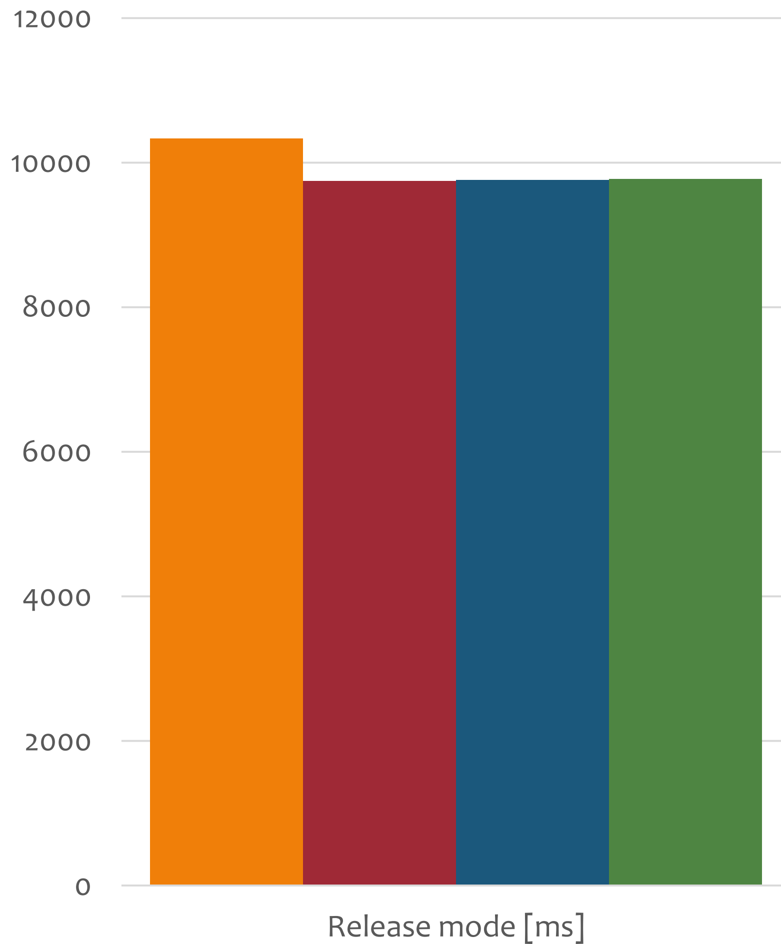
- Volání funkce stojí určitou režii. Je třeba uložit aktuální kontext, abychom byli schopni se do místa volání korektně vrátit. Pak je třeba uložit parametry a návratovou adresu na zásobník, a vygenerovat skok do podprogramu. Při návratu z funkce pak provést návrat zpět.
- Pokud není volání funkce složité, lze uvažovat o tom, že místo volání funkce se do místa volání vloží přímo její kód (jako by to bylo makro). To můžeme překladači doporučit formou tzv. `inline` funkce:

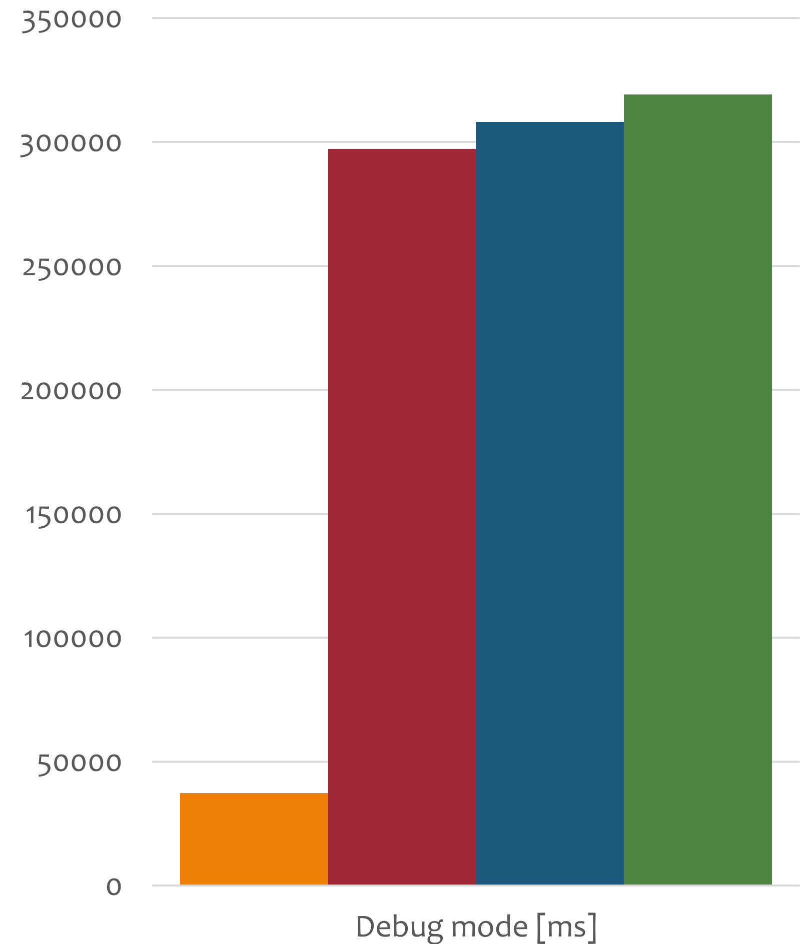
```
inline string concatenate(const string& a, const string& b) {
    return a + b;
}
```

Makro versus inline

```
#include <chrono>
#include <iostream>
using namespace std;
using ll = long long;
using clk = chrono::steady_clock;

double tmdiff(chrono::time_point<clk> t) {
    return chrono::duration<double, milli>(clk::now() - t).count();
}

#define MAX(X,Y) ((X)>(Y)?(X):(Y))
#define POCET 10000000000

inline ll max1(ll x, ll y) { return x > y ? x : y; }
ll max2(ll x, ll y) { return x > y ? x : y; }
ll max3(const ll& x, const ll& y) { return x > y ? x : y; }

int main() {
    { ll z = 0; auto t = clk::now(); for (ll i = 1; i < POCET; i++) z = MAX(z, i);
    cout << "Jako makro: " << tmdiff(t) << " ms, " << z << endl; }
    { ll z = 0; auto t = clk::now(); for (ll i = 1; i < POCET; i++) z = max1(z, i);
    cout << "Jako inline funkce: " << tmdiff(t) << " ms, " << z << endl; }
    { ll z = 0; auto t = clk::now(); for (ll i = 1; i < POCET; i++) z = max2(z, i);
    cout << "Jako funkce: " << tmdiff(t) << " ms, " << z << endl; }
    { ll z = 0; auto t = clk::now(); for (ll i = 1; i < POCET; i++) z = max3(z, i);
    cout << "Jako funkce s referencemi: " << tmdiff(t) << " ms, " << z << endl; }
    cin.get();
}
```

Makro versus inline





Parametry programu

- Funkce **main** může být bez parametrů:

```
int main(void) { … }
```

- Funkce **main** může mít parametry:

```
int main(int pocet, char *slova[]) { … }
```

  - 1. parametr udává počet slov v příkazovém řádku (slova jsou oddělena mezerou)
  - 2. parametr je seznam slov

Příklad: při vyvolání programu copy.exe

```
>copy muj.txt tvuj.txt
```

dostane funkce **main** parametry:

```
pocet == 3
slova[0] == "copy"        slova[1] == "muj.txt"
slova[2] == "tvuj.txt"    slova[3] == NULL (0)
```

Parametry programu (pokr.)

Příklad: správné řešení programu copy.exe s parametry:

```
#include <iostream>

int main(int argc, char *argv[]) {
  if (argc != 3)
    std::cout <<
      "Chybne volani copy, spravne: copy vstup vystup\n";
  else
    std::cout <<
      "Spravne volani copy ve tvaru: copy " <<
             argv[1] << " " << argv[2] << "\n";

  return 0;
}
```

Návratová hodnota funkce `main`

- Pokud skončí hlavní program úspěšně, měla by být funkce `main` zakončena příkazem: `return 0;`
- Pokud funkce `main` skončí bez tohoto příkazu, předpokládá se úspěšný konec (tj. jakoby se provedl příkaz `return 0;`). To platí jen pro funkci `main` a případně funkci, která vrací hodnotu typu `void`. Někteří autoři nedoporučují implicitní `return 0;` jako špatnou praktiku.
- Pokud vrátí funkce `main` hodnotu `0` (ať implicitně nebo explicitně), je to interpretováno jako úspěšný konec programu. Vrácená hodnota může být zpravidla prostředím dále využita.
- Jiné hodnoty než `0` mohou být interpretovány jako příznak chyby. Zaručené hodnoty jsou definovány v knihovně stdlib.h (cstdlib):

Hodnota	Popis
0	úspěšný konec
EXIT_SUCCESS	úspěšný konec
EXIT_FAILURE	neúspěšný konec, chyba

The End